LEHRSTUHL FÜR TECHNISCHE ELEKTRONIK
Friedrich-Alexander-Universität Erlangen-Nürnberg
Vorstand: Prof. Dr.-Ing. Dr.-Ing. habil. Robert Weigel


Diplomarbeit im Fach
Elektrotechnik

# Entwicklung eines hochintegrierten Demonstrators zur Winkelmessung mittels Sechstortechnologie

Thorsten Spätling

| | |
|---|---|
| Betreuer: | Dipl.-Ing. Stefan Lindner<br>Prof. Dr.-Ing. Dr.-Ing. habil. Robert Weigel |
| Beginn: | 15.10.2012 |
| Abgabetermin: | 15.04.2013 |

## Selbstständigkeitserklärung

Ich versichere, dass ich diese Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ort, Datum

Unterzeichner

# Zusammenfassung

During the lasts years some hardware setups for angle positioning have been developed at the chair. These all together have the same drawback. In order to acquire, process or visualize data they do depend on a PC system. The thesis' goal was to develop a highly integrated system that got rid of that handicap. Kontron's embedded single board computer product MSMST was chosen to reach the set aim. The product consists of a normal PC combined with a FPGA which is necessary to collect measurement data. So the core elements that are discussed are: The programming of the FPGA, the writing of a linux device driver that passes data from the FPGA to the PC system and the creation of a graphical interface in order to display the data.

Am Lehrstuhl wurden bereits mehrere Hardwareaufbauten zum Winkelortungsverfahren via Sechstortechnologie in Betrieb genommen. Bei keiner der Entwicklungen handelte es sich um ein allein stehendes System, das sowohl Datenakquirierung und -weiterverarbeitung als auch die Visualisierung komplett übernehmen konnte. Ziel dieser Arbeit war ein solches zu entwickeln. Mit dem MSMST einem Embedded Single Board Computer von Kontron fand sich eine Plattform, die alle Anforderungen erfüllte. In ihr verschmilzt ein FPGA, der zum Sammeln der Messdaten erforderlich ist, mit einem PC-System, das eine einfache Datenauswertung ermöglicht. Diese Arbeit beschäftigt daher mit der Programmierung des FPGAs, der Entwicklung eines Linux-Treibers und Applikation zur Darstellung der Messungen.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# KAPITEL 1

## Einführung

In vorangegangenen Arbeiten ([14], [13]) zum Thema Winkelschätzungsverfahren bei 24 GHz konnten funktionsfähige Versuchsaufbauten realisiert, in Betrieb genommen und stellenweise ausgebaut werden. Als Grundhardware diente das Entwicklungsboard `Altera ® DE2`, das im Grunde genommen nur aus einem FPGA und der dazugehörigen Verdrahtung besteht.

Dieser Umstand erlaubt zwar die Datenakquirierung durch den FPGA, jedoch eignet sich das Board nicht zur Darstellung bzw. Auswertung der Messwerte. Dafür war es immer von Nöten die gewonnenen Daten an einen PC zu übertragen auf dem die Weiterverarbeitung innerhalb von `Matlab` stattfand. Um möglichst alle Funktionen auf einer einzigen Plattform zu integrieren, sichtete man die am Markt befindlichen Produkte und wurde bei der Firma Kontron[1] fündig. Das gewählte Produkt `Microspace ® MSMST` entspricht einem Standard Computer, auf dem jedoch der FPGA `Arria II GX` von Altera[2] fest verbaut und mit dem PCI Express Bus verbunden ist. Die einzelnen Pins des FPGAs sind über verschiedene Schnittstellen nach außen geführt und nach Spannungsfestigkeit zusammengefasst. Von den hauptsächlich verwendeten Pins wurde eine Karte erstellt, die im Anhang A gefunden werden kann. Für die genaueren Einzelheiten sei auf die jeweiligen Kapitel verwiesen. Vorerst soll eine Gliederung des Dokument gegeben werden.

Der Einführung schließt sich ein Kapitel über die Grundlagen zum Richtschätzungsverfahren mittels Sixport-Technologie an. Die genaueren mathematischen Zusammenhänge werden dabei, soweit sie für die Erklärung des Sachverhalts nicht von

[1] http://www.kontron.com/
[2] http://www.altera.com/

Nöten sind, nicht behandelt und es sei an dieser Stelle auf die Dissertationen [12] und [20] verwiesen. Das nächste Kapitel widmet sich dem FPGA und geht dabei kurz auf die technischen Fähigkeiten, die Programmiersprache VHDL sowie die einzelnen Module ein. Wert wird hierbei vorsätzlich auf den Übergang von Analog- zu Digitalwerten gelegt, denn auf letztere fußt die restliche Signalverarbeitung.

Mit den erhobenen Daten ist bis jetzt noch nichts gewonnen. Für die weitere Verwertung müssen diese über die PCI Express (PCIe) Schnittstelle in den Computer übertragen werden. Daher ist das vierte Kapitel PCIe gewidmet. Dabei werden die einzelnen Protokollschichten besprochen. Im gleichen Atemzug indem PCIe genannt wird, muss der Begriff ***D**irect **M**emory **A**ccess* (DMA) fallen, denn diese Technik ermöglicht erst hohe Datenübertragungsraten, weshalb ihr in diesem Kapitel ein längerer Abschnitt eingeräumt ist.

All die bisher besprochenen Themen ebnen zwar den Weg der Daten in den Computer, jedoch muss auf Betriebssystemseite auch jemand bereitstehen der diese abruft. Im Allgemeinen wird dies von einem Treiber gehandhabt und von dessen internen Aufbau handelt das fünfte Kapitel. Für die Auswertung und Darstellung der Daten bedarf es mehr als nur eines Treibers und so musste noch eine Applikation geschrieben werden. Diese wird im vorletzten Kapitel besprochen, während im letzten ein Ausblick über noch umsetzbare Möglichkeiten gegeben wird.

Unglücklicherweise konnte der Demonstrator nicht komplett fertig gestellt werden, da die Plattform kurz vorm Abschluss der Arbeit zum Garantiefall wurde. Aufgrund eines herstellerseitigen Designfehlers musste jedes Mal, wenn der FPGA mit einem neuen Image bespielt werden sollte, das BIOS neu beschrieben werden. Nun sind BIOS-Chips nicht gerade für all zu viele Schreibzyklen ausgelegt, weshalb vermutet wird, dass dieser durch das häufige flashen kaputt gegangen ist.

Im Dokument finden sich an den Seitenrändern hin und wieder Büroklammern, wie die nebenstehende. Dabei handelt es sich um Verweise auf Dateien, die in das PDF-Dokument eingebettet sind. Nicht jedes PDF-Betrachtungsprogramm kann mit diesen Verweisen umgehen, deshalb finden sich die ins das Dokument eingepflegten Dateien auch auf der beiliegenden CD. Diese enthält noch weit aus mehr Daten, wie die Auflistung des CD-Inhalts im Anhang B zeigt.

Als letzter Punkt sei noch auf die Schwierigkeit hingewiesen gute Quellen zu PCIe zu finden. Die PCI-SIG (PCI ***S**pecial **I**nterest **G**roup*) gestattet nur ihren Mitgliedern den Zugriff auf die PCIe-Spezifikation und die Mitgliedschaft ist mit nicht unerheblichen, jährlichen Kosten verbunden.

# KAPITEL 2

## Grundlagen und Hardwareaufbau

Wie in der Einleitung bereits erwähnt, behandelt dieses Kapitel die physikalischen Zusammenhänge, die dem Messverfahren zugrunde liegen. Begonnen wird dabei mit dem allgemeinen Aufbau und der Zusammenschaltung der beteiligten Komponenten. Anschließend werden die naturgesetzlichen Beziehungen, auf denen das Winkeldetektionsverfahren fußt, besprochen.

## 2.1 Aufbau

Die Anordnung ist in Abbildung 2.1 schematisch angedeutet und zeigt zwei in einer Ebene im Abstand $d$ befindliche Empfangsantennen und eine Sendeantenne. Hinzukommt ein Sechstor (*Sixport*), ein Analog-Digital-Wandler (ADC) und ein FPGA. Im Lot zur Antennenebene steht die Referenzebene und liegt in der mittigen Spiegelachse. Befindet sich der Sender auf dieser Achse, trifft das Sendesignal an beiden Empfangsantennen gleichzeitig ein. Entfernt sich der Sender hingegen von der Referenzebene, so wird das Signal an einer der beiden Antennen früher eintreffen als an der anderen. Das spiegelt sich in einem Phasenunterschied wider und unter Ausnutzung dieses Zusammenhangs kann auf die Position des Senders zurück geschlossen werden. Doch bevor die mathematischen Beziehungen ausgebreitet werden, sollen noch ein paar Worte zu den einzelnen Komponenten verloren werden.

Den Anfang macht das zentrale Element, das Sechstor. Im Grunde genommen handelt es sich um einen Mischer, der das HF-Signal ins Basisband herunter mischt. Ein Umweg über eine Zwischenfrequenz muss dabei nicht genommen werden, da es sich um einen Homodyn-Mischer handelt. Das Sechstor selbst ist eine Mikrostrei-

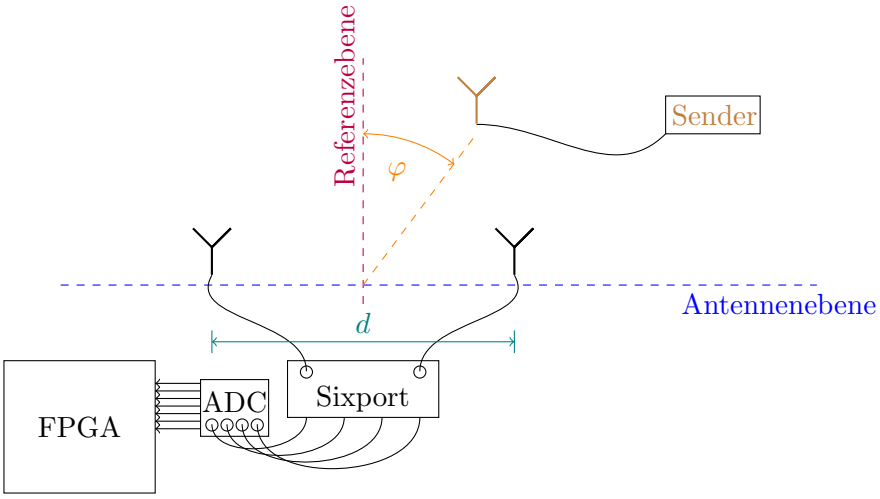



Abb. 2.1: Anordnung

fenstruktur, die – wie der Name suggeriert – sechs Ein- bzw. Ausgängen hat und je nach Beschaltung die Eingangssignale verschieden überlagert an den Ausgängen ausgibt. Detaillierte Zusammenhänge sind in [12] zu entnehmen. Damit das Ausgangssignal innerhalb der Auflösung des nachfolgenden ADCs liegt, ist es gegebenenfalls notwendig dieses vorher auf maximal 5 V zu dämpfen. Zur Wahl stehen dabei zwei Angriffspunkte. Direkt an der Antenne mittels des programmierbaren Widerstandes MCP42100 oder durch Einstellung der Diodenvorspannung, die vom Baustein TLV5626 geliefert wird.

Da das Signal nun innerhalb des quantisierbaren Wertebereichs liegt, kann der Analog-Digital-Umsetzer seinen Zweck erfüllen. Gewählt wurde dafür der Baustein `MAX1305` der Firma Maxim[1], der vier Kanäle mit einer 12-Bit Auflösung quantisiert und deren Wandlung, bei einer anliegenden Taktfrequenz von 20 MHz, nach 1,26 $\mu s$ abgeschlossen ist.

Das letzte Element, das die Abbildung 2.1 zeigt, ist der FPGA `Arria II GX` von Altera[2]. Diesem ist jedoch ein eigenes Kapitel gewidmet, weshalb hier auf dieses verwiesen werden soll.

Bisher wurde die Winkelmessung nur in einer Ebene betrachtet. Erweitert man den Aufbau um ein weiteres Sechstor, einen ADC und zwei Antennen, die in der Referenzebene der Abbildung 2.1 angebracht sind, kann der gesamte Raum erfasst werden. Um sich das leichter vorstellen zu können, werfe man einen Blick

[1] http://www.maximintegrated.com/
[2] http://www.altera.com/

auf Abbildung 2.2. Während mit einem Aufbau nur eine Aussage über $\varphi$ getroffen

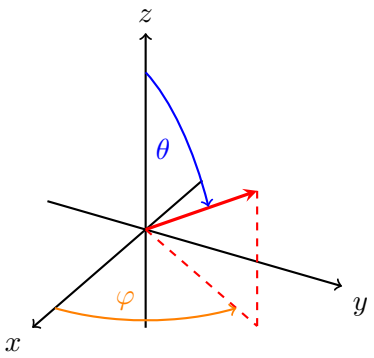



Abb. 2.2: Koordinatensystem

werden kann, ermöglicht der zweite eine über $\theta$ und diese zwei Winkel reichen für dreidimensionale Aussagen völlig aus.

Was bisher unerwähnt blieb ist die Höhe der Winkelauflösung. Die korrespondiert mit der Distanz zwischen dem Empfangsantennenpaar und wird neben anderen mathematischen Zusammenhängen im nächsten Abschnitt behandelt.

## 2.2 Sechstor

Es wurde bereits angedeutet, dass letztlich der unterschiedlich lange Signalweg zur Winkelbestimmung ausgenutzt wird. In Abbildung 2.3 ist dies anschaulich aufbereitet. Sie zeigt eine Sendeantenne von der konzentrisch Kreissegmente in

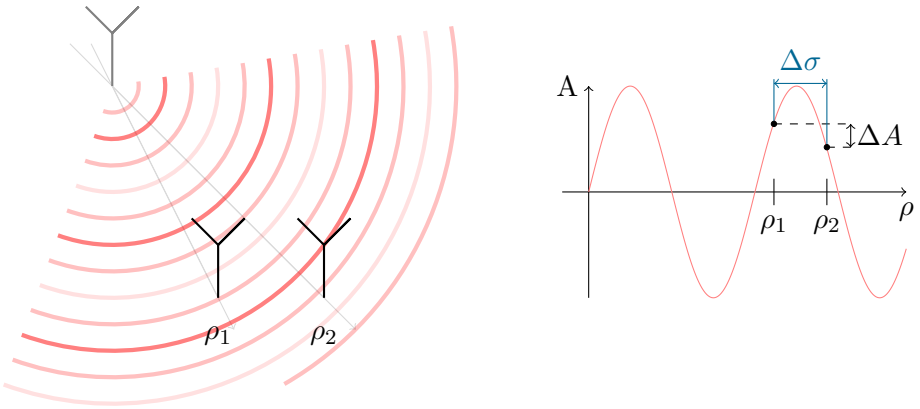



Abb. 2.3: Phasendifferenz

Richtung der Empfangsantennen ausgehen. Der Farbton von diesen entspricht dabei der Signalleistung. Die Empfangsantennen stehen jeweils verschieden weit vom

Sender entfernt und so sehen sie unterschiedliche Signalamplituden. Diese Diskrepanz kann auch als Phasendifferenz $\Delta\sigma$ ausgedrückt werden.

Um Missverständnissen vorzubeugen sei noch angemerkt, dass die Sendeantenne im Dauerstrichbetrieb (*cw*-Betrieb) sendet und ein einfaches Sinussignal emittiert. Die gesamte Anordnung kann somit als stationär betrachtet werden.

Ein wichtiger Punkt ist in der Abbildung 2.3 bereits erkennbar, jedoch noch nicht ausgearbeitet. Er betrifft die Periodizität des Sendesignals, welche einen maßgeblichen Einfluss auf die Winkelauflösung hat. Denn aufgrund dieser kann nicht unterschieden werden, ob die eine Welle der anderen um beispielsweise 18° oder um $18° + (n \cdot 360°)$ nacheilt. Um die Winkelauflösung zu bestimmen bedient man sich der Gleichung (2.1).

$$\theta_{max} = \frac{\lambda}{d} \cdot \frac{360°}{2\pi} \tag{2.1}$$

Dabei entspricht $\lambda$ der Wellenlänge des Sendesignals und $d$ ist die Distanz zwischen dem Empfangsantennenpaar. Der zweite Faktor wandelt den Winkel vom Bogenmaß ins Winkelmaß um. Für die Herleitung der Gleichung sei auf [19] verwiesen. Wichtig ist hier nur die Erkenntnis, dass man entweder eine hohe Auflösung oder eine große Winkelabdeckung haben kann. Um beides zu erreichen, benötigt man für eine Ebene zwei Aufbauten. Durch eine geschickte Wahl der Abstände, wie in [19] ebenfalls gezeigt wird, und durch Überlagerung der berechneten Winkel lässt sich ein Bereich von ±80° genau auflösen.

Das vorletzte Puzzlestück besteht in der Rekonstruktion der Phasendifferenz aus den Sechstorwerten. Zur Erinnerung, aus den zwei Eingangssignalen werden durch Überlagerung und Verzögerung vier unterschiedliche Ausgangssignale generiert. Zusammen beschreiben sie einen komplexen Zeiger, der sich als Real- und Imaginärteil oder in Betrag und Phase schreiben lässt. Um das Argument (die Phase) des Zeigers zu erhalten, bildet man den Quotienten aus Imaginär- und Realteil und berechnet davon den Arkustangens, wie Gleichung (2.2) zeigt.

$$\Delta\sigma = \arctan\left(\frac{\Im\{z\}}{\Re\{z\}}\right) \tag{2.2}$$

Die mathematische Herleitung findet sich in [13] weshalb hier auf sie verzichtet wird.

Von der Phasendifferenz zum Winkel ist es nur noch ein kleiner Schritt, der anhand Abbildung 2.4 verdeutlicht werden soll. Über geometrische Betrachtungen ist schnell ersichtlich, dass der Winkel, der zwischen dem Sender und der Referenzebene aufgespannt wird, auch zwischen der Empfangsebene und den Wellenfronten zu

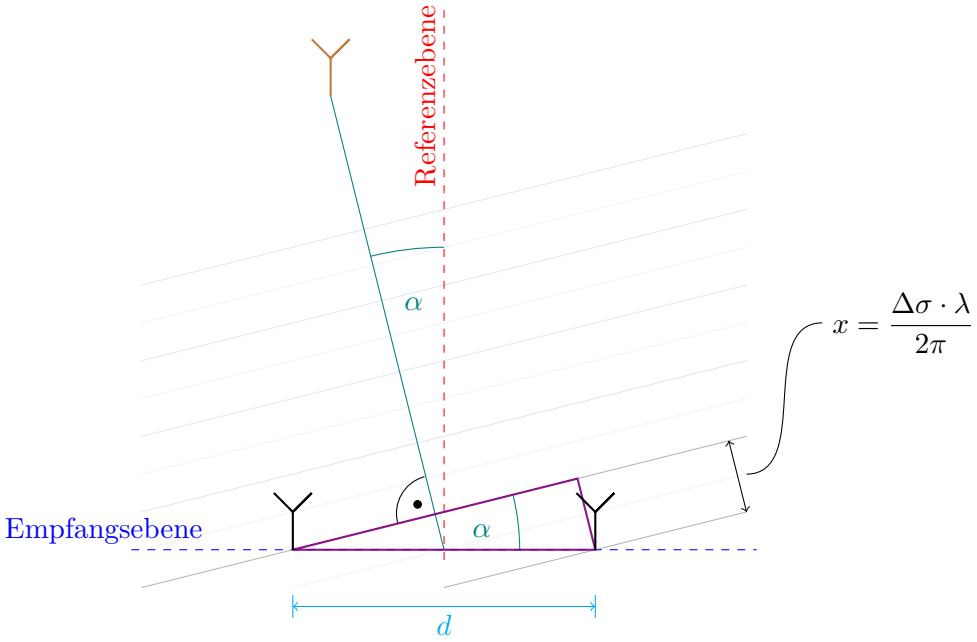



Abb. 2.4: Geometrische Zusammenhänge der Winkelmessung

finden ist. Das Signal muss, um die rechte Antenne zu erreichen, einen zusätzlichen Weg zurücklegen, der von Gleichung (2.3) erfasst wird.

$$x = \frac{\Delta\sigma \cdot \lambda}{2\pi} \tag{2.3}$$

$\Delta\sigma$ entspricht weiterhin der Phasendifferenz zwischen den an den Antennen ankommenden Signalen und $\lambda$ ist die Wellenlänge des Sendesignals. Mit dem gewählten Antennenabstand $d$ sind nun genügend Größen des violetten Dreiecks, das zwischen den Empfangsantennen und der Wellenfront aufgespannt wird, bekannt. Die Verwendung der Winkelfunktion Sinus liefert die in Gleichung (2.4) angegebene Beziehung, aus der sich zu guter Letzt der Einfallswinkel bestimmen lässt.

$$\sin(\alpha) = \frac{\Delta\sigma \cdot \lambda}{2\pi \cdot d} \tag{2.4}$$

$$\alpha = \arcsin\left(\frac{\Delta\sigma \cdot \lambda}{2\pi \cdot d}\right) \tag{2.5}$$

Da jetzt alle physikalischen Zusammenhänge erklärt sind, wird im nächsten Kapitel gezeigt wie die Analogsignale ausgelesen und verarbeitet werden.

# KAPITEL 3

# FPGA

Vor Jahrzehnten bestand die Hardwareentwicklung aus dem Aufbau von Schaltungen mittels diskreter Bauteile. Die immer weiter voranschreitende Verkleinerung und zunehmende Integration machte diesen Ansatz nicht mehr handhabbar. Die Entwicklung verlagerte sich vom Steckbrett in den Computer und aus den diskreten Bauteilen wurden vielschichtige, Nanometer große, anwendungsspezifische, integrierte Schaltkreise (ASIC, ***A****pplication-****S****pecific* ***I****ntegrated* ***C****ircuit*). Der Vorteil besteht in der Packungsdichte, die die Bauteilgröße letztlich drastisch reduziert, dieser wird jedoch mit der Unveränderlichkeit der Logikanordnung erkauft, was die Kosten für Fehler in die Höhe treibt.

Abhilfe schaffen hier programmierbare Logikbausteine. Die bekanntesten Vertreter sind CPLD (*complex programmable logic device*) und FPGA (*field programmable gate array*). Beiden ist die variable Konfiguration der inneren Verknüpfungen gemeinsam, jedoch unterscheiden sie sich in den Durchlaufzeiten, die beim CPLD exakter vorhergesagt werden können. Ein weiterer wichtiger Unterschied ist, dass CPLDs über weniger Flipflops (Speicherbits) verfügen und sich daher kaum für Schieberegister oder Zähler eignen. Ihr vornehmliches Einsatzgebiet bilden daher rein kombinatorische Schaltungen.

Die vorgesehene Anwendung erfordert eine höhere Komplexität als die Verschaltung von einzelnen Pins über UND- oder ODER-Gattern und es bedarf daher eines FPGAs.

In den folgenden Abschnitten werden einige Eigenschaften des FPGAs, dessen generelle Programmierung in der Programmiersprache VHDL und die für die Ar-

beit geschriebenen Module behandelt. Letztere sind in das Dokument eingebettet und können über die Büroklammern extrahiert werden.

## 3.1 Eigenschaften

Neben einfachen Logikverknüpfungen können mit einem FPGA auch wesentlich komplexere Anforderungen umgesetzt werden. Jeder Pin ist, soweit dieser keine Sonderfunktion erfüllt, mit einem Logikblock verknüpft, der aus einem Flipflop und einer vorgelagerten LUT (***l**ook-**u**p-**t**able*) besteht.

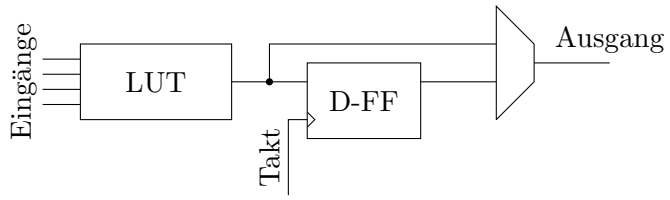



Abb. 3.1: Logikblock

Je nach Programmierung werden die Interna der Nachschlagetabelle (LUT) und die Verknüpfung zu anderen LUTs so verschaltet, dass die gewünschte Funktion abgebildet wird. Zusätzlich dazu warten aktuelle FPGAs mit spezialisierten, kaum veränderlichen, Kernen auf, den sogenannten *hard cores*. In den meisten Fällen werden sie als *ip cores* bezeichnet, wobei ip für *intellectual property* steht.

Die Kerne werden i.d.R. mit dem Erwerb von Lizenzen freigeschaltet. Neben den lizenzpflichtigen werden auch quelloffene, für gewöhnlich lizenzfreie Kerne angeboten. Eine gute Anlaufstelle bildet die Plattform `http://opencores.org/`, wobei es sich hierbei um *soft cores* handelt.

**hard core** Hersteller moderner FPGAs integrieren in ihren Modellen fertige Schaltkreise hoher Komplexität. Darunter fallen z.B. PCI(e) Controller, Netzwerkkarten oder ganze Mikroprozessoren. Die *hard cores* werden beim Fertigen fest platziert und sind im Allgemeinen konfigurierbar, in ihrer Struktur jedoch unveränderlich. Dafür benötigen sie weniger Chipfläche und können mit einem höheren Takt laufen.

**soft core** Neben den harten, innerlich fixierten Kernen gibt es noch veränderliche, sogenannte *soft cores*. Dabei handelt es sich ebenso um vorgefertigte Schaltkreise, die genauso komplexe Anforderungen erfüllen wie *hard cores*, jedoch sind diese unabhängig vom eingesetzten FPGA und komplett adaptierbar.

Der große Vorteil bei der Verwendung von vorgefertigten Kernen ist, neben der einfachen Benutzung, die Gewissheit, dass diese bereits ausgiebig getestet wurden und daher funktionstüchtig sind. Weiterhin ergibt sich neben der Zeit- durchaus eine Kostenersparnis, da z.B. für die PCI Express Spezifikationen jährliche Mitgliedsbeiträge[1] in Höhe von 3.000 $ zu erstatten sind.

Die Mankos sind sicherlich eine erschwerte Fehlersuche, ein geringeres Detailverständnis und vor allem die Abhängigkeit vom Anbieter des Kerns.

Neben den Kernen sind FPGAs noch mit andere Komponenten ausgestattet.

- Taktgeneratoren erzeugen aus dem eingespeisten Taktsignal mittels einer Phasenregelschleife (*phase-locked loop*, PLL) oder einer DLL (*delay-locked loop*) durch Taktteilung bzw. -vervielfachung weitere Taktsignale mit der vorher festgelegten Frequenz.

- Speicherblöcke

- Ein-/Ausgangsblöcke deren Spannungslevel an die Begebenheiten der Außenwelt (TLL, CMOS, LVDS, ...) anpassbar sind.

- Digitale Filter und extra Multipliziereinheiten.

Im bisherigen Abschnitt wurde ein grobes Verständnis über den vereinfachten Aufbau und die Fähigkeiten eines FPGAs vermittelt. Um diesen programmieren zu können, widmet sich der nächste Abschnitt einer der dafür entworfenen Programmiersprache.

## 3.2 VHDL

Für die Hardwareentwicklung bieten sich mehrere Hardwarebeschreibungssprachen (***H**ardware **D**escription **L**anguage*) an. Da wären ABEL (***A**dvanced **B**oolean **E**quation **L**anguage*), AHDL (Altera HDL), GHDL (Genrad HDL), Verilog und VHDL (***V**ery high speed integrated circuit* HDL). Von den genannten sind die letzten beiden die Platzhirsche. Laut Hörensagen findet Verilog vornehmlich in den USA Verwendung während im europäischen Raum VHDL den Vorzug genießt.

In ihrer Mächtigkeit unterscheiden sie sich nicht, da jedoch am Lehrstuhl hauptsächlich in VHDL programmiert wird, sind auch alle während der Arbeit entstandenen Module in dieser Sprache geschrieben worden.

[1] http://www.pcisig.com/

### Sprachelemente

VHDL besitzt 40 Konstrukte von denen nur ein paar genutzt wurden. Dazu zählen in erster Linie Kontrollstrukturen, Schleifen und Zuweisungen. Die Bedeutung der einzelnen Befehle kann in den meisten Fällen der Bezeichnung selbst entnommen werden und wo es nicht gelingt, wird die Funktion – wie z.B. in Tabelle 3.1 – erklärt. Ehe auf die Erstellungen eines Moduls eingegangen wird, soll noch darauf

| | Beispiel | Beschreibung |
|---|---|---|
| <= | sig1 <= sig2; | Signal 1 wird der Wert von Signal 2 zugewiesen. |
| := | sig3 := '0'; | Signal 3 wird mit dem Wert '0' initialisiert. |
| => | bez1 : entity foo port map (<br>in1 => sig4,<br>...<br>); | Die Entität foo wird eingebunden und deren Eingang in1 wird fest mit dem Signal 4 verknüpft. |

Tab. 3.1: Zuweisungsoperatoren

hingewiesen werden, dass die Programmiersprache nicht zwischen Groß- und Kleinschreibung unterscheidet und dass zusammengehörige Blöcke nicht durch Klammern eingefasst werden.

### Modulaufbau

Bei den ersten Hochsprachen handelte es sich um imperative, d.h. in den Programmen wurde schlicht, die Reihenfolge in der die einzelnen Befehle abgearbeitet werden, vorgegeben. In den später entstandenen objektorientierten Sprachen wurde vor allem auf die Wiederverwendbarkeit und die Kapselung der Daten Wert gelegt. In VHDL findet sich das Konzept der OOP wieder, was in den folgenden Ausführungen ersichtlich wird.

In Programmiersprachen schreibt man normalerweise Funktionen, in Hardwarebeschreibungssprachen definiert man hingegen Module. Diese werden als Entität (*entity*) bezeichnet und man kann sie sich als eine Blackbox mit Ein- und Ausgängen vorstellen, wie in Abbildung 3.2 zu sehen ist.

Es ist zwar zu sehen, dass die Box vier Eingänge und einen Ausgang hat, was jedoch im Inneren passiert ist nicht ersichtlich. Definiert wird solch eine Box wie es im Codebeispiel 3.1 gezeigt wird. Das Schlüsselwort `entity` gefolgt vom Namen und `is` leitet die neue Entität ein, die nur aus der Definition der Anschlüsse besteht und mit `end` und dem eingeführten Namen abgeschlossen wird. Neben den Richtungen `in` und `out` gibt es noch `inout` für bidirektionale Signale. Normalerweise wird

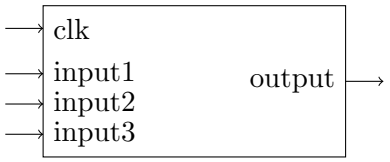



Abb. 3.2: Entität

jeder Befehl mit einem Semikolon abgeschlossen, die Ausnahme macht hier die Zeile 5, da in der darauffolgenden Zeile das übergeordnete Kommando `port` schließt. Die Regel wann ein Semikolon zu setzen ist sind zugegebenermaßen auf den ersten Blick verwirrend, der Compiler wirft jedoch bei fehlenden oder falsch gesetzten Semikola aussagekräftige Fehlermeldungen. Apropos Compiler, wie in allen anderen Sprachen muss auch bei VHDL dem Compiler mitgeteilt werden, welche Bibliotheken er einbinden soll. Das ist in VHDL die `ieee` Bibliothek, die mittels `library ieee;` angekündigt und durch den Aufruf `use ieee.std_logic_1164.all;` verwendet wird. Erst danach sind die Datentypen `std_logic` und `std_logic_vector` bekannt.

```
1 entity Bausteinname is
2   port (
3     clk                    : in  std_logic;
4     input1,input2,input3   : in  std_logic;
5     output                 : out std_logic
6   );
7 end Bausteinname;
```

Listing 3.1: Grundgerüst einer Entität

Nachdem das Grundgerüst steht, müssen noch die inneren Zusammenhänge festgelegt werden. Für das oben gewählte Beispiel wurde die in Abbildung 3.3 dargestellte Verschaltung gewählt. Die zwei angegebenen Varianten unterscheiden sich nicht in ihrer Funktion, da die eine Variante aus der anderen unter Anwendung der De Morganschen Regeln entstanden ist.


(a) UND Gatter (b) ODER Gatter

Abb. 3.3: Architektur der Entität

Beiden ist gemeinsam, dass der dritte Eingang nicht verwendet wird und dass der Takt zum jeweiligen Logikgatter weitergereicht wird. Folglich wird dieses mit jedem Takt den Ausgang in Abhängigkeit von seinen Eingängen ändern. Das nachfolgende Codebeispiel 3.2 generiert die Variante (a) und wird nun genauer erläutert.

Die erste Zeile kündigt einen Block an, der das innere Zusammenspiel der Entität mit dem Namen Bausteinname beschriebt. Prinzipiell ist es möglich beliebig viele Architekturen für ein und die selbe Entität zu entwerfen, sie müssen sich nur im Bezeichner (Innenleben) unterscheiden. In den nächsten zwei Zeilen werden die intern verwendeten Signale definiert. Beim einen handelt es sich um einen Vektor der Länge 3 und beim anderen um einen 1-Bit Datentyp. Mit `begin` in Zeile 5 werden die Signal Definitionen abgeschlossen und die eigentlichen Signalzuweisungen werden getroffen. Diese kann dabei an Bedingungen geknüpft werden oder statischer Natur sein. Die Beziehungen zwischen den Bausteineingängen (input1,input2) und dem Zwischensignal input entsprechen letzterem und die Aktualisierung des Bausteinausgangs (output) wird nur vollzogen, wenn eine steigende Flanke im Taktsignal detektiert wird. Die ereignisgesteuerte Behandlung wird mit einem Unterprozess erzwungen, der mit `process begin` eingeleitet und mit `end process` geschlossen wird. Es ist prinzipiell möglich beliebig viele Unterprozesse zu erstellen, nur darf ein Signal niemals in zwei Unterprozessen verwendet werden.

```
1  architecture Innenleben of Bausteinname is
2    signal input  : std_logic_vector (2 downto 0);
3    signal clock  : std_logic := '0';
4
5    begin
6      input(0) <= input1;
7      input(1) <= input2;
8
9    process begin
10     wait until rising_edge(clk);
11     output <= (input(0) and (not input(1)));
12   end process;
13 end Innenleben;
```

Listing 3.2: Architektur der Entität

Im Beispiel wurden unnötige oder überdimensionierte Signale eingeführt, was kein Problem darstellt, da diese später vom Compiler verworfen bzw. angepasst werden. Auf eine wichtige Eigenheit sei noch hingewiesen. Eingänge können immer nur gelesen und Ausgänge können nur beschrieben werden.

### Simulation

Das Modul ist geschrieben, allerdings ist bis jetzt noch nicht gewiss, dass es funktioniert. Dafür bietet VHDL die Möglichkeit eigene Simulationsumgebungen (*testbench*), wie sie im Codebeispiel 3.3 gezeigt wird, zu erstellen.

Auch für die Simulation müssen erst einmal die notwendigen Bibliotheken eingebunden werden, damit die Datentypen bekannt sind. Danach signalisiert eine leere Entität, dass es sich um eine Testumgebung handelt, die im Anschluss mit Leben gefüllt wird. Dazu müssen so viele Signale definiert werden, wie die zu testende Entität Ein- und Ausgänge hat, denn im nächsten Schritt werden sie diesen zugeordnet. Der Bezeichner beim Zuordnungsblock ist zwingend zu setzen!

In den letzten Zeilen werden die eigentlichen Signalwerte festgelegt. Der Takt wird hier beispielhaft nach einer Nanosekunde invertiert und die zwei Eingänge nehmen die vier möglichen Zustände an.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity Test_tb is
5  end Test_tb;
6
7  architecture tb1 of Test_tb is
8
9  signal in1,in2,in3,clock : std_logic := '0';
10 signal out1              : std_logic := '0';
11
12 begin
13
14   Bezeichner : entity work.Bausteinname
15   port map(
16     clk    => clock,
17     input1 => in1,
18     input2 => in2,
19     input3 => in3,
20     output => out1
21   );
22
23   clock <= not clock after 1 ns;
24
25   in1 <= '0', '1' after 50 ns;
```

```
26    in2 <= '0', '1' after 10 ns,
27                '0' after 20 ns,
28                '1' after 50 ns,
29                '0' after 80 ns;
30  end tb1;
```

Listing 3.3: Testbench für das Modul

VHDL Module können – wie in jeder anderen Programmiersprache auch – in einem gewöhnlichen Texteditor verfasst werden. Um den Code für die Hardware zu übersetzen ist man jedoch an die Entwicklungsumgebung des Herstellers gebunden. Denn nur in diesen sind alle Hardwaredetails hinterlegt. Eine Einführung in die Entwicklungsumgebung Quartus II von Altera würde viel zu weit führen. Es sei daher auf die (Video-)Tutorials auf der Herstellerhomepage[2] verwiesen, die einen guten Einstieg bieten.

Für die Simulation des einfachen Beispiels kann man jedoch bedenkenlos auf die Open Source Programme GHDL[3] und GTKWave[4] zurückgreifen. GHDL ist hierbei nicht mit der eingangs erwähnten Hardwarebeschreibungssprache Genrad HDL zu verwechseln! Bei GHDL handelt es sich um einen VHDL-Compiler, der den VHDL-Code in eine ausführbare Anwendung übersetzt, diese startet und mit den Simulationswerten aus dem Testbench speist. Die Simulationsergebnisse können anschließend mit GTKWave betrachtet werden, wie in Abbildung 3.4 gezeigt ist.

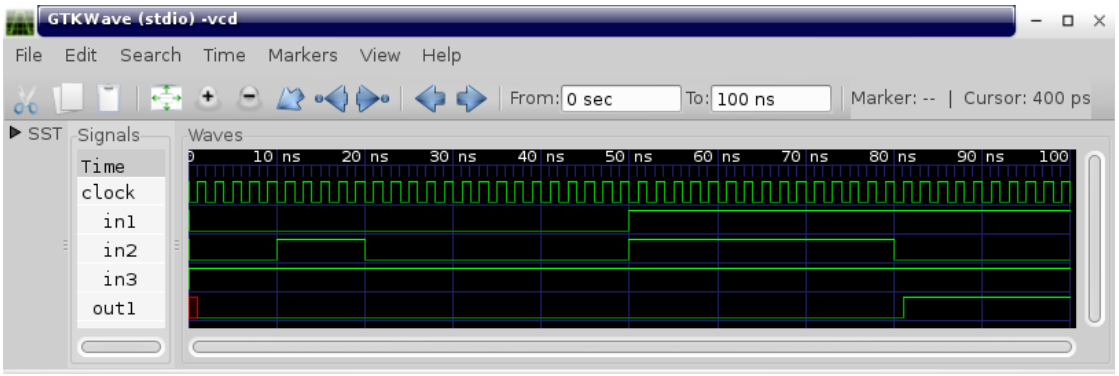



Abb. 3.4: GTKWave

Der Weg dorthin wurde – wie unter Linux üblich – über ein Makefile vereinfacht, das in allgemeiner Form durch Klick auf die nebenstehende Heftklammer bezogen werden kann. Für die Anzeige muss es dreimal aufgerufen werden, wie es das Li-

[2] http://www.altera.com/education/training/curriculum/fpga/trn-fpga.html
[3] http://ghdl.free.fr/
[4] http://gtkwave.sourceforge.net/

sting 3.4 zeigt.

```
$ make ghdl-compile
$ make ghdl-run
$ make ghdl-view
```

Listing 3.4: Starten der Simulation

## 3.3 Module

Im vorherigen Abschnitt wurden die VHDL-Grundlagen zum Verständnis des Gesamtsystems gelegt, das in Abbildung 3.5 vereinfacht dargestellt ist. Das zentrale Element ist das `SYS`-Modul, das alle anderen Module miteinander verknüpft. Angesteuert wird es über die PCIe Schnittstelle, was in der Treiber- und Applikationsbesprechung genauer erläutert wird.

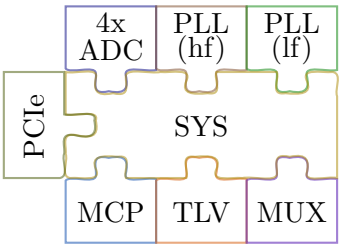



Abb. 3.5: Modulübersicht

Mit den PLL-Modulen wird auf die Phasenregelschleifen des FPGAs zurückgegriffen, die aus dem PCIe-Takt von 100 MHz die benötigten Taktraten ableiten. Die vier ADC-Module stoßen die Abtastung von jeweils vier Messkanälen an und liefern die Ergebnisse an die PCIe-Schnittstelle. Im Modul MUX wird die Wiederholfrequenz, mit der die Messungen angestoßen werden, selektiert. Zur Auswahl stehen die von PLL (lf) generierten Frequenzen, die im Bereich zwischen 10 und 100 kHz liegen, während PLL (hf) Systemtakte im zweistelligen Megahertzbereich liefert. Die Module MCP und TLV kommunizieren beide via SPI (*serial peripheral interface*) über jeweils eigene Leitungen mit ihren jeweiligen Bausteinen und konfigurieren diese je nach Bedarf. Das PCIe-Modul bildet, wie bereits in der Einleitung erwähnt, die Schnittstelle zum Computer, der für die Datenauswertung zuständig ist.

Der VHDL-Quellcode zu den einzelnen Modulen findet sich in den zugehörigen Unterlagen und entspricht in den meisten Fällen einem Zustandsautomaten (*state*

*machine*), der einmal angestoßen seine Zustände abläuft und zum Schluss wieder im Wartezustand verharrt. Zusätzlich gibt es zu allen Modulen eine passende Testumgebung, die das gewünschte Verhalten zeigt und das Verständnis erleichtert.

Damit die einzelnen Puzzlestücke richtig zusammenstecken, werden sie in einer sogenannten Top-Entität eingebunden und miteinander verknüpft. Zur Kennzeichnung benennt man die zugehörige VHDL-Datei nach dem Projekt mit dem Suffix _TOP.

### MCP

Beim Baustein MCP42100 handelt es sich um ein digitales Potentiometer mit zwei Ausgangskanälen, das über *SPI* gesteuert wird. Weiterhin unterstützt der Chip *Daisy Chaining*, worunter man eine Reihenschaltung von mehreren Komponenten versteht. Dabei reicht die erste Komponente die empfangenen Daten an die nächste

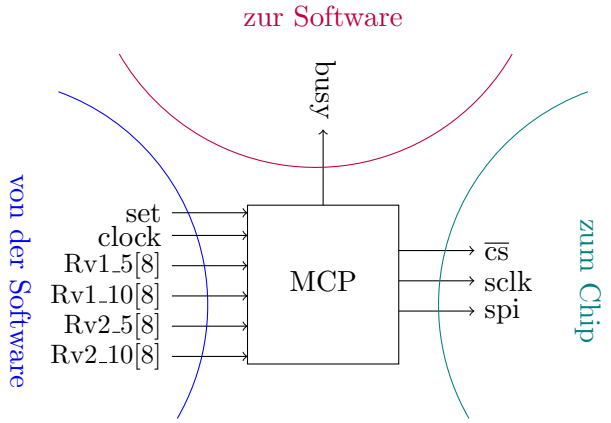



Abb. 3.6: Modul MCP

weiter. Allerdings nur so lange ein Takt anliegt! Angenommen in einer Kette soll dem dritten Glied ein 8 Bit Befehl gesendet werden, so wird das Kommando mit den ersten acht Takten zum ersten Kettenglied, mit den nächsten zum zweiten und mit den Takten 17 bis 24 zum dritten übertragen.

Für die Konfiguration wird an den Baustein ein Befehls- und ein Datenbyte gesendet. Über ersteres wird Kanal Chip ausgewählt und mit letzterem wird der Potentiometerwert gesetzt. Auf dem Board sind zwei Chips via *Daisy Chaining* verbunden weshalb das Modul – wie in Abbildung 3.6 zu sehen ist – vier Werte entgegen nimmt. In Folge dessen werden abwechselnd über SPI die insgesamt vier Potentiometerwerte eingestellt. Während dieses Vorgangs, der durch einen positiven Flankenwechsel im Signal `set` angestoßen wurde, zeigt `busy`, indem es *high*

wird, die Aktivität des Moduls an. Die Dauer der Operationen wird maßgeblich vom anliegenden Takt bestimmt.

### TLV

Der Chip TLV5625 ist ein Digital-Analog-Wandler mit zwei Ausgangskanälen und einer 8 Bit Auflösung. Betrieben werden kann er mit einer internen oder einer

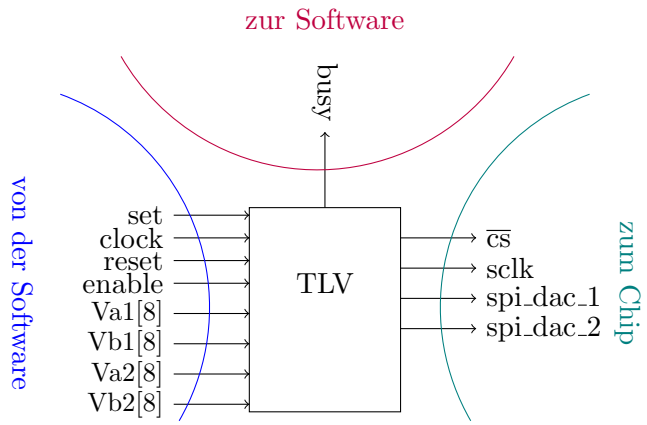



Abb. 3.7: Modul TLV

externen Referenzspannung. Angesprochen wird er über SPI, ist allerdings nicht *Daisy Chaining* fähig. Das VHDL-Modul besitzt deswegen, wie Abbildung 3.7 zeigt, zwei SPI-Ausgänge, schließlich müssen die zwei Bausteine pro Sechstor konfiguriert werden. Da mit beiden gleichzeitig geredet wird, genügen eine Takt- und eine Selektionsleitung.

Beim Starten stellt das Modul den Chip automatisch von externer auf interne Referenz um. Im Anschluss können die *DACs* auf die übergebenen Werte gesetzt werden, indem an `set` ein positiver Flankenwechsel erzeugt wird. Ist `enable` dabei *low*, wird der Chip jedoch heruntergefahren. So lange `reset` *high* ist, verweilt das Modul im Reset-Zustand und wartet auf eine negative Flanke an `reset`. Daraufhin verhält es sich wie beim Hochfahren. Es wird die Spannungsreferenz umgestellt und auf weitere Anweisungen gewartet. Für die Dauer der Operationen ist auch bei diesem Modul `busy` *high* und sie stehen in direkter Abhängigkeit zur angelegten Taktfrequenz.

### ADC

Das Modul für den ADC `MAX1305` ist ein wenig komplexer als die vorherigen, da in beide Richtungen kommuniziert wird. Außerdem benötigt es zwei Taktleitungen wovon die eine (`clock_20MHz`) den ADC taktet und die andere (`clock`) das Modul selbst betreibt. Dabei muss `clock` eine deutlich höhere Frequenz aufweisen als `clock_20MHz`, wenn eine zeitnahe Reaktion gewünscht ist. Der Grund dafür ist in der Detektion der positiven Flanke zu finden, die mit Hilfe des Codeauszugs 3.5 erklärt wird.

```
1  architecture v1 of max1305 is
2    signal insr_get_data   : std_logic_vector (2 downto 0);
3    signal get_data_is_high : std_logic := '0';
4    (..)
5
6    begin
7
8  --using 20MHz from top design:
9    sclk <= clock_20MHz;
10
11   (..)
12
13   UpdateGetData : process begin
14     wait until rising_edge(clock);
15     insr_get_data <= insr_get_data(1 downto 0) & get_data;
16   end process;
17
18   GetData_Edges : process begin
19     wait until rising_edge(clock);
20     if( insr_get_data(2 downto 1) = "10") then --falling
21       get_data_is_high <= '0';                 --edge
22     end if;
23     if( insr_get_data(2 downto 1) = "01") then --rising
24       get_data_is_high <= '1';                 --edge
25     end if;
26   end process;
27
28   (..)
29
30 end v1;
```

Listing 3.5: Auszug aus dem *ADC*-Modul

Der Prozess `UpdateGetData` verschiebt mit jeder steigenden Flanke im Signal `clock` den Inhalt des Schieberegisters `insr_get_data` um eine Position nach links und an die frei gewordene Stelle tritt der Wert des Signals `get_data`. Der zweite Prozess `GetData_Edges` sucht zeitgleich nach einem Flankenwechsel indem er die Positionen 2 und 1 des Schieberegisters mit „10“ bzw. „01“ vergleicht. Der Begriff zeitgleich darf dabei nicht wortwörtlich verstanden werden. Der eigentliche Ablauf wird mit Blick auf Abbildung 3.8 deutlich.

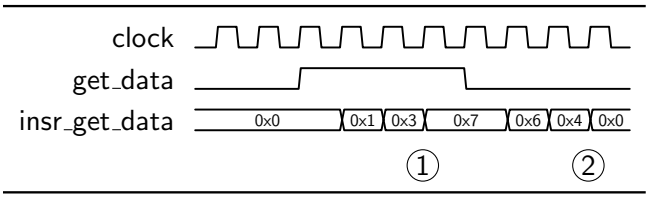



Abb. 3.8: Flankendetektion

Das Schieberegister enthält anfangs Nullen und ändert seinen Wert nicht sofort mit dem Erkennen eines *high* Pegels im Signal `get_data`, sondern erst mit dem nächsten Takt. Dann bedarf es eines Taktes der die eins im Register an die Position verschiebt, die im Prozess `GetData_Edges` untersucht wird, und eines weiteren bis dieser das interne Signal `get_data_is_high` auf eins gesetzt hat, was zum Zeitpunkt ① geschieht. Der im Codeausschnitt nicht mehr gezeigte Zustandsautomat benötigt ebenso einen Takt bis der gewünschte Taktzyklus angestoßen wird. Eine Reaktion ist somit erst nach fünf Takten sichtbar und ein schnelleres Ansprechen kann nur durch Steigerung der Betriebsfrequenz erreicht werden. Da zum Zeitpunkt ② `get_data_is_high` auf Null zurückgesetzt wird muss bis spätestens dahin der Zustandsautomat angestoßen worden sein.

Sobald der Automat jedoch läuft, wird die Konvertierung der vier ADC-Kanäle gestartet. Dazu wird `convst` für etwa 0,1 µs *low* gehalten und mit der steigenden Flanke beginnt der ADC mit dem Abtasten. Durch auf *low* setzen der $\overline{\text{eoc}}$ Leitung signalisiert der Chip, den Abschluss einer Konvertierung, die im Anschluss über die zwölf `data_in` Pins gelesen wird. Nach Beendigung der vierten Wandlung zeigt die ***e**nd **o**f **l**ast **c**onversion* das gleiche Verhalten und erlaubt eine Synchronisierung. Auch dieses Modul zeigt über `busy` seine Aktivität an und sobald diese endet, stehen die Werte an `ch1..4` zur Verfügung.

### PLL

Die Quartus II Entwicklungsumgebung bietet mit dem MegaWizard Plug-In Manager eine einfache Möglichkeit zur Erstellung eines PLL Moduls. Dazu muss man

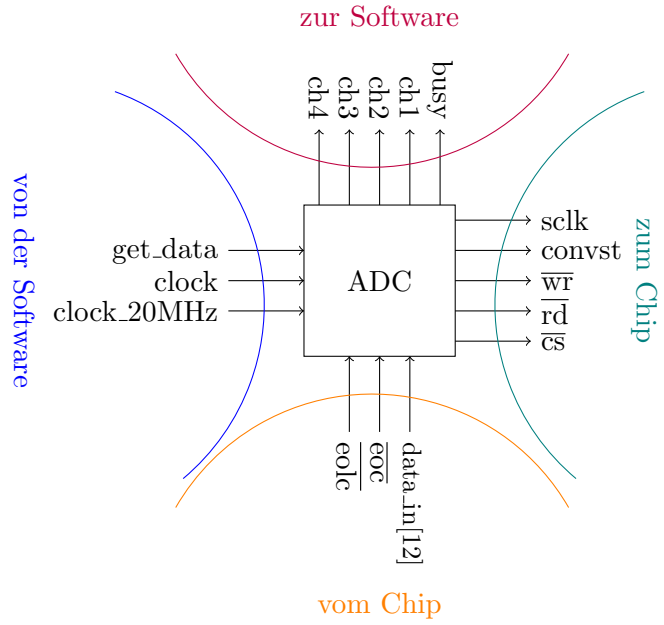



Abb. 3.9: Modul ADC

nur dessen Anweisungen folgen, die Eingangs- so wie alle Ausgangsfrequenzen festlegen und der Rest geschieht von selbst. Am Ende hat man einen VHDL-Baustein, wie ihn Abbildung 3.10 zeigt, dem nur das bekannte Taktsignal zugeführt werden muss.

### PCIe

Neben dem schon erwähnten MegaWizard Plug-In Manager bietet Quartus II noch weitere grafische Helfer zur Modulerstellung. Einer davon ist der SOPC Builder mit dem die PCIe Einheit entworfen wurde. Da dem PCIe Bus ein eigenes Kapitel gewidmet ist, wird in den folgenden Zeilen nur kurz dargelegt wie man Änderungen vornimmt und worauf man dabei achten muss. Bei dem in der Abbildung 3.11 gezeigten Modul, handelt es sich um eine stark vereinfachte Darstellung, die jedoch für das Verständnis völlig ausreicht. Für die Details sei auf den Quellcode der Top-Entität und auf die Herstellerdokumentation [7] verwiesen.

Auf der linken Seite sind vier Signale gezeigt, die in die PCIe Einheit geschrieben werden. Dabei handelt es sich um die von jedem Sechstor erzeugten vier 12 Bit-Werte. Die rechte Seite bildet das Gegenstück und erlaubt das Lesen der notierten Signale. Ihrer Benennung ist zu entnehmen, dass das erste Signal an das MCP-Modul und die nächsten beiden an das TLV-Modul weitergereicht werden. Das

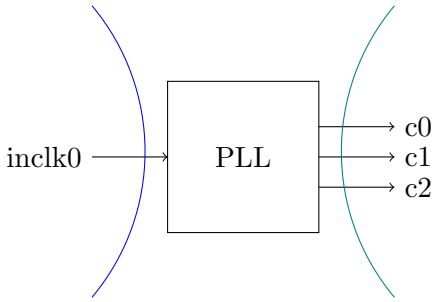



Abb. 3.10: Modul PLL

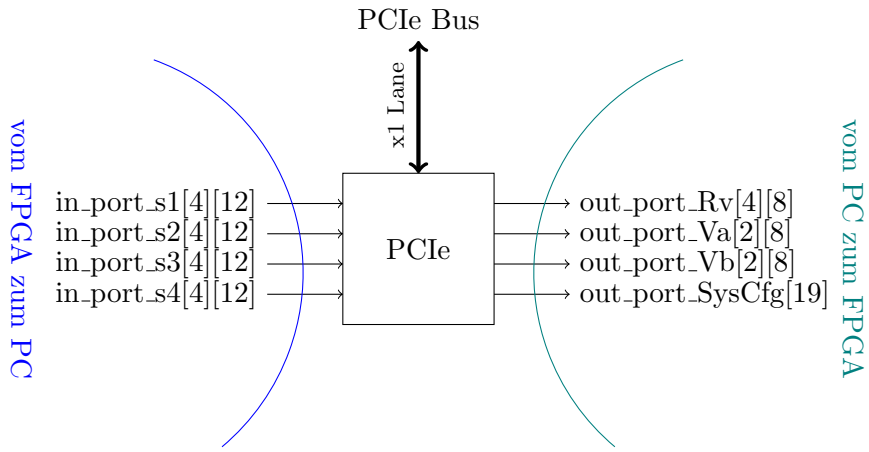



Abb. 3.11: Modul PCIe

vierte Signal wird ausführlich im nächsten Abschnitt behandelt. Doch vorher wird – wie angekündigt – der SOPC Builder besprochen.

In Abbildung 3.12 ist dieser gezeigt und er untergliedert sich in drei Bereiche. Mit der Baumstruktur links können alle zur Auswahl stehenden Komponenten betrachtet und zum Projekt hinzugefügt werden. Das untere Textfeld informiert über den Projektzustand und den größten Platz vereinnahmt die Darstellung des eigentlichen Projekts. Dabei lassen sich einzelne Komponenten an- und abschalten, neu verknüpfen und sich deren Speicheradresse anpassen.

Die PCIe Einheit besteht prinzipiell aus einem PCIe-IP-Kern (IP Compiler for PCI Express), einem DMA Controller und aus vielen parallelen Ein- und Ausgängen (PIO (Parallel I/O)), die für den Datenaustausch nötig sind. Die genauen Einstellungen können über die nebenstehende Büroklammer bezogen werden. Diese ist

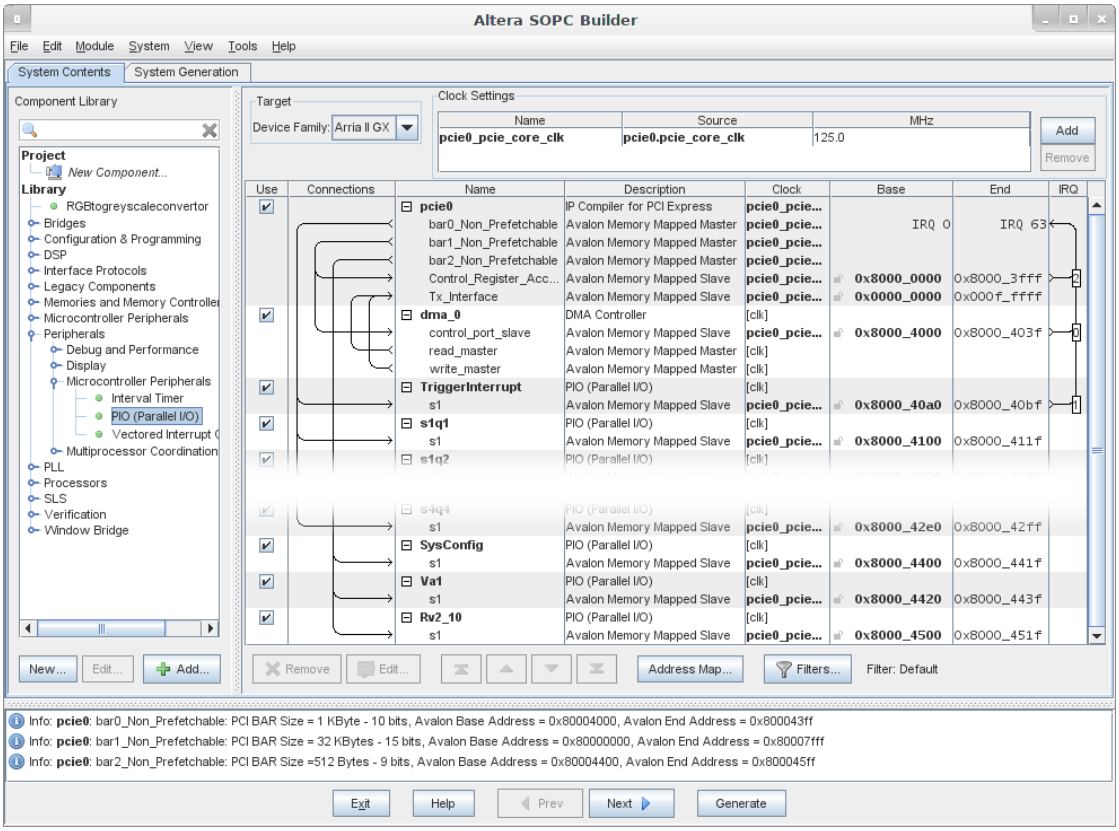



Abb. 3.12: SOPC Builder

ein Verweis auf ein Archiv, in dem sich Screenshots von der festgelegten Konfiguration finden.

Der PCIe-IP-Kern wurde so konfiguriert, dass er über drei sogenannte BARs (***B**ase **A**ddress **R**egisters*) verfügt. Dabei handelt es sich um Konfigurationsregister, die das Betriebssystem über die Speicherbereiche des PCIe-Geräts unterrichten. Drei BARs bedeuten folgerichtig drei Speicherbereiche. In den ersten werden die Messdaten (s1q1, s1q2, ... s4q4) geschrieben, in den dritten schreibt der Linux-Treiber die Systemkonfiguration und in den zweiten ist der DMA Controller abgebildet.

Der Informationszeile in Abbildung 3.12 kann die Größe der einzelnen Speicherbereiche entnommen werden. 1 kB für den ersten, 512 Bytes für den dritten und 32 kB für den zweiten. Weiterhin ist zu sehen, dass die Anfangs- und Endadressen des zweiten Bereichs, die anderen beiden mit einschließen. Die Ursache dafür findet sich in Abschnitt 4.4.

In diesem Abschnitt sollen nur noch, mit Blick auf Abbildung **??**, die Einstellungsmöglichkeiten der PIO erläutert werden. In den Basiseinstellungen kann die Bitlänge des Signals festgelegt werden, die zwischen eins und 32 liegen muss. Weiterhin muss die Datenrichtung gewählt werden, dabei sind Daten, die vom FPGA in die PCIe Einheit laufen, als Eingangssignale (*input*) und Daten mit entgegengesetzter Richtung als Ausgangssignale (*output*) zu sehen.

Eine sehr wichtige Anmerkung sei jedoch noch erlaubt. Einem VHDL-Signal kann zwar der Inhalt einer PIO-Komponente zugewiesen werden, allerdings wirkt sich eine Änderung dieser **nicht** direkt auf das Signal aus! Um Veränderungen mitzubekommen, müssen die PIO-Komponenten periodisch abgefragt werden.

### SYS

Das letzte Modul entspricht einer zentralen Steuereinheit für die bisher besprochenen Module, soweit diese steuerbar sind. Hauptsächlich fallen darunter das

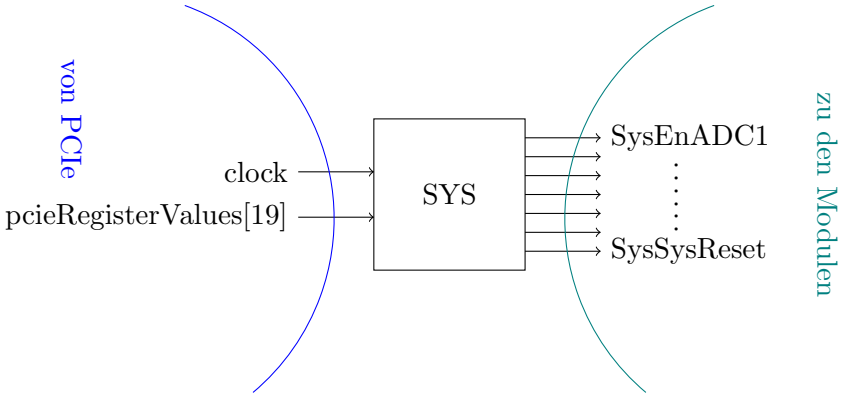



Abb. 3.13: Modul SYS

ADC-, das TLV- und das MCP-Modul. Bei diesem Modul handelt es sich um keine endgültige Fassung, auch wenn die in Tabelle 3.2 ausgeschriebenen Signalnamen dies nahelegen, sondern lediglich um einen zwischenzeitlichen Stand.

Wie die einzelnen Werte geändert werden, wird im Kapitel 5 noch ausgiebig dargelegt, weshalb hier darauf verzichtet wird. Prinzipiell macht dieses Modul jedoch nichts weiteres als mit jedem Takt 19 Bits von einer festgelegten Adresse in der PCIe-Einheit zu lesen und auf vorhandene Änderungen zu reagieren. Dabei gilt zu beachten, dass die Werte vom Modul nicht verändert werden können. Ein Systemreset liegt daher so lange an wie `SysSysReset` auf `1` steht und das Propagieren eines neuen TLV-Wertes erfolgt erst, wenn `SysSetTLV` zwischen zeitig auf `0` gesetzt wurde.

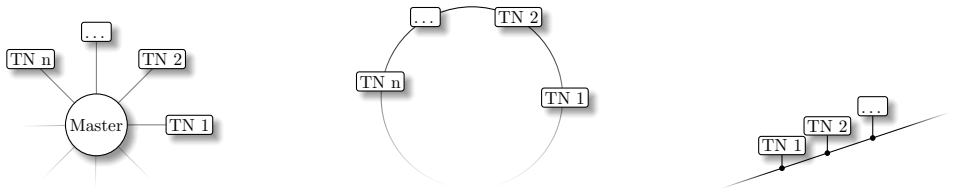
| **Bezeichner** | **Standardwert** |
|---|---|
| SysSysReset | 0 |
| SysEnADC[1..4] | 1 |
| SysAvgADC[1..4] | 0 |
| SysResetAVG | 0 |
| Reserved | 0 |
| SysSampleFreq10k | 1 |
| SysSampleFreq40k | 0 |
| SysSampleFreq100k | 0 |
| SysExtTrig | 0 |
| SysSetMCP[1..2] | 0 |
| SysSetTLV | 0 |
| SysResetTLV | 0 |

Tab. 3.2: Signale des SYS-Moduls

# KAPITEL 4

# PCIe

Bei der Verknüpfung von mehr als zwei Komponenten bedarf es, wenn jeder Teilnehmer mit jedem reden können soll, eines Busses. Um dabei den Verdrahtungsaufwand gering zu halten wurde verschiedene Topologien ausgearbeitet von denen drei exemplarisch in Abbildung 4.1 zu sehen sind. Unterschiede finden sich vor allem in




(a) Sternbus (b) Ringbus (c) Linienbus

Abb. 4.1: Bus Topologien

der Art der Verdrahtung, der Hierarchie und besonders im Gesprächsprotokoll. Je nach Anforderungen legt der eine Bus mehr Wert auf den Datendurchsatz während ein anderer den Fokus auf eine störungsfreie Übertragung setzt. Moderne PCs unterstützen eine Vielzahl von Bussystemen, die i.d.R. ihr Augenmerk auf eine hohe Datenrate richten. Als Beispiele wären der **u**niverselle **s**erielle **B**us (USB), Firewire oder auch der ***P**eripheral **C**omponent **I**nterconnect*-Bus zu nennen.

Während die ersten beiden dazu dienen externe Hardware mit dem PC zu verbinden, ist der letztgenannte Bus in erster Linie dafür entworfen worden die Peripherie (Sound-, Grafik-, Netzwerkkarte, etc.) innerhalb eines Computers miteinander zu verbinden. Auch wenn seit einigen Jahren die genannten Komponenten teilweise in

die CPU integriert werden und man von einem ***S**ystem **o**n a **C**hip* (SoC) spricht, bedarf es nach wie vor eines Busses der den Datenaustausch zwischen den integrierten Chipsätzen ermöglicht.

Bei der Übertragung von Daten gibt es prinzipiell nur zwei Arten, die in Abbildung 4.2 gezeigt sind. Eine parallele und eine serielle, die sich je nach Rah-


(a) seriell (b) parallel

Abb. 4.2: Übertragungsarten

menbedingungen miteinander kombinieren lassen. Sowohl in 4.2a als auch in 4.2b werden die gleichen Informationen übertragen. Im parallelen Ansatz werden sie auf zwei Leitungen bitweise verteilt (ungerade Bits über `data[0]` und gerade über `data[1]`) während im seriellen alle Bits über eine Leitung gehen. Beide Beispiele haben den selben Datendurchsatz, da die serielle Variante mit dem doppelten Takt der parallelen arbeitet und damit die fehlende Leitung kompensiert. In der Praxis würde man Vielfache von Bytes nicht jedoch Bits über verschiedene Leitungen verteilen.

| | **Jahr** | **Änderungen** |
|---|---|---|
| PCI 1.0 | 1992 | Ursprünglicher Entwurf, 33 MHz |
| PCI 2.0 | 1993 | *Plug&Play* |
| PCI 2.1 | 1994 | *Power Management*, 66 MHz |
| PCI 2.2 | 1998 | *Hot Plug* Fähigkeit |
| PCI-X 1.0 | 1999 | miniPCI, 133 MHz, 64 Bit |

Tab. 4.1: PCI Spezifikationen [16]

Der PCI-Bus wird als paralleler Bus je nach Spezifikation mit 33,3 MHz bzw. 66,6 MHz betrieben. Außerdem gibt es 32 und 64 Bit Ausführungen. Erstere verfügt über insgesamt 2 x 62 Leitungen, wobei lediglich 32 davon zur Datenübertragung genutzt werden und alle restlichen als Versorgungs-, Steuerleitungen oder anderen Zwecken dienen. Bei der zweitgenannten verhält es sich ähnlich, hier übertragen 64 von 2 x 92 Leitungen Daten. Eine Auflistung der Veränderungen, die mit den jeweiligen Spezifikationen einherging, ist in Tabelle 4.1 gezeigt. In ihr ist auch

die Änderung der Bezeichnung von PCI auf PCI-e***X***tended vermerkt, mit der fortan all diejenigen Spezifikationen bezeichnet wurden, die eine Steigerung der Übertragungsrate durch eine Takterhöhung erreichen. Einen gänzlich anderen Weg beschreitet der PCI Express (PCIe) Standard, dessen erste Version im Jahre 2002 veröffentlicht wurde. Doch bevor auf die Unterschiede eingegangen wird, werden im folgenden Abschnitt die Gemeinsamkeiten erörtert.

## 4.1 Configuration Space

In einem typischen System werden beim Hochfahren alle an den PCI-Bus angeschlossenen Komponenten abgefragt, um zu ermitteln wie viel Speicher für sie reserviert werden muss, ob sie einer Interruptleitung bedürfen und wie viele Funktionen die Hardware bereitstellt. Diese und weitere Informationen sind im Konfigurationsbereich (*configuration space*) hinterlegt, dessen Aufbau in Tabelle 4.2 gezeigt ist und im folgenden besprochen wird.

Den Anfang macht die `Vendor ID` hinter der sich z.B. ein Chiphersteller verbirgt. In Kombination mit der `Device ID` verfügt das Betriebssystem über genügend Informationen, um nach einem passenden Gerätetreiber suchen zu können. Die Felder `Subsystem Vendor ID` und `Subsystem Device ID` treten in Erscheinung, wenn ein Hersteller einen Chip nicht selbst entwickelt sondern lizenziert hat. Besonders im Grafikkartenbereich ist so etwas häufig anzutreffen.

<table>
<tr><td colspan="4">127 … 96</td><td colspan="2">95 … 64</td><td colspan="2">63 … 32</td><td colspan="2">31 … 16</td><td>15 … 0</td><td></td></tr>
<tr><td>BIST</td><td>Header Type</td><td>Latency Timer</td><td>Cache Line Size</td><td colspan="2">Class Code</td><td>Rev ID</td><td>Status Register</td><td>Command Register</td><td colspan="2">Device ID</td><td>Vendor ID</td><td>0x00</td></tr>
<tr><td colspan="4">Base Address 3</td><td colspan="3">Base Address 2</td><td colspan="2">Base Address 1</td><td colspan="2">Base Address 0</td><td>0x10</td></tr>
<tr><td colspan="2">Subsystem Device ID</td><td colspan="2">Subsystem Vendor ID</td><td colspan="3">Reserved</td><td colspan="3">Base Address 5</td><td colspan="2">Base Address 4</td><td>0x20</td></tr>
<tr><td>0x00</td><td>0x00</td><td>IRQ Pin</td><td>IRQ Line</td><td colspan="5">Reserved</td><td>Cap. Ptr.</td><td colspan="2">Expansion ROM Base Address</td><td>0x30</td></tr>
</table>

Tab. 4.2: *PCI type endpoint - configuration space header*[7, S. 6-2]

Das `Command` und das `Status Register` entsprechen einer Bitmaske über die Funktionen de- und aktiviert bzw. einzelne Status ausgelesen werden können. Informationen über die einzelnen Bits in den Registern finden sich unter [11]. In den nächsten 8 Bits steht die `Revision ID` und kennzeichnet unterschiedliche Hardwareversionen einer Baureihe. Die Geräteklasse wird im `Class Code` gesetzt. Mögliche Klassen sind in der Kerneldatei *pci_ids.h* definiert.

Mit den nächsten vier Bytes wird großer Einfluss auf den Gerätetyp (*root complex*, *switch*, *bridge* oder *endpoint*) genommen. Die einzelne Funktion der Gerätetypen ist leichter anhand der Abbildung 4.3 verständlich. Im *root complex* laufen alle Fäden zusammen und er stellt im PC-Umfeld die Schnittstelle in den Prozessor dar. An ihn können PCI-Geräte (*endpoints*) direkt oder über Zwischenstufen angeschlossen werden. Dabei kann der *switch* als eine Steckplatzerweiterung angesehen werden und die *bridge* erlaubt die Anbindung anderer Bustypen sowie die Einführung einer separaten Busebene. Besprochen wird hier nur der Gerätetyp Endpunkt.

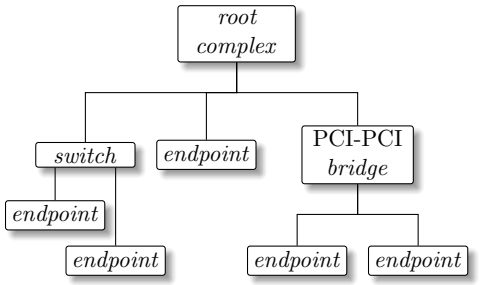


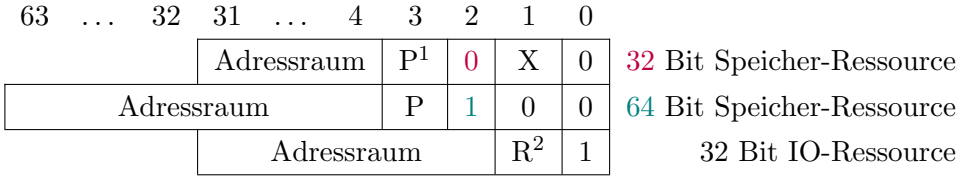

Abb. 4.3: *Header Types*

Bei diesen werden die Register `Latency Timer` und `BIST` (***B**uilt **i**n **s**elf* **t**est) fest auf `0x00` gesetzt, da sie nicht von Bedarf sind. Wenn sie es sind, enthält `BIST` Kontroll- und Statusbits für einen Selbsttest und in `Latency Timer` sind die Latenzzeiten in Vielfachen des Bustakts hinterlegt. Im Register `Cache Line Size` ist die Größe des Zwischenpuffers in Wörtern (ein Wort hat zwei Bytes) gespeichert, was nicht weiter von Belang ist, denn PCIe-Geräte haben keine Verwendung für dieses Feld.

Als nächstes können bis zu sechs Adressbereiche angegeben werden, für die das Betriebssystem Platz im Hauptspeicher reservieren muss. Dabei wird sowohl eine 32 als auch eine 64 Bit Adressierung unterstützt. Für letztere müssen jedoch zwei `Base Address` Register zusammengefasst werden, da eines nur 32 Bit breit ist. Die verschiedenen Konfigurationsmöglichkeiten sind in Tabelle 4.3 zusammengestellt und bedürfen zur vollständigen Erklärung noch einiger Anmerkungen.

Für Verwunderung dürfte in erster Linie sorgen, dass weder für eine 32 noch für eine 64 Bit Adressierung der komplette Adressraum zur Verfügung steht. Diese Einschränkung führt jedoch lediglich dazu, dass Speicher-Ressourcen immer min-

[1] *prefetchable*
[2] reserviert




Tab. 4.3: `Base Address` Typen [11]

destens 16 Bytes und Ein-/Ausgabe-Ressourcen 4 Bytes groß sein müssen. Angegeben wird die Größe durch eine Bitmaske von Nullen. Für einen 512 Bytes großen Speicherblock werden daher die Bits 4 bis 8 auf null und alle höherwertigen auf eins gesetzt. Da mindestens das 31. Bit gesetzt sein muss, kann bei 32 Bit Adressierung maximal eine 2 GB große Ressource angeboten werden. Bleibt noch zu klären was mit R und P gemeint ist. Das zweite Bit in einer IO-Ressource ist reserviert und das vierte in Speicher-Ressourcen legt fest, ob diese *prefetchable* (1) oder *non-prefetchable* (0) sind. Daten sind ersteres, wenn sie sich durch einen Lesevorgang nicht verändern und letzteres, wenn sie es tun[5]. Als Beispiel für *non-prefetchable* Daten wäre ein Ringpuffer zu nennen, da der Lesezeiger nach jedem Lesen seine Position verändert.

Als nächstes bedarf `Expansion ROM Base Address` einer Erklärung. In diesem Register wird wie in den gerade erklärten Registern die Größe eine Speicherblocks angegeben. Der Unterschied zu den vorherigen ist jedoch, dass beim Booten der Inhalt dieses Blocks in den Arbeitsspeicher geladen und ausgeführt wird [18]. Über diesen Weg können Netzwerkkarten Features wie „*Boot from LAN*“ anbieten. Der `Capabilities Pointer` zeigt auf eine verkettete Liste in der die Fähigkeiten der PCI-Einheit notiert sind. Dazu gehört in den meisten Fällen das *Power Management* und bei PCIe-Geräten ***A****dvanced* ***E****rror* ***R****eporting* (AER) und ***M****essage* ***S****ignaled* ***I****nterrupts* (MSI).

Dem reservierten und nicht näher spezifizierten Bereich schließen sich noch zwei Register zum Thema Interrupt an. In `Interrupt Line` notiert das BIOS über welche Leitung die Interruptanfragen (***i****nterrupt* ***r****e****q****uest*) vom PCI-Controller zur CPU laufen und `Interrupt Pin` informiert über den genutzten Interruptpin des PCI-Controllers. Die möglichen Werte sind INTA#,INTB#,INTC# und INTD# wobei die Pins von Steckplatz zu Steckplatz um eine Positionen weiter rutschen, da PCI-Karten mit nur einer Funktion generell INTA# zugewiesen wird. Auf diese Weise lassen sich die Interrupts fair verteilen. PCI-Karten mit mehreren Funktionen ist es erlaubt auch die anderen Pins zu nutzen. Prinzipiell verläuft jedoch jede Interruptanfrage über die `Interrupt Line`, die sich letztlich alle PCI-Endpunkte teilen. Die richtige Interruptzuordnung ist daher im Gerätetreiber zu gewährleisten.

Bei PCI-Geräten ist der Konfigurationsbereich 256 Bytes groß, für PCIe-Geräte erstreckt er sich auf 4 kB und eine Besprechung der hinzugekommenen Einstellungsmöglichkeiten würde viel zu weit führen. Interessierte seien daher auf [7] oder [9] verwiesen.

Unter Linux lässt sich der Konfigurationsbereich eines PCI-Geräts ziemlich einfach abrufen. Dazu genügt ein Aufruf von `lspci`[3] mit dem Parameter `-x` in der Konsole und man erhält eine Ausgabe ähnlich der im Listing 4.1 gezeigten.

```
01:00.0 VGA compatible controller: AMD nee ATI Cedar PRO
00: 22 10 f9 68 07 04 10 00 00 00 03 10 00 80 00
10: 0c 00 00 d0 00 00 00 00 04 00 62 fe 00 00 00 00
20: 01 e0 00 00 00 00 00 00 00 00 00 4b 17 27 e1
30: 00 00 60 fe 50 00 00 00 00 00 00 00 0b 01 00 00

01:00.1 Audio device: AMD nee ATI Cedar HDMI Audio
00: 22 10 68 aa 07 04 10 00 00 00 03 04 10 00 80 00
10: 04 00 64 fe 00 00 00 00 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 4b 17 68 aa
30: 00 00 00 00 50 00 00 00 00 00 00 00 0a 02 00 00

02:05.0 Ethernet controller: 3Com Corporation 3c905B 100BaseTX (rev 30)
00: b7 10 55 90 07 00 10 02 30 00 00 02 10 20 00 00
10: 01 d0 00 00 00 00 52 fe 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 b7 10 55 90
30: 00 00 50 fe dc 00 00 00 00 00 00 00 0a 01 0a 0a
```

Listing 4.1: lspci -x

Die Programmausgabe zeigt zwei Geräte. Eine Netzwerkkarte und eine Grafikkarte, die über eine Soundchip verfügt. Dabei gilt es zu beachten, dass die Darstellung verglichen mit Tabelle 4.2 gespiegelt ist und die Byte-Reihenfolge, in der die Daten gespeichert werden, bekannt sein muss. Die gebräuchlichsten zwei Möglichkeiten Informationen abzuspeichern sind in Tabelle 4.4 gezeigt. Dort wird in beiden Fällen die Dezimalzahl 305.419.896 ($= 12345678_{16}$) abgespeichert.

PCI wurde von Intel definiert und da deren Hausformat *little endian* ist, ist folglich die Hersteller ID von AMD 0x1022 und die von 3Com 0x10b7. Bisher wurde zwar erklärt wie ein PCI-Endpunkt beschaffen sein muss, wie er jedoch adressiert wird, ist noch nicht behandelt worden.

Jedes PCI-Gerät kann anhand seiner Bus-, Geräte- und Funktionsnummer eindeutig identifiziert werden, die ebenfalls im Programmausdruck 4.1 angegeben sind. So hängt die Grafikkarte an Bus 01 und der Netzwerkcontroller an 02. Bei der Busnummer handelt es sich um eine 8 Bit Zahl, so dass von einem System bis

[3] Normalerweise im `pciutils`-Paket enthalten.

| Adresse | Wert |
|---|---|
| 0x00 | 0x12 |
| 0x02 | 0x34 |
| 0x04 | 0x56 |
| 0x06 | 0x78 |
| 0x08 | ... |

big endian

| Adresse | Wert |
|---|---|
| 0x00 | 0x78 |
| 0x02 | 0x56 |
| 0x04 | 0x34 |
| 0x06 | 0x12 |
| 0x08 | ... |

little endian

Tab. 4.4: Byte-Reihenfolge (*endianness*)

zu 256 Busse unterstützt werden. Die Gerätezahl ist 5 Bit breit weshalb pro Bus maximal 32 Geräte bedient werden können, von denen jedes bis zu $2^3$ Funktionen haben kann.[10, S. 303]

Alle bis hierher genannten Merkmale treffen – soweit nicht anders gekennzeichnet – sowohl auf PCI als auf die Weiterentwicklung PCIe zu. Zwischen beiden bestehen allerdings gravierende Unterschiede, die im nächsten Abschnitt besprochen werden.

## 4.2 PCIe ist kein Bus!

Der allergrößte Unterschied zwischen PCI und PCIe ist in der komplett anderen Architektur zu finden. Während der PCI-Bus sternförmig aufgebaut ist und die Bus-Leitungen von mehreren Geräten gemeinsam genutzt werden, findet sich bei PCIe ein gänzlich anderes Konzept. Es gleicht viel mehr einer Punkt-zu-Punkt Kommunikation, wie man sie aus Netzwerken kennt und in Abbildung 4.4 gezeigt ist. Die zwei gezeigten Endpunkte können somit nicht direkt miteinander kommunizieren, sondern jegliche Kommunikation muss über einen gemeinsamen Knoten laufen.

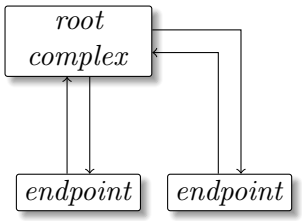



Abb. 4.4: Punkt-zu-Punkt Architektur

Doch nicht nur der Aufbau hat sich geändert. Die Kommunikation selbst findet nun nicht mehr über viele parallele Leitungen, sondern seriell über sogenannte *la-*

*nes* statt. Eine *lane* besteht aus zwei differentiellen Leitungen pro Richtung und einem weiteren differentiellen Paar für den Referenztakt [15]. Für die voll-duplex-fähige (gleichzeitiges Lesen und Schreiben) Schnittstelle ergeben sich somit insgesamt sechs Leitungen. Für differentielle Leitungspaare entscheidet man sich seit langem. Der Vorteil liegt in der Störfestigkeit, die in Abbildung 4.5 veranschaulicht ist.

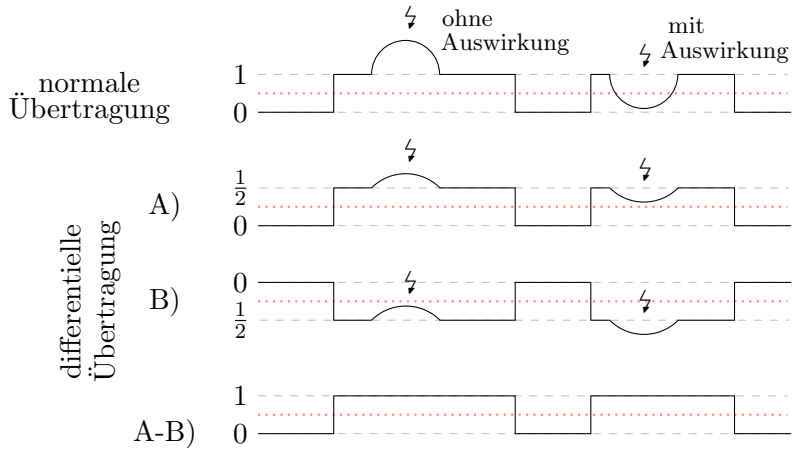



Abb. 4.5: Differentielle Übertragung

Mit der Umstellung auf einen seriellen Datenaustausch muss sich im gleichen Zuge das Protokoll ändern. Denn die Leitungen über die bei PCI die Metadaten (Flusskontrolle, Prüfsummen, etc.) übertragen wurden, sind schlichtweg nicht mehr vorhanden. Angelehnt an Netzwerktechnologien versenden PCIe-Geräte Pakete in die die Rohdaten verpackt sind. Die Pakete bestehen aus drei Schichten und werden im nächsten Abschnitt erläutert.

## 4.3 Paketaufbau

Ein Paket besteht aus drei aufeinander aufbauenden Schichten. Die oberste ist die Transaktionsschicht (*Transaction Layer*), gefolgt von der Sicherungsschicht (*Data Link Layer*) und die unterste bildet die Bitübertragungsschicht (*Physical Layer*). Die oberste Schicht wird in der Literatur mit TLP (***T****ransaction* ***L****ayer* ***P****acket*) abgekürzt und wird im Folgenden genauer betrachtet.

Ein Schreibzugriff macht den Anfang. Das Doppelwort `0x12345678` soll an die Adresse `0xfa9fb060` geschrieben werden. Wie bei einem Paket, das mit der Post aufgegeben wird, muss auch bei Datenpaketen der Absender und der Empfänger

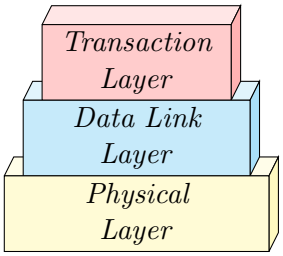



Abb. 4.6: Paketaufbau

vermerkt sein. Diese Informationen finden sich im *Header*, der aus zwei Doppelwörtern besteht. Die Bedeutung der einzelnen Bits können in Tabelle 4.5 erahnt werden, eine genauere Beschreibung folgt jedoch so gleich.

Bei mit R markierten Feldern handelt es sich um reservierte, die mit Nullen

| | 31 | 30 29 | 28 27 26 25 24 | 23 | 22 21 20 19 18 17 16 | 15 | 14 | 13 12 | 11 10 9 | 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| DW0 | R<br>0 | Fmt<br>0x2 | Type<br>0x00 | R<br>0 | TC<br>0x0 | R<br>0x0 | TD<br>0 | EP<br>0 | Attr<br>0 | R<br>0 | Length [9:0]<br>0x001 |
| DW1 | Requester ID [31:16]<br>0x0000 | | | | | Tag (unused)<br>0x00 | | | Last BE<br>0x0 | 1st BE<br>0xf |
| DW2 | Adress [31:2]<br>0x3ea7ec18 | | | | | | | | | R<br>0 |
| DW3 | Data DW 0<br>0x12345678 | | | | | | | | | |

Tab. 4.5: TLP – Schreiben

gefüllt werden müssen und eventuell in späteren Versionen der Spezifikation eine Änderung erfahren. Die Felder `Fmt` und `Type` Feld weisen das Paket als eine Schreibanfrage aus und mit `TC` (***T**raffic **C**lass*) wird die Datenkategorie gewählt. Auf diese Weise ließen sich virtuelle Kanäle schaffen, in der Praxis jedoch findet dieses Feld kaum Anwendung und ist mit null belegt. `TD` signalisiert, dass es keine zusätzlichen CRC-Prüfdaten gibt, was i.d.R. der Fall ist, denn der umschließende *Data Link Layer* wird sowieso einen ***C**yclic **R**edundancy **C**heck* anfertigen und anhängen. Es soll ein 32-Bit Wert geschrieben werden, folglich ist die Länge auf eins zu setzen. Mit der `Requester ID` wird kenntlich gemacht von wem die Anfrage stammt (hier *root complex*) und wohin die Bestätigung bzw. die Nicht-Bestätigung geschickt werden soll. Das anschließende `Tag` Feld ist im Schreibmodus ungenutzt, wird jedoch bei Leseanfragen Verwendung finden.

Sollen nur einzelne Bytes geschrieben werden, selektiert man diese über die `BE`

Felder (***B**yte **E**nable*). In unserem Falle sollen alle vier Bytes geschrieben werden, weshalb `1st BE` den Wert `0xf` enthält. Soll hingegen nur das erste (`0x78`) und das dritte Byte (`0x34`) geschrieben werden, ist `1st BE` auf `0x5` zu setzen. Beim gewählten Beispiel wird nur ein Doppelwort übertragen, weshalb `Last BE` auf Null zu setzen ist, sollen jedoch bei einem von mehreren Doppelwörtern bestimmte Bytes selektiert werden, muss `Last BE` entsprechend gesetzt werden.

An welche Position die Daten geschrieben werden sollen, wird über `Adress` bestimmt. Dabei erhält man die zu übermittelnde Adresse von der gewünschten, indem man diese um zwei Bit nach rechts verschiebt. Aus `0xfa9fb060` wird somit `0x3ea7ec18`. Schlussendlich folgen die eigentlichen Daten.

Ein TLP ist je nach Mächtigkeit des nächsten Kommunikationspartners maximal 128, 256 oder 512 Bytes groß. Wenn der Anteil der Nutzdaten wesentlich größer ist als der des Paketkopfes, spricht man von einem *burst*-Zugriff.

Eine Leseanfrage gestaltet sich recht ähnlich, wie Tabelle 4.6 zeigt. Auch hier muss angegeben werden von wo (`Adress`) wie viele (`Length`) Doppelworte gelesen werden soll und wer nach diesen fragt (`Requester ID`). Die Felder `Fmt` und `Type` wandeln sich leicht ab und Aufmerksamkeit muss lediglich dem `Tag` Feld geschenkt werden. Je nach Verfügbarkeit werden Anfragen nicht sofort beantwortet und erstmal in eine Warteschlange eingereiht. Es kann daher passieren, dass ein Gerät von

| | 31 | 30 29 | 28 27 26 25 24 | 23 | 22 21 20 19 18 17 16 | 15 | 14 | 13 12 | 11 10 9 | 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| DW0 | R<br>0 | Fmt<br>0x0 | Type<br>0x00 | R<br>0 | TC<br>0x0 | R<br>0x0 | TD<br>0 | EP<br>0 | Attr<br>0 | R<br>0 | Length [9:0]<br>0x001 |
| DW1 | Requester ID [31:16]<br>0x0000 | | | | | Tag [15:8]<br>0x0c | | | | Last BE<br>0x0 | 1st BE<br>0xf |
| DW2 | Adress [31:2]<br>0x3ea7ec18 | | | | | | | | | | R<br>0 |

Tab. 4.6: TLP – Lesen

einem Absender mehrere Anfragen erhält. Damit dieser die Antworten eindeutig zuordnen kann, bedarf es einer Markierung, die mittels `Tag` umgesetzt wird. Dafür dürfen jedoch nur die unteren 5 Bit (12:8) genutzt werden, da die oberen laut Standard Null sein müssen. Für die Markierung selbst, gibt es keine Vorschriften.

Für die Antwort verschickt der Angefragte ein Paket, das dem Format in Tabelle 4.7 gleicht. `Fmt` und `Type` machen kenntlich, dass es sich um eine Antwort auf eine Leseanfrage handelt und in `Length` steht weiterhin die Größe der im Paket enthaltenen Daten. Es wurde nach einem Doppelwort gefragt, folglich enthält die

| | 31 30 29 | 28 27 26 25 24 | 23 | 22 21 20 19 18 17 16 | 15 | 14 | 13 12 | 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|
| DW0 | R<br>0 | Fmt<br>0x2 | Type<br>0x0a | R<br>0 | TC<br>0x0 | R<br>0x0 | TD<br>0 | EP<br>0 | Attr<br>0 | R<br>0 | Length<br>0x001 |
| DW1 | Completer ID [31:16]<br>0x0100 | | | | Status<br>0x00 | | BCM<br>0 | Byte Count<br>0x4 | | | |
| DW2 | Requester ID [31:16]<br>0x0000 | | | | Tag [15:8]<br>0x0c | | | R<br>0 | Lower Address<br>0x60 | | |
| DW3 | Data DW 0<br>0x12345678 | | | | | | | | | | |

Tab. 4.7: TLP – Komplettierung

Antwort nur eines. Jedoch ist es auch möglich, die angeforderten Daten auf mehrere Pakete aufzuteilen. Um dabei den Überblick zu behalten, wertet man `Byte Count` aus. In diesem Feld stehen die noch zu übertragenden Bytes, wobei die im Paket enthaltenen noch nicht abgezogen sind!

Ein anderer Ansatz ist die Auswertung von `Lower Address`. Dieses Feld enthält die unteren sieben Bit der Adresse von welcher aus die Daten gelesen wurden. Das Bit `BCM` ist nur dann auf Eins gesetzt, wenn zwischen dem Anfragenden und dem Antwortenden eine PCI-X zu PCIe Brücke liegt und wenn `Status` nur Nullen aufweist, markiert das eine erfolgreiche Übertragung. `Tag` ermöglicht wie schon erwähnt die eindeutige Zuordnung der Antwort(en) auf die Anfragen.

Bleibt noch die `Completer ID`. Deren Zusammensetzung entspricht der eindeutigen Busadressierung wie sie auf Seite 32 bereits besprochen wurde. Die Quintessenz steht nochmals in Tabelle 4.8.

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | | |
|---|---|---|
| Requester/Completer ID<br>0x0100 | | |
| Bus Number [15:8]<br>0x01 | Dev. Number<br>0x00 | Function<br>0x0 |

Tab. 4.8: TLP – Requester/Completer ID

Neben den bereits genannten Unterschieden zwischen PCI und PCIe gibt es noch einen weiteren zu nennen. Während PCI *little endian* ist, verschickt PCIe die Daten ***big endian*** [4]. Je nach Zielplattform müssen die Daten daher vorher konvertiert werden.

Ehe auf die Sicherungsschicht eingegangen wird, soll noch der mit PCI-2.2 eingeführte Weg Interrupts abzusetzen besprochen werden. Die bisherige Methode ist bereits im Abschnitt 4.1 erörtert worden und hat den Nachteil, dass ein Gerät maximal vier separate Interrupts akquirieren kann und im Treiber der Ursprung erst noch herausgefunden werden muss. Die neue Interruptmethode heißt ***M**essage **S**ignaled **I**nterrupts* (MSI) und der Name ist Programm. Anstelle Interruptanfragen über spezielle Leitungen zu signalisieren, verschickt man eine gewöhnliche Schreibanfragen an eine bestimmte Adresse innerhalb vom *root complex*. Da im Paketkopf alle Informationen über den Absender (`Requester ID`) stehen, spart man sich die Interruptzuordnung und zusätzlich sind auf diese Weise bis zu 32 unabhängige Interrupts pro Gerät möglich. In der mit PCI-3.0 definierten Weiterentwicklung MSI-X sind sogar bis zu 2048 Interrupts pro Gerät erlaubt.

Die Transaktionsschicht handhabt die Kommunikationsrichtung, die Sicherheitsschicht garantiert die Übermittlung der TLPs. Analog zu diesen werden Pakete der Sicherheitsschicht als *Data Link Layer Packet* (DLLP) bezeichnet. Von diesen gibt es vier Typen.

**Ack DLLP** signalisiert die erfolgreiche Übertragung eines TLPs.

**Nack DLLP** wird gesendet, wenn die Übermittlung eines TLPs nicht erfolgreich war bzw. die Daten nicht rekonstruiert werden konnten oder das Paket nicht innerhalb eines Timeouts zugestellt wurde. In diesen Fällen wird das Paket erneut angefordert.

**Flow Control DLLP** stellt die Funktionalität der Schnittstelle sicher.

**Power Management DLLP** ermöglicht durch geeignete Maßnahmen (z.B. Taktreduktion) den Energieverbrauch zu senken.

Die Flusskontrolle ist am kompliziertesten, weshalb sie genauer betrachtet wird, während auf die anderen nicht weiter eingegangen wird. Die Kontrollmechanismen erfordern sechs eigenständige Puffer, die folgende, in der Spezifikation als *credit type* bezeichnete, Daten halten.

1. *Posted Request TLP's headers*
2. *Posted Request TLP's data*
3. *Non-Posted Request TLP's headers*
4. *Non-Posted Request TLP's data*
5. *Completion TLP's headers*

6. *Completion TLP's data*

Von den aufgezählten Datentypen sind bereits alle behandelt worden, lediglich der Begriff *posted* und *non-posted* ist noch nicht erklärt, was sogleich nachgeholt wird.

**non-posted** heißen Pakettypen, bei denen eine Antwort erwartet wird und daher nicht vergessen werden dürfen, wie es bei einer Leseanfrage der Fall ist.

**posted** bezeichnet Pakete, die abgeschickt werden und die keine Empfangsbestätigung erwarten. Ein Beispiel hierfür ist eine Schreibanfrage.

Um einen Überblick über den Datenverkehr zu behalten, wird in der Flusskontrolleinheit jeder *credit type* in einem eigenen 16 Bit Zähler mit notiert. Besteht ein Paket nur aus einem einfachen *header* (3-4 Bytes), so wird der Zähler um eins erhöht. Handelt es sich jedoch um ein größeres Datenpaket, so wird die Menge der enthaltenen Doppelwörter durch vier dividiert, das Ergebnis aufgerundet und zum Zähler addiert. Anhand des Füllstandes kann die PCIe-Einheit die anderen Gesprächspartner darüber informieren für wie viele Pakete sie noch Platz hat. Auf diese Weise werden einem PCIe-Geräte niemals mehr Pakete gesendet als es annehmen kann. Dadurch spart man sich unnötige Abfragen und steigert den Nettodatendurchsatz.

Im DLL wird jedes TLP mit einer Nummer gekennzeichnet, die zur vorhergehenden um eins erhöht ist. Anhand dieser Sequenznummern können die einzelnen TLPs identifiziert und gruppiert werden. Das ist nötig, denn wenn ein Empfänger, der die Sequenznummern pro Sender mit notiert, ein Paket aufgrund eines Fehlers mit Nack quittiert, muss der Sender ausgehend vom letzten mit Ack quittierten Paket alle Pakete nochmals senden, da der Empfänger bereits empfangene mit einer höheren Sequenznummer markierte Pakete verwirft. Dieses Verhalten ist in Abbildung 4.7 nochmals verdeutlicht und sie zeigt auch, dass die Pakete nicht in der richtigen Reihenfolge ankommen müssen.

Die Zählerstände in der Flusskontrolle werden mit Erhalt der Empfangsbestätigung (Ack) dekrementiert, worauf den Kommunikationspartner mitgeteilt wird wie viele Pakete entgegen genommen werden können. Zeit dafür findet sich immer, schließlich ist PCIe duplex-fähig, d.h. es kann gleichzeitig gesendet und empfangen werden.

Prinzipiell ist die Kommunikation auf den Leitungen nicht vorhersagbar, schließlich nehmen viel zu viele Einzelfaktoren Einfluss. Es ist jedoch möglich das Ende eines Schreibzugriffs durch eine Leseanfrage zu erfahren, denn diese wird erst verschickt, wenn alle zum Schreibzugriff zugehörigen Pakete übermittelt wurden. Da

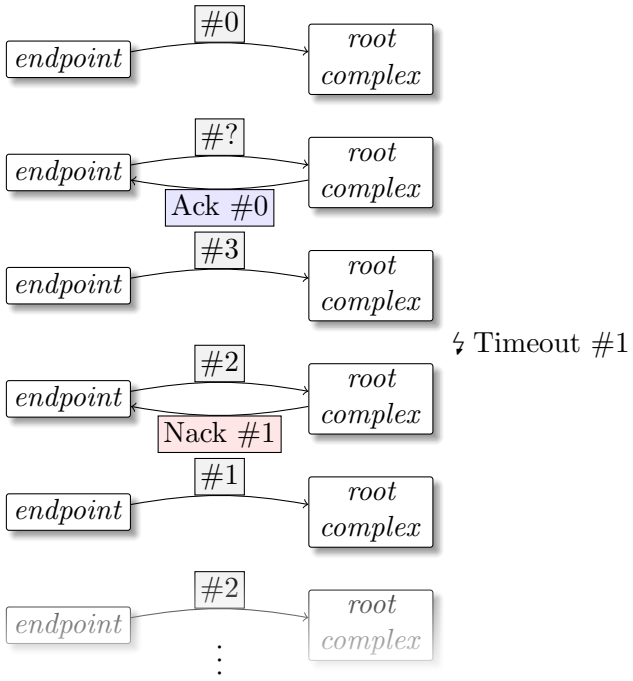



Abb. 4.7: PCIe – Ack/Nack Flusskontrolle

man allerdings i.d.R. nicht an der Antwort interessiert ist, kann man sich eines Tricks bedienen und eine spezielle Leseanfrage aufgeben. Der Kniff besteht darin `Length` auf `0x001` und `1st BE` auf `0x0` zu setzen. Das empfangene Doppelwort hat einen zufälligen Wert und muss daher verworfen werden, was jedoch nicht von Belang ist, da man im Endeffekt nur über das Ende des Schreibvorgangs informiert werden wollte.

Für die sichere Übertragung über ein unsicheres Medium wird an jedes TLP eine 32 Bit CRC-Prüfsumme angehängt, um eventuell auftretende Bitfehler korrigieren zu können. Auf physikalischer Ebene bedient man sich noch weiterer Methoden, das Auftreten von Fehlern zu vermeiden.

Eine davon ist der 8b-10b-Code bei dem 10 Bits genutzt werden, um ein Byte zu übertragen. Das erhöht zwar die Menge der zu übertragenden Bits um ein Viertel, jedoch überwiegen die damit gewonnen Vorteile. Die da unter anderem sind:

**Gleichspannungsausgleich** Zwischen einzelnen Leiterbahnen und ihrer Umgebung sind immer unvermeidbare, parasitäre Kapazitäten anzutreffen. Liegt nun ein *high*-Pegel längere Zeit an, werden diese Kapazitäten unter Umständen so weit aufgeladen, dass der anschließende *low*-Pegel am Empfänger nicht als

solcher detektiert wird. Bei der 8b-10b-Kodierung unterscheidet sich die Anzahl der Einser pro Symbol von der Anzahl der Nullen um maximal zwei. Betrachtet man alle Kombinationsmöglichkeiten treten in einem Datenstrom 252 neutrale und gleich viele positive wie negative Kombinationen auf, was im Mittel eine gleichspannungsfreien Übertragung entspricht.

**Taktrückgewinnung** Es wird maximal fünfmal das gleiche Bit übertragen, danach muss der Pegel wechseln. Diesen Umstand macht man sich für eine Taktrückgewinnung zunutze.

Neben der Kodierung definiert die Spezifikation auch wie die einzelnen Bits physikalisch übertragen werden. Zur Wahl stehen die zwei Schnittstellen-Standards LVDS (***L**ow-**V**oltage **D**ifferential **S**ignaling*) und PCML (***P**seudo **C**urrent **M**ode **L**ogic*). Bei beiden Standards geht es im Grunde darum das EMV-Verhalten zu verbessern und dennoch hohe Taktraten zu erzielen. Dazu bedient man sich der in den Grundlagen der Elektrotechnik hergeleiteten Strom-/Spannungsbeziehungen am Kondensator, wie er in Abbildung 4.8 zu sehen ist. Demnach besteht zwi-

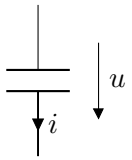



Abb. 4.8: Beziehungen am Kondensator

schen den Größen, der in Gleichung (4.1) ausgedrückte Zusammenhang, dem die wichtigsten Einflussfaktoren entnommen werden können.

$$i(t) = C \cdot \frac{\partial u(t)}{\partial t} \tag{4.1}$$

In Worten gibt die Gleichung an, dass immer wenn sich die Spannung über dem Kondensator ändert, ein Ausgleichsstrom fließt. Wie bereits erwähnt sind parasitäre Kapazitäten zwischen Leiterbahnen unvermeidbar, allerdings sind diese so klein, dass sie erst bei hohen Frequenzen in Erscheinung treten. Für eine Erhöhung der Datenrate ist eine Steigerung der Frequenz jedoch schlichtweg unvermeidlich und zur Begrenzung der unerwünschten Ausgleichsströme verbleibt daher nur die Reduzierung der Spannungsänderung. Dieser Weg wurde in der Vergangenheit mehrmals beschritten, nur ist man mittlerweile an der Grenze des machbaren angelangt. Für eine weitere Steigerung der Datenrate bedurfte es folglich eines neuen Ansatzes.

Während Ströme aus Spannungsquellen vom höheren zum niedrigeren Potential fließen, fließen Ströme aus Stromquellen über den Verbraucher zurück in die Quelle. Es bot sich daher an von Spannungsquellen, deren Ströme auch über parasitäre

Kapazitäten abfließen, auf Stromquellen umzusteigen. Die Bitzustände wurden

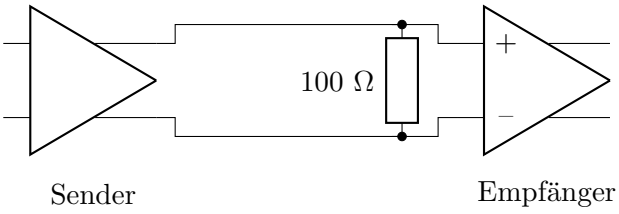



Abb. 4.9: LVDS

nun nicht mehr über Spannungslevel festgelegt, sondern über die Richtung in der der Strom aus dem Sender durch den Empfänger fließt. Die erwähnten Standards (LVDS, PCML) fußen auf genau diesem Konzept und anhand der Abbildung 4.9 soll die Detektion am Beispiel von LVDS erklärt werden. Die gezeigten Operationsverstärker können zum leichteren Verständnis als ideal angenommen werden. In diesem Fall haben sie einen unendlich großen Eingangswiderstand und der gesamte Strom, der aus dem Sender fließt, muss durch den Widerstand fließen. Bei einem Strom von 3,5 mA und einem Widerstand von 100 Ω ergibt sich je nach Stromrichtung nach dem ohmschen Gesetz ein Spannungsabfall von ±350 mV.

Gegenüber Gleichtaktstörungen ist dieser Ansatz sehr tolerant, schließlich heben diese das Spannungsniveau auf beiden Leitungen gleichermaßen, was bei einer Differenzbildung nicht zum Vorschein tritt. Gegentaktstörung können sich hingegen wegen der schon klein gewählten Betriebsströme stärker auswirken. Beim Leitungsdesign ist deshalb darauf zu achten, dass die Leiterpaare eine möglichst kleine Fläche aufspannen durch die ein magnetisches Feld Störungen einkoppeln könnte. Weitere Informationen zu LVDS bzw. (P)CML können [2] und [1] entnommen werden.

In den vorhergehenden Seiten wurden die inneren Zusammenhänge von PCI bzw. PCIe erklärt. Wenn nur ein paar Pakete verschickt werden sollen, sind die gezeigten Möglichkeiten völlig ausreichend. Für einen wesentlich schnelleren Datentransfer gibt es allerdings noch einen Turbo, dem der nächsten Abschnitt gewidmet ist.

## 4.4 DMA

In den bisher gezeigten PCIe-Operationen richteten sich alle Anfragen vom *root complex* in Richtung eines Endpunktes. Für die Gegenrichtung ist es nötig, dass die PCIe-Teilnehmer in ihrem `Command Register` das `Code Master` Bit setzen, was ihnen erlaubt eigenständig Pakete zu verschicken. Auf diese Art können Geräte

Änderungen ihrer Zustände direkt an das System übermitteln, ohne auf eine Abfrage warten zu müssen. Dazu muss der Endpunkt allerdings wissen wohin die neuen Daten geschrieben werden sollen.

Generell lädt ein Betriebssystem beim Ausführen eines Programmes dieses mit all seinen Daten in den Hauptspeicher, auch Arbeitsspeicher genannt, da so ein wesentlich schnellerer Datenzugriff möglich ist. Von dort werden die einzelnen Instruktionen in die CPU geladen, ausgeführt, und das Ergebnis wird im Anschluss in den Arbeitsspeicher zurückgeschrieben bzw. die veränderten Daten werden spätestens bei Beendigung des Programms auf einem Datenträger gespeichert.

In den Speicher lassen sich allerdings auch ohne Probleme ganze Speicherbereiche, die sich auf peripheren Komponenten befinden, abbilden. Unter Linux geschieht diese Zuordnung (*mapping* genannt) automatisch und es kann ohne größeren Aufwand im Speicher der Zustand des Peripherieressourcen betrachtet werden. Allerdings wird die Peripherie nur einmalig ausgelesen! Die Daten im Arbeitsspeicher entsprechen folglich zumindest nach einer gewissen Zeit nicht mehr den Daten in der Peripherie.

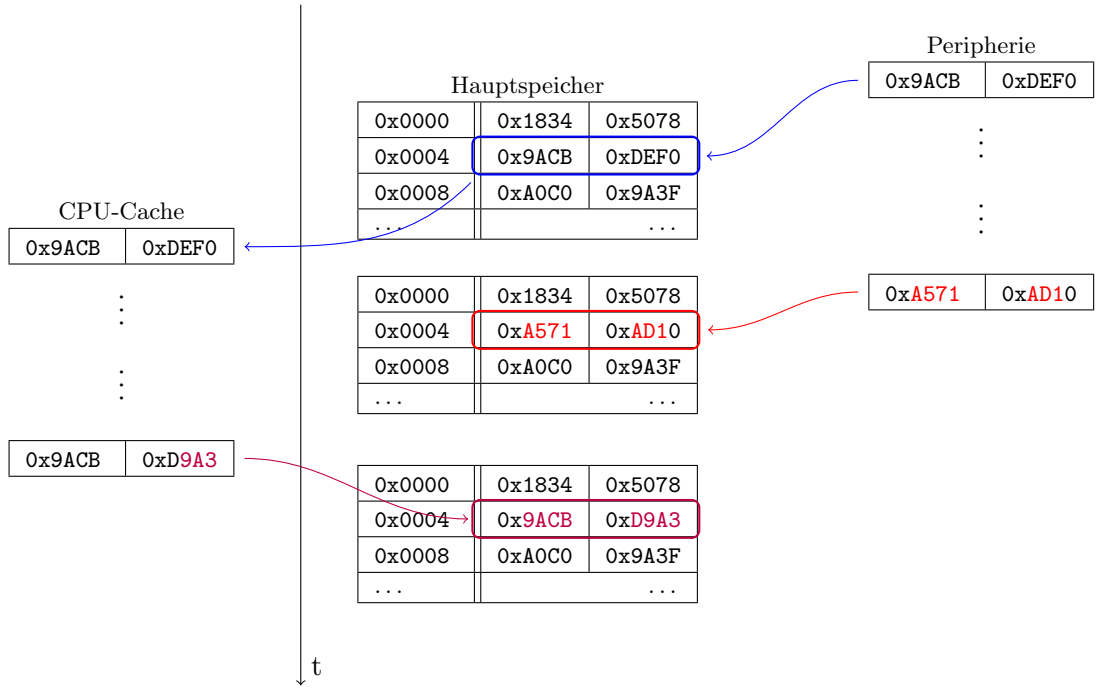



Abb. 4.10: DMA Problematik [8]

Um die Daten aktuell zu halten, kann man diese entweder zyklisch abfragen oder man sagt der Hardware sie solle den Arbeitsspeicherbereich selbst auf dem aktuellsten Stand halten. Für letzteres bedient man sich dem ***D**irect **M**emory **A**ccess* (DMA). Damit treten jedoch Schwierigkeiten in Erscheinung.

Die Peripherie schreibt, wie in Abbildung 4.10 zeigt, zu einem gewissen Zeitpunkt neue Daten in den Arbeitsspeicher, welche etwas später in die CPU geladen und verarbeitet werden. Nur einen kurzen Augenblick später schreibt die Hardware aktuellere Daten in den Hauptspeicher, die auf die laufende Verarbeitung allerdings keinen Einfluss mehr haben. Nachdem die Datenverarbeitung innerhalb der CPU abgeschlossen ist, werden die dadurch veränderten Daten zurück und die aktuellsten überschrieben. Kurz gesagt können auf diese Art Informationen verloren gehen.

Abhilfe schafft hier nur ein gutes Design, was mit der Erklärung des DMA-Controllers klarer werden dürfte. Dazu sind in Tabelle 4.9 die wichtigen Register gezeigt. Die einzelnen Registerfunktionen werden im zugehörigen Handbuch [6]

| Offset | Register Name | Read/ Write | 31...5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x00 | status | RW | Reserved | LEN | WEOP | REOP | BUSY | DONE |
| 0x04 | readaddress | RW | Read master start address | | | | | |
| 0x08 | writeaddress | RW | Write master start address | | | | | |
| 0x0c | length | RW | DMA transaction length (in bytes) | | | | | |
| 0x10 | -- | -- | Reserved | | | | | |
| 0x14 | -- | -- | Reserved | | | | | |
| 0x18 | control | RW | DMA transaction settings | | | | | |
| ... | | | | | | | | |

Tab. 4.9: DMA Controller Register [6, S. 24-7]

genauer erläutert, hier sollen lediglich die wichtigsten Elemente für eine DMA-Operation besprochen werden. In `readaddress` muss die Adresse hinterlegt werden ab der die Daten gelesen werden sollen und in `writeaddress` steht analog die Zieladresse. Mit `length` wird dem Controller mitgeteilt wie viele Bytes transferiert werden sollen und in `control` wird unter anderem festgelegt um was für Daten es sich handelt (Bytes, Wörter, Doppelwörter, ...) und durch setzen des enthaltenen Bits `GO` wird die Transaktion angestoßen.

Die Datenübertragung selbst handhabt der DMA-Controller, die Konfiguration hingegen muss von außen vorgenommen werden. Es ist möglich diesen so zu konfigurieren, dass mit dem Ende der Übertragung ein Interrupt (MSI) abgesetzt wird,

um auf diese Weise das Betriebssystem über den Abschluss zu informieren. Mittels Interrupts könnte folglich die in Abbildung 4.10 angesprochene Problematik umgangen werden.

So schön einfach DMA-Operationen erscheinen mögen, in der Praxis sieht es ein klein wenig anders aus. Fordert man auf PC-Seite vom Betriebssystem einen größeren Speicherbereich an, so erhält man von diesem auf den ersten Blick einen zusammenhängenden Block. In Wirklichkeit handelt es sich allerdings um einen virtuellen Speicherbereich, der aus einzelnen wild im Arbeitsspeicher verstreuten Blöcken besteht, wie Abbildung 4.11 verdeutlicht. Mit dem Wissen, dass man dem

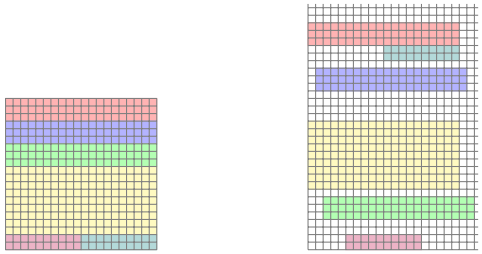



Abb. 4.11: Virtueller und physischer Speicher

DMA-Controller nur den Anfang eines Adressbereiches übergeben kann, werden die sich daraus ergebenden katastrophalen Folgen schnell klar. Denn der Controller würde nicht nachdem der rote Block beschrieben wurde am Anfang des blauen weiterschreiben, sondern einfach nur in die nächste Zeile rutschen. Abhilfe schafft hier ein sogenannter *Scatter-Gather DMA-Controller*, der nur der Vollständigkeit halber genannt wird. Denn die von den Sechstoren generierten Daten benötigen insgesamt 1 kB und Gedanken bzgl. fragmentierte Speicherbereiche muss man sich erst ab 4 MB machen.

KAPITEL 5

# Treiber

In den vorangegangenen Kapiteln wurde – einfach gesprochen – die Hardware und ihr Innenleben behandelt. Der Inhalt dieses und des nächsten Kapitels widmet sich nun der Softwareseite. Den Anfang macht der Treiber, der über die am Rand befindlichen Büroklammern bezogen werden kann. Es handelt sich um einen Treiber für den Linux Kernel, da dieser quelloffen ist und im Internet viele Hilfestellungen zu finden sind. Selbstredend spielte der persönliche Geschmack die größere Rolle bei der Wahl des Betriebssystems.

Es gibt zwar nur einen Linux-Kernel, jedoch gibt es eine schier unendliche Anzahl an Distributionen, die wiederum ihre eigenen Vor- und Nachteile mit sich bringen. Die `MSMST` Plattform wird mit der Distribution `Fedora`[1] ausgeliefert, allerdings bereitete es Schwierigkeiten diese auf die aktuellste Version upzudaten und so wurde stattdessen `Gentoo`[2] installiert.

Im Unterschied zu den meisten anderen Distribution liefert `Gentoo` keine fertig kompilierten Programme, sondern deren Quelltext aus. Folgerichtig müssen die Anwendungen vor ihrer Verwendung erst noch übersetzt werden. Je nach Leistungsstärke des Systems ist dies mehr oder weniger zeitintensiv. Bei der schwachen Atom-CPU dauerte es dementsprechend etwas länger. Aber der große Vorteil besteht unbestritten in der Fähigkeit das System den eigenen Wünschen individuell anpassen zu können. Beispielhaft sei das An- und Abwählen von Features (USB, Firewire, ...) einzelner Programme genannt oder die Erzeugung von Programmcode, der auf die CPU zugeschnitten ist.

[1] http://www.fedoraproject.org/
[2] http://www.gentoo.org/

Das Kapitel gliedert sich in eine kurze Wiedergabe der Eindrücke, die im Open Source Umfeld gewonnen wurden und geht im Anschluss über in die detaillierte Erläuterung des Treibers. Den Anfang macht die Beleuchtung des Kernel-Modul-Headers, da dieser die im Treiber genutzten Variablen und Makros deklariert. Der Treibercode selbst splittet sich in zwei Komponenten auf. Der eine Abschnitt behandelt die genutzten Schnittstellen (***A**pplication **P**rogramming **I**nterface*, API) für den Zugriff auf PCI(e)-Geräte und der andere bespricht das Erstellen einer Gerätedatei über die Anwendungen die Messdaten auslesen können.

Open Source Software ist zwar im Quelltext vorhanden, das muss jedoch nicht heißen, dass dieser gut dokumentiert ist bzw. es eine Dokumentation zu den Programmierschnittstellen gibt. Beim größten Open Source Projekt ist glücklicherweise beides der Fall. Hauptinformationsquelle bildeten die Bücher [17], [10] und selbstverständlich das Internet. In beiden Büchern wird der Kernel in der Version 2.6 behandelt. Aktuell war jedoch zu Beginn der Entwicklung die Version 3.5 und es dauerte einige Zeit bis ich zufälligerweise entdeckt hatte, dass sich mit dem Versionssprung die API nicht geändert hat. Auch nicht einfacher machen es die vielen Beispiele im Internet, die für eine ältere API geschrieben sind und aus Kompatibilitätsgründen weiterhin funktionieren. Neben den genannten Büchern sind noch die Internetseiten `http://kernelnewbies.org/` und `https://lwn.net/` sowie die in den Kernel-Quellen beiliegende Dokumentation zu nennen.

Das Kompilieren des Treibers wurde durch ein Makefile vereinfacht. Liegt es zusammen mit den Quelldateien des Treibers in einem Verzeichnis, reicht es die im Listing 5.1 aufgeführten Befehle als Superuser auszuführen. Die Quellen des Linux-Kernels müssen selbstredend installiert sein.

```
$ make
$ insmod msmst.ko
```

Listing 5.1: Kompilieren und Verwenden des Kernel-Moduls

## 5.1 Header

Der Quelltext für C Programme teilt sich gewöhnlich in eine .c-Datei, die den eigentlichen Programmcode enthält, und in eine Header-Datei (.h) auf. In letzterer werden häufig Makros, eigene Datentypen und Datenstrukturen definiert und wenn die geschriebenen Funktionen noch in anderen Programmen genutzt werden sollen, werden sie hier bekanntgemacht. Aus den im folgenden besprochenen Quellcodeauszügen wurden die Kommentare entfernt. Aus diesem Grund sind die angegebenen Zeilennummer nur als eine Art Hilfestellung zu sehen.

In diesem Abschnitt wird der Inhalt der Header-Datei Schritt für Schritt erläutert. Dem schließt sich eine Betrachtung der in *msmst.c* inkludierten Funktionsbibliotheken als auch die Erklärung von Kernel-Modul-Merkmalen an. Danach ist sowohl der *char device* Programmierung als auch der Erläuterung der genutzten PCI-API jeweils ein eigener Abschnitt gewidmet.

Wie für Header-Dateien üblich wird zu Beginn überprüft, ob diese nicht schon eingebunden wurden. Das kontrolliert die Präprozessoranweisung in Zeile 1 und verhindert, wenn die Bedingung unwahr ist, eine Verarbeitung des nachfolgenden Codes.

```
1 #ifndef __MSMST_H_
2 #define __MSMST_H_
```

Ist sie hingegen wahr, wird das Makro auf das vorher geprüft wurde bekanntgemacht. Danach folgen weitere Makros, die für die Treiberzuordnung die gleichen Werte enthalten, wie sie im Konfigurationsbereich (siehe 4.1) hinterlegt sind.

```
3 #define MY_VENDOR_ID    0x187C
4 #define MY_DEVICE_ID    0x0004
5 #define MY_SUBSYS_ID    0x0004
6 #define MY_SUBVEN_ID    0x187C
7 #define MY_CLASSCODE    0x110000
8 #define MY_REV_ID       0x04
```

Mit den später genutzten PCI-Funktionen lässt sich bestimmen an welcher Adresse die Daten liegen, jedoch erhält man nur den Anfang des Speicherblocks zurück. Um einen einfachen Zugriff auf die einzelnen Sechstorwerte zu erhalten, addiert man zu dem Speicherblockanfang einen zugehörigen Offset.

```
9 #define OFF_S1Q1   0x100
10 #define OFF_S1Q2   0x120
11  ..
12 #define OFF_S4Q4   0x2e0
```

Selbiges gilt für den beschreibbaren Datenbereich.

```
13 #define OFF_SYSCONFIG 0x0
14 #define OFF_VA1       0x20
15  ..
16 #define OFF_RV2_10    0x100
```

Mit Blick auf Abbildung 3.12 sei in Erinnerung gerufen, dass insgesamt drei BARs definiert wurden. Folglich müssen ebenso drei Strukturinformationen (Anfang, Ende und Länge der Speicherbereiche sowie deren Zustand) gespeichert werden können.

```
17 #define BAR_NUM          (3)
```

Jedes Kernel-Modul muss ein paar grundlegende Informationen bereitstellen. Zwingend erforderlich ist dabei nur der Name und die Lizenz. Der Autor, die Beschreibung sowie die Version sind optional. Die verwendete Lizenz (GPL) wird im Treiber selbst gesetzt.

```
18 #define DRV_NAME         "msmst"
19 #define DRIVER_AUTHOR    "Thorsten Spaetling <schotter@ts-soft.info>"
20 #define DRIVER_DESC      "PCIe driver to read values from " \
21                          "the FPGA on Kontron's MSMST."
22 #define DRIVER_VER       "0.2"
```

Die nachfolgende Struktur deklariert den Platz der von den Sechstorwerten benötigt wird. Die vom ADC gelieferten Werte sind zwar nur 12 Bit breit, jedoch gibt es keinen 12 Bit Datentyp unter C und so wird die nächst größere Datenart genutzt.

```
23 struct sixport_values {
24   u16 s1q1;
25   u16 s1q2;
26   ..
27   u16 s4q4;
28 } __attribute__((packed));
```

In der Struktur `msmst_dev` sind alle Treiber- und Geräteinformationen hinterlegt und diese wird im Treiber selbst initialisiert und genutzt werden. Gerätetreiber für PCI(e)-Geräte müssen einen Zeiger (*pointer*) auf eine `pci_dev` Struktur haben, um die PCI(e)-Peripherie abfragen und verändern zu können.

```
29 struct msmst_dev {
30   struct pci_dev *pci_dev;
```

Die Eigenschaften der einzelnen Speicherbereich wie deren Beginn, Länge, Ende und Zustand wird in den nachstehenden Variablen festgehalten.

```
31   void __iomem bar[BAR_NUM];
32   unsigned long bar_start[BAR_NUM];
33   unsigned long bar_end[BAR_NUM];
34   unsigned long bar_length[BAR_NUM];
35   unsigned long bar_flags[BAR_NUM];
36   int mem_requested[BAR_NUM];
```

Im Abschnitt 4.4 wurde erwähnt, dass ein DMA-Controller Daten in den Hauptspeicher schreiben kann. Das Betriebssystem schränkt diese Zugriffe jedoch aus reinem Selbstschutz ein. Man muss daher über eine Funktion den DMA-Zugriff anmelden. Der Rückgabewert dieser wird in der nächsten Variable gespeichert und bei Entladen des Kernel-Moduls muss der erlaubte DMA-Zugriff mittels dieser Variable zurückgenommen werden.

```
37    dma_addr_t dma_addr;
```

Die nächsten Variablen sind anhand ihres Namens bereits ausreichend beschrieben.

```
38    int msi_enabled;
39    int irq_line;
40    int irq_pin;
41    int irq_counter;
```

Der Zugriff auf die Sechstorwerte wird über eine Gerätedatei erfolgen und diese kann entweder vom Typ *block device* oder *char device* sein. Bei ersterem ist zeitgleich der Inhalt des ersten sowie des letzten Datenblocks bekannt und bei letzterem kommen die Daten Stück für Stück an. Festplatten oder Arbeitsspeicher entsprechen einem *block device* und Bildschirme oder die serielle Schnittstelle werden als *char device* verstanden. Für das *char device* muss daher eine Struktur angelegt werden und die Gerätedatei benötigt noch eine eindeutige Kennung, die in `devt` festgehalten wird.

```
42    struct cdev cdev;
43    dev_t devt;
```

Die Zugriffskontrolle wird normalerweise durch ein Semaphore geregelt. Das ist eine Datenstruktur, die über Methoden reserviert und freigegeben werden kann und so den Zugriff organisiert. Der Begriff Semaphore stammt ursprünglich aus der Bahntechnik und bezeichnet ein Formsignal, das anzeigt, ob ein Gleisabschnitt befahren werden darf oder nicht. Ihr primärer Verwendungszweck ist daher das Vermeiden von *race conditions*, das sind Situationen in denen z.B. eine Variable von zwei verschiedenen Prozessen gleichzeitig verändert wird, wodurch unweigerlich eine Änderung verloren geht, siehe Abbildung 5.1. Besonders im Kontext von Mehrkernprozessoren sind Semaphoren unabdingbar. In `dev_opened` wird jedoch lediglich festgehalten, ob die Gerätedatei bereits von einem Anwendungsprozess geöffnet wurde oder nicht. Auf diese Weise wird ein Gerätezugriff von mehr als einer Anwendung unterbunden.

```
44    int dev_opened;
```

Am Ende der Struktur sind die privaten Daten willkürlich platziert

```
45    struct sixport_values values;
46    u8 tlv;
47    u8 mcp;
48 };
```

und am Ende der Datei schließt der von `#ifndef` aufgespannte Block.

```
49 #endif
```

*msmst.h* ist besprochen, weiter geht es mit *msmst.c*. Den Anfang machen wieder Präprozessoranweisungen über die Kernelbibliotheken geladen werden. `kernel.h`

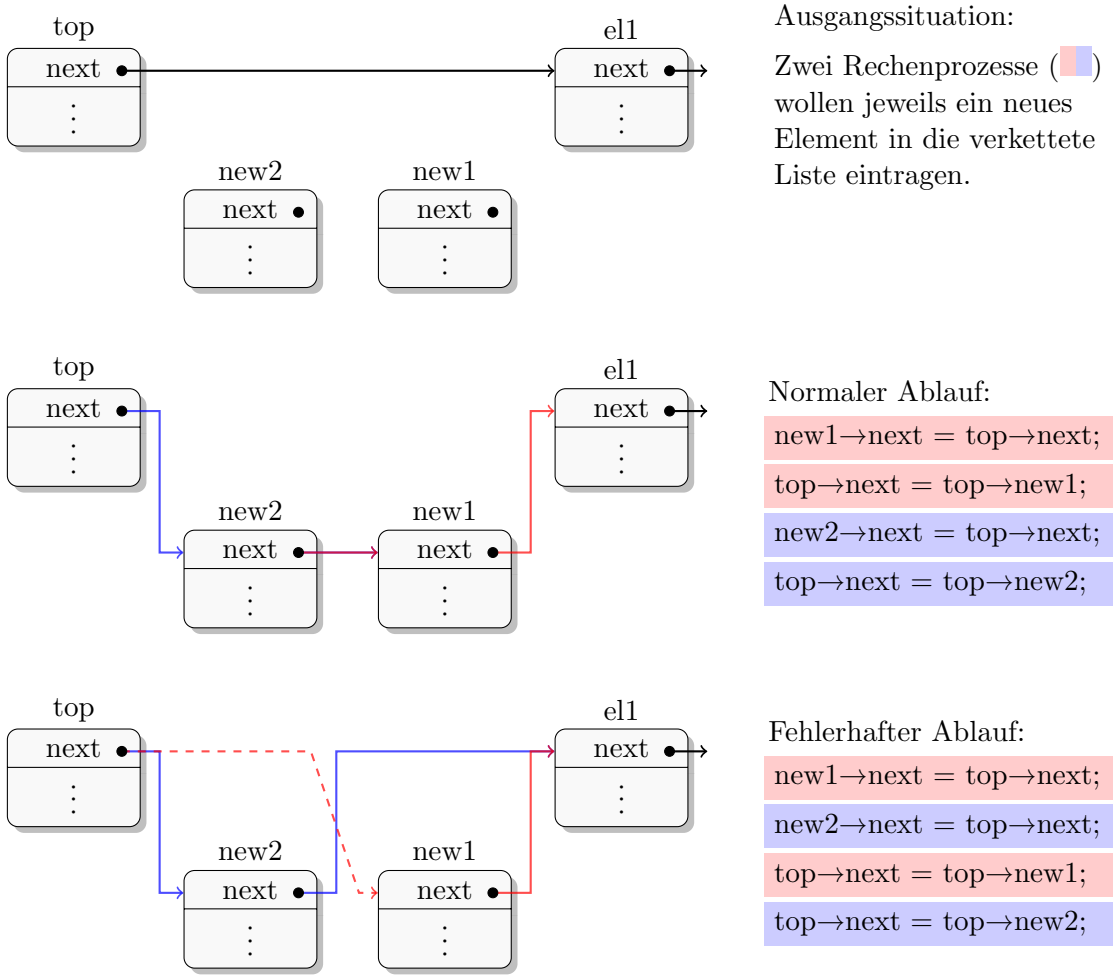



Abb. 5.1: *Race Condition* [17, S. 196]

führt hauptsächlich die Funktion `printk()` ein, die große Ähnlichkeit mit `printf()` hat, jedoch ist letztere nicht im Kernelkontext nutzbar. Mit `printk()` können Ausgaben generiert werden, die mit dem Programm `dmesg` gelesen werden können. Die einzelnen Nachrichten können kategorisiert werden und die zur Auswahl stehenden Kategorien sind in `kern_levels.h` hinterlegt.

```
1 #include <linux/kernel.h>
```

Treiber können entweder fest in den Kernel verbaut werden oder als Modul nachgeladen werden. Der Vorteil eines Moduls besteht darin, dass es nicht nur geladen- sondern auch entfernt werden kann. Das erleichtert die Entwicklung erheblich, denn so können zügig Änderungen getestet werden, während im Falle eines statischen Kernels dieser erst komplett neu kompiliert werden muss und zum Testen

ein Neustart erforderlich ist.

Um den Treiber als ein Modul übersetzen zu lassen, ist es nötig `module.h` einzubinden. Dort ist das Korsett definiert, das an den Treiber angelegt wird und ihn als Modul klassifiziert. Manche Funktionen und Daten eines Treibers werden nur einmalig verwendet und verbrauchen nach ihrem Aufruf sozusagen unnötig Speicher. Da der Speicherbereich in dem der Kernel liegt (*Kernel Space*) relativ klein dimensioniert ist, macht es daher Sinn den Speicher den diese Funktionen benötigen nach ihrer Ausführung freizugeben. Zur Markierung verwendet man die mit `init.h` eingeführten Attribute (`__init`, `__initdata` oder `__exitdata`).

```
2 #include <linux/module.h>
3 #include <linux/init.h>
```

Damit der Kernel beim Laden eines Moduls weiß, welche der enthaltenen Funktion den Treiber initialisiert, muss sie angegeben werden. Dazu bedient man sich des Makros `module_init()` und übergibt ihm die entsprechende Funktion. Da am Ende des Treibercodes alle Funktionen bekannt sind, findet man den genannten Makroaufruf i.d.R. dort. Analoges gilt für `module_exit()`.

```
500 module_init(msmst_init);
501 module_exit(msmst_exit);
```

Die Funktionen selbst sind nicht sonderlich spannend. Es wird lediglich die PCI-Treiberstruktur `msmst_driver` beim PCI-Manager registriert. Anhand dieser kann entschieden werden welcher Treiber welche Hardware bedienen kann. Sie wird im Abschnitt 5.2 noch genauer besprochen.

```
505 static int __init msmst_init(void) {
506   return pci_register_driver(&msmst_driver);
507 }
```

Das Gegenstück meldet den Treiber vom PCI-Manager ab.

```
508 static void __exit msmst_exit(void) {
509   pci_unregister_driver(&msmst_driver);
510 }
```

Die Modulinformationen, die in das Modulkorsett eingeflochten werden, müssen nicht zwingend am Ende stehen, jedoch sind sie dort im Allgemeinen anzutreffen.

```
515 MODULE_LICENSE("GPL");
516 MODULE_AUTHOR(DRIVER_AUTHOR);
517 MODULE_DESCRIPTION(DRIVER_DESC);
518 MODULE_VERSION(DRIVER_VER);
```

Da es sich um einen Gerätetreiber handelt, der auf PCIe-Peripherie zugreift und die von dort erhaltenen Daten über ein *char device* Applikationen anbietet, müssen die

folgenden Dateien inkludiert werden. `fs.h` definiert Zugriffsarten und `uaccess.h` erlaubt das Kopieren von Daten aus dem *Kernel* in den *User Space* und in die Gegenrichtung.

```
4 #include <linux/device.h>
5 #include <linux/pci.h>
6 #include <linux/cdev.h>
7 #include <linux/fs.h>
8 #include <asm/uaccess.h>
```

Beim Auftreten von Fehlern soll der Anwendung gesagt werden können, um welchen Fehlertyp es sich handelt, damit diese entsprechend reagieren kann. Erfolgreiche Funktionsaufrufe geben unter Linux die Zahl 0 zurück und im Fehlerfall eine negative. Die in `errno.h` definierten Fehlercodes entsprechen allerdings einer positiven Ganzzahl. Deswegen wird den Fehlermakros im Programmcode ein Minuszeichen vorangestellt.

```
9 #include <linux/errno.h>
```

Die ursprünglichen Datentypen (`char`, `int`, `float`, ...) der Programmiersprache C sind plattformabhängig, d.h. die Länge des Datentyps Integer variiert, je nach CPU-Architektur (x86, alpha, arm, ...) und Bitbreite (64, 32, ...). Um sich das Leben einfacher zu machen, wurden im C99 Standard Datentypen eingeführt, die auf jeder Plattform den ihrer Benennung nach nötigen Platz brauchen. Sie werden über `types.h` bekannt gemacht.

```
10 #include <linux/types.h>
```

Ursprünglich sollte der Treiber durch einen Interrupt über neue Daten informiert werden. Umgesetzt werden konnte dieses Vorhaben aufgrund des Hardwaredefekts nicht, jedoch sind im Treiber alle nötigen Vorkehrungen getroffen, damit dieser mit Interrupts arbeiten kann.

```
11 #include <linux/interrupt.h>
```

Zuletzt wird die modulspezifische Header-Datei inkludiert, deren Inhalt im nächsten Abschnitt sogleich Verwendung finden wird.

```
12 #include "msmst.h"
```

Jedes Gerät wird über die Struktur `device` verwaltet. In ihr sind eine Vielzahl von Informationen festgehalten. Unter anderen von welchem Treiber es bedient wird, um was für ein Gerät es sich handelt und an welchem Bus es eventuell angeschlossen ist.

```
13 static struct device *msmst_dev;
```

## 5.2 PCIe

Es mag verwundern, dass keine PCIe spezifischen Bibliotheken eingebunden wurden, dafür gibt es allerdings einen einfachen Grund. Beim Design von PCIe wurde viel Wert auf Abwärtskompatibilität zu PCI gelegt und dieser Gedanke wurde von den Linux-Kernel-Entwicklern aufgegriffen und weiterverfolgt. Daher können mit ein und der selben API ältere PCI- als auch aktuelle PCIe-Geräte gehandhabt werden.

Im Abschnitt 4.1 wurde besprochen anhand welcher Register die Hardware identifiziert wird. Es ist daher nicht verwunderlich die Identifikationsmerkmale in einem Treiber anzutreffen. Sie werden in eine konstante Struktur eingetragen, die mit dem Schlüsselwort `static` und dem Attribut `__devinitdata` versehen ist. Das Attribut erfüllt den gleichen Zweck wie die schon genannten und mittels `static` verliert die Struktur erst mit dem Entfernen des Moduls ihre Gültigkeit.

```
20 static const struct pci_device_id msmst_ids[] __devinitdata = {
21   {
22   .class = MY_CLASSCODE,
23   .class_mask = 0x000000,
24   .vendor = MY_VENDOR_ID,
25   .device = MY_DEVICE_ID,
26   .subvendor = PCI_ANY_ID,
27   .subdevice = PCI_ANY_ID,
28   .driver_data = 0
29   }, { 0, /* end: all zeroes */ }
30 };
```

In der vorhergehenden Struktur wurde zwar festgehalten zu welcher Hardware der Treiber passt, was jedoch bei der Detektion von einem passenden Gerät passiert, ist noch nicht festgelegt. Das übernimmt die `pci_driver` Struktur, die den Namen, die Identifikationsmerkmale und zwei aufzurufende Funktionen speichert. Die `probe` Funktion wird ausgeführt, sobald die passende Hardware gefunden bzw. angesteckt wurde und die `remove`-Funktion, wenn diese entfernt wurde.

```
31 static struct pci_driver msmst_driver = {
32   .name = DRV_NAME,
33   .id_table = msmst_ids,
34   .probe = probe,
35   .remove = __devexit_p(remove)
36 };
```

Der letzte Schritt informiert das System über den PCI-Gerätetreiber.

```
37 MODULE_DEVICE_TABLE(pci, msmst_ids);
```

Auf den nächsten Seiten werden die Interna der Treiberfunktionen behandelt. Dabei wird das Augenmerk auf die genutzte Kernel-API gelegt und für selbsterklärend erachtete Codestücke nicht genauer besprochen. Den Anfang macht die `probe`-Funktion deren Aufgabe es ist die erkannte PCI-Hardware für die Verwendung vorzubereiten. Dazu werden ihr zwei Strukturen übergeben. In der konstanten stehen die Identifikationsmerkmale der Hardware (Vendor ID, Device ID, ...) und in der veränderlichen stehen viele weitere Daten mit denen ein PCI-Gerät im Kernel beschrieben wird.

```
100 static int __devinit probe(struct pci_dev *dev, \
101                           const struct pci_device_id *id) {
102   u64 dma_req_mask;
103   int ret = 0;
```

Die Adresse der übergebenen `pci_dev`-Struktur wird in Zeile 110 in der treibereigenen festgehalten, für diese muss jedoch erst Speicherplatz angefordert werden. `kzalloc()` übernimmt diese Aufgabe, indem der Funktion als ersten Parameter die gewünschte Größe und als zweiten die Art und Weise wie der Speicherbereich beschafft werden soll übergeben wird. Zur Auswahl stehen `GFP_ATOMIC` und `GFP_KERNEL`, die sich prinzipiell nur in ihrer der Hartnäckigkeit unterscheiden. `GFP_ATOMIC` muss innerhalb weniger CPU-Takte Speicher gefunden haben, da die Suche *atomic*, d.h. unterbrechungsfrei (Interrupts sind gesperrt), abläuft und solche Umstände kurz zu halten sind. `GFP_KERNEL` nimmt sich in einem unterbrechbaren Kontext hingegen mehr Zeit und befriedigt mit größerer Wahrscheinlichkeit die Anfrage.

```
104   struct msmst_dev *msmst = NULL;
105   msmst = kzalloc(sizeof(struct msmst_dev), GFP_KERNEL);
```

Konnte wider erwarten nicht genügend Speicher alloziert werden, zeigt `msmst` auf `null` und die folgende Abfrage ist wahr. Daraufhin wird eine Fehler protokolliert und an die Sprungstelle `failed_alloc` gesprungen. Sprünge sind in der Anwendungsprogrammierung verpönt, im Kernel-Kontext findet man sie allerdings nahezu überall, da – bei richtiger Anwendung – schlanker und damit schneller Maschinencode entsteht.

```
106   if(!msmst) {
107     printk(KERN_ERR DRV_NAME": Couldn't kzalloc()ate memory.\n");
108     goto failed_alloc;
109   }
```

Auf die schon erwähnte Adresszuweisung folgt der Aufruf von `dev_set_drvdata()`, der die treiberspezifische Struktur `msmst` an das Gerät bindet.

```
110   msmst->pci_dev = dev;
111   dev_set_drvdata(&dev->dev, msmst);
```

Im Anschluss werden durch `pci_enable_device()` *low-level* Operationen getätigt mit denen die Hardware für die Verwendung vorbereitet wird.

```
112   ret = pci_enable_device(dev);
113   if(ret) {
114     printk(KERN_DEBUG DRV_NAME": pci_enable_device() failed\n");
115     goto failed_enable;
116   }
```

Das Gerät scheint funktionstüchtig zu sein und so kann ihm gestattet selbst Pakete verschicken zu dürfen. Für die Verwendung von DMA ist dieser Aufruf unabdingbar.

```
117   pci_set_master(dev);
```

Da die Peripherie nun im Stande ist selbst Pakete zu verschicken, wird sie und der *root complex* im nächsten Schritt auf den Versand von paketbasierten Interrupts (MSI) eingestellt.

```
118   ret = pci_enable_msi(dev);
119   if(ret) {
120     printk(KERN_DEBUG DRV_NAMEPRNT"Could not enable MSI.\n");
121     msmst->msi_enabled = 0;
122   } else {
123         printk(KERN_DEBUG DRV_NAMEPRNT"Enabled MSI.\n");
124         msmst->msi_enabled = 1;
125   }
```

Wie in Abschnitt 4.4 dargestellt, greift ein DMA-Controller auf die an ihn übergeben Adressen zu und liest von diesen bzw. beschreibt sie. Begrenzt werden der Adressbereich durch die Adressierbarkeit des Controllers. Handelt es sich um einen 32 Bit DMA-Controller so kann dieser eben auf keine Speicheradresse zugreifen, die außerhalb seines Adressraumes liegt. Mit `dma_get_required_mask()` wird dieser abgefragt und im nächsten Schritt protokolliert.

```
126   dma_req_mask = dma_get_required_mask(msmst_dev);
127   printk(KERN_DEBUG DRV_NAME": required dma mask: %lx\n", \
128                                 (unsigned long) dma_req_mask);
```

Das Ergebnis aus der vorherigen Abfrage wird nicht weiter genutzt, da der unterstützte Adressraum bekannt ist und vom nachfolgenden Try-And-Error-Konstrukt abgedeckt wird.

```
129   if (pci_set_dma_mask(dev, DMA_BIT_MASK(64)) == 0) {
130     printk(KERN_DEBUG DRV_NAME":64-bit DMA mask.\n");
131   } else {
132     if (pci_set_dma_mask(dev, DMA_BIT_MASK(32)) == 0) {
133       printk(KERN_DEBUG DRV_NAME":32-bit DMA mask.\n");
```

```
134       } else {
135         printk(KERN_DEBUG DRV_NAME": no suitable DMA.\n");
136         ret = -1;
137         goto failed_mask;
138       }
139     }
```

Damit das Betriebssystem dem DMA-Controller gestattet in gewisse Hauptspeicherbereiche seine Daten zu schreiben, muss es über die betroffenen Speicherbereiche unterrichtet werden. `pci_map_single()` gibt an welches Gerät, an welche Stelle im Speicher, wie viele Daten schreiben bzw. lesen will. Der letzte Parameter beschränkt den Controller auf Schreiboperationen. Leseoperationen (`PCI_DMA_TODEVICE`) oder beides (`PCI_DMA_BIDIRECTIONAL`) sind auch möglich.

```
140     msmst->dma_addr = pci_map_single(msmst->pci_dev, \
141       &msmst->values, sizeof(msmst->values), PCI_DMA_FROMDEVICE);
```

Die Funktion `pci_map_single()` trifft keinerlei Aussage darüber, ob sich erfolgreich war oder nicht. Dazu bedarf es eines extra Aufrufes von `pci_dma_mapping_error()`, um Gewissheit zu bekommen.

```
142     if(pci_dma_mapping_error(msmst->pci_dev, msmst->dma_addr)) {
143       printk(KERN_ERR DRV_NAME": failed to map memory!\n");
144       ret = -ENOMEM;
145       goto failed_dmamem;
146     }
```

In vorherigen Schritten wurde MSI zwar aktiviert, jedoch weiß das Betriebssystem noch nicht, dass es im Falle eines Interrupts die Interruptroutine dieses Treibers aufrufen soll. In der PCI-Gerätestruktur `pci_dev` ist der Konfigurationsbereich des PCI-Geräts komplett abgebildet und so kann die Interrupt-Leitung über diese bezogen werden. Die Interruptroutine wird durch `request_irq()` gesetzt. Bei den übergebenen Parametern handelt es sich um die Interrupt-Leitung, der Interruptroutine, die Art des Interrupts, den Treibernamen und der Adresse der Treiberdaten. Sämtliche Interruptarten sind in *interrupt.h* definiert, jedoch teilen sich – aus schon genannten Gründen – alle PCI-Geräte eine Interruptleitung womit `IRQF_SHARED` gewählt werden muss.

```
147     msmst->irq_line = dev->irq;
148     ret = request_irq(msmst->irq_line, msmst_msi, IRQF_SHARED,\
149                       DRV_NAME, (void *)msmst);
150     if(ret) {
151       printk(KERN_DEBUG DRV_NAME": Couldn't request"
152           " IRQ #%d, error %d\n", msmst->irq_line, ret);
153       msmst->irq_line = -1;
154       goto failed_irq;
```

```
155     }
```

Bei `msmst_msi()` als auch bei `map_bars()` handelt es sich um eigene Funktionen, die späteren Verlauf noch erklärt werden.

```
156     ret = map_bars(msmst, dev);
157     if(ret)
158       goto failed_map;
```

Gleiches gilt für `cdev_setup()` in der das *char device* angelegt wird. Hier sei auf den Abschnitt 5.3 verwiesen.

```
159     ret = cdev_setup(msmst);
160     if(ret)
161       goto failed_cdev;
```

Wird der Treibercode bis hierher ausgeführt, bedeutet dies, dass keine Probleme aufgetreten sind. In allen anderen Fällen wäre die Initialisierung abgebrochen worden und das PCI-Gerät wäre nicht einsatzfähig. Die erfolgreiche Inbetriebnahme wird durch Rückgabe einer Null an den Kernel signalisiert.

```
162     ret = 0;
163     printk(KERN_DEBUG DRV_NAME": probe()_successful.\n");
164     goto end;
```

Tritt bei der Initialisierung ein Fehler auf, werden die bis zu diesem Punkt vorgenommenen Einstellungen zurück genommen. Die einzelnen Speicherbereiche werden aufgehoben,

```
165   failed_cdev:
166     unmap_bars(msmst, dev);
```

die Interruptroutine wird entbunden,

```
167   failed_map:
168     if(msmst->irq_line >= 0)
169       free_irq(msmst->irq_line, (void *)msmst);
```

und die Interrupts werden mit dem PCI-Gerät selbst abgeschaltet.

```
170   failed_irq:
171     if(msmst->msi_enabled)
172       pci_disable_msi(dev);
173     pci_disable_device(dev);
```

Danach wird die Erlaubnis für den DMA-Zugriff auf den Speicherbereich zurückgenommen,

```
174   failed_dmamem:
175     pci_unmap_single(dev, msmst->dma_addr,\
176           sizeof(msmst->values), PCI_DMA_FROMDEVICE);
```

der von der Treiberstruktur genutzte Speicher freigegeben

```
177 failed_mask:
178 failed_enable:
179     if (msmst)
180       kfree(msmst);
```

und der entsprechende Fehlercode an den Kernel geliefert.

```
181 failed_alloc:
182 end:
183   return ret;
184 }
```

Damit ist die `probe`-Funktion nahezu besprochen, es fehlen noch die aus ihr heraus aufgerufenen Funktionen `msmst_msi()`, `map_bars()` und das Gegenstück `unmap_bars()`. Den Anfang macht die Interruptroutine.

Das Betriebssystem übergibt ihr zum einen die Nummer des ausgelösten Interrupts und zum anderen einen Adresszeiger. Dieser deutet auf die Adresse, die bei der Funktion `request_irq()` als fünfter Parameter übergeben wurde. Damit kann während der Interruptbehandlung auf treiberspezifische Daten zugegriffen werden.

```
200 static irqreturn_t msmst_msi(int irq, void *dev_id)
201 {
```

Der Zeiger wird jedoch nur auf die zugehörige Adresse gesetzt, wenn der Interrupt von der zum Treiber gehörigen Hardware ausgelöst wurde. Ansonsten ist `*dev_id` ein Nullzeiger und in diesem Fall muss die Routine verlassen werden, ohne den Interrupt zu löschen.

```
202   if(!dev_data)
203     return IRQ_NONE;
```

Hat jedoch alles seine Richtigkeit, kann die Routine ihrer Bestimmung nachgehen und den Interrupt nach dessen Behandlung löschen.

```
204   return IRQ_HANDLED;
205 }
```

Netterweise wird die Interruptleitung für die Dauer des Funktionsaufrufs ausmaskiert, so dass von anderen Geräten, die an der gleichen Leitung hängen, keine weiteren Interrupts ausgelöst werden können. Das erleichtert die Interruptbehandlung erheblich!

Mit `map_bars()` werden – wie der Name vermuten lässt – die einzelnen BARs des PCI-Geräts abgefragt und mit Speicherbereichen im Arbeitsspeicher verknüpft. Bei der Abfrage wird nacheinander von jedem BAR

```
220 static int __devinit map_bars(struct msmst_dev *msmst,\
221                                 struct pci_dev *dev) {
222   int ret=0, i=0;
223
224   for(i=0; i<BAR_NUM; i++) {
225     msmst->bar_end[i]=0;
226     msmst->bar_flags[i]=0;
```

der Begin,

```
227     msmst->bar_start[i] = pci_resource_start(dev, i);
228
229     if(msmst->bar_start[i]) {
```

das Ende, die Länge und der Zustand der Ressource bestimmt und anschließend zu Protokoll gegeben.

```
230       msmst->bar_end[i] = pci_resource_end(dev, i);
231       msmst->bar_length[i] = pci_resource_len(dev, i);
232       msmst->bar_flags[i] = pci_resource_flags(dev, i);
233       printk(KERN_DEBUG DRV_NAME":BAR[%d] 0x%08lx-0x%08lx "\
234              "flags 0x%08lx\n", i, msmst->bar_start[i],\
235              msmst->bar_end[i], msmst->bar_flags[i]);
236     }
```

BARs der Länge null werden übersprungen

```
237     if (!msmst->bar_length[i])
238       continue;
```

und die anderen werden auf ihren Typ (I/O oder Memory) hin untersucht und im Hauptspeicher abgebildet. Dazu wird den jeweiligen Funktionen die Startadresse und die Größe der Ressource mitgeteilt.

```
239     if(msmst->bar_flags[i] & IORESOURCE_IO) {
240       if(request_region(msmst->bar_start[i], msmst->bar_length[i],
241                                              DRV_NAME) == NULL) {
242         printk(KERN_DEBUG DRV_NAME":BAR[%d] is busy\n", i);
243         ret = -ENOMEM;
244         goto fail;
245       } else
246         msmst->mem_requested[i] = 1;
247     } else if(msmst->bar_flags[i] & IORESOURCE_MEM) {
248       msmst->bar[i] = ioremap_nocache(msmst->bar_start[i],
249                                       msmst->bar_length[i]);
250       if(msmst->bar[i] == NULL) {
251         printk(KERN_DEBUG DRV_NAME":BAR[%d] no mem\n", i);
```

```
252             ret = -ENOMEM;
253             goto fail;
254           } else
255             printk(KERN_DEBUG DRV_NAME": mapping bar %d as mem\n", i);
256
257           if(request_mem_region(msmst->bar_start[i], msmst->bar_length[i],
258                                                 DRV_NAME) == NULL) {
259             printk(KERN_DEBUG DRV_NAME":BAR[%d] is busy (mem)\n", i);
260             ret = -ENOMEM;
261             goto fail;
262           } else
263             msmst->mem_requested[i] = 1;
```

Soweit keine unvorhersehbaren Komplikationen auftreten.

```
264       } else {
265         printk(KERN_ERR DRV_NAME": neither IO nor mem region."\
266               " Can't handle this :(\n");
267       }
268     }
```

Passt jedoch soweit alles, wird die Funktion verlassen und ihr erfolgreicher Aufruf durch Rückgabe einer 0 signalisiert.

```
264     ret = 0;
265     goto success;
```

Analog zur `probe()`-Funktion werden im Fehlerfall alle bis dahin gemachten Einstellungen zurückgenommen.

```
267 fail:
268     unmap_bars(msmst, dev);
269 success:
270     return ret;
271 }
```

Zu guter Letzt fehlt noch das Gegenstück zu `map_bars()`. Es löst wie erwartet die Zuordnungen zwischen Arbeitsspeicher und Ressourcen. Das geschieht Stück für Stück,

```
280 static void unmap_bars(struct msmst_dev *msmst,\
281                         struct pci_dev *dev) {
282     int i=0;
283     for (i=0; i<BAR_NUM; i++) {
284       if (msmst->mem_requested[i] == 1) {
285         iounmap(msmst->bar[i]);
286         msmst->bar[i] = NULL;
```

```
287         }
288     }
289 }
```

es geht jedoch schneller durch einen Aufruf von `pci_release_regions()`.

## 5.3 Char Device

Im vorhergehenden Abschnitt wurde erklärt wie die Hardware aktiviert und eingebunden wird. Für einen Zugriff auf die Messdaten aus dem *User Space* bedarf es allerdings noch einer Schnittstelle. Diese Aufgabe kann von dem schon erwähnten *char device* übernommen. Dieses unterscheidet sich kaum von einer gewöhnlichen Datei. Beide können geöffnet, gelesen, beschrieben und geschlossen werden. Nur im Anlegen und Löschen gibt es Unterschiede.

Der PCI-Part erzwang die Nutzung von PCI-Datenstrukturen, analog verlangt ein *char device* die Verwendung von Strukturen, die seinesgleichen beschreiben. Mit Struktur `file_operations` ist diese Anforderung erfüllt. Hierbei handelt es sich im Grunde genommen, um eine Angabe von Funktionen, die die jeweiligen Dateioperationen ausführen. Einzig die erste Zeile enthält keine Funktion, sondern stellt durch das Makro `THIS_MODULE` sicher, dass das Modul nicht aus dem Kernel entfernt wird solange es noch in Benutzung ist.

```
1 static struct file_operations msmst_fops = {
2     .owner   = THIS_MODULE,
3     .open    = cdev_open,
4     .release= cdev_close,
5     .read    = cdev_read,
6     .write   = cdev_write,
7 };
```

Für Geräte, die einer Gerätedatei bedürfen, muss ein Zeiger auf die `class`-Struktur angelegt werden. Ihr Nutzen wird bei ihrer Verwendung erklärt.

```
8 static struct class *msmst_class;
```

Gegen Ende der `probe()`-Funktion wird `cdev_setup()` aufgerufen, für deren Besprechung auf diesen Abschnitt verwiesen wird und nun erfolgt. Gerätetreiber liegen unter Linux im Verzeichnis */dev* und werden vom Kernel anhand ihrer Major Nummer identifiziert. Diese lässt sich, wie in Listing 5.2 gezeigt, in Erfahrung bringen, da sie in der fünften Spalte der Ausgabe von `ls -Al /dev` steht. In der

```
drwxr-xr-x 2 root root             80 12. Apr 10:57 dri
crw-rw---- 1 root root       13,   64 12. Apr 10:57 event0
crw-rw---- 1 root root       13,   65 12. Apr 10:57 event1
crw-rw---- 1 root root       13,   66 12. Apr 10:57 event2
crw-rw---- 1 root video      29,    0 12. Apr 10:57 fb0
crw------- 1 root root       10,  228 12. Apr 10:57 hpet
crw-rw-rw- 1 root root        1,    3 12. Apr 10:57 null
brw-rw---- 1 root disk        1,    0 12. Apr 10:57 ram0
brw-rw---- 1 root disk        1,    1 12. Apr 10:57 ram1
```

Listing 5.2: ls -Al /dev

sechsten Spalte steht die Minor Nummer anhand derer einzelne identische Geräte unterschieden werden können. Beide Zahlen sind 8 Bit breit, was zum Ergebnis hat, dass auf diese Art maximal 256 verschiedene Geräte bedient werden können. Diese Grenze kann durchaus erreicht werden, weshalb es unter den Kernel-Entwicklern seit längerem Bestrebungen gibt, dass sich Gerätetreiber Major Nummern teilen. Das soll jedoch nicht Gegenstand der gegenwärtigen Betrachtungen sein, die Major Nummern wurden erwähnt, weil sie für das Anlegen einer Gerätedatei notwendig sind.

Über `alloc_chrdev_region()` wird ein Speicherbereich für ein *char device* angefordert. In den ersten Parameter notiert die Funktion im Erfolgsfall die Major Nummer, der zweite bestimmt welche Minor Nummer genutzt werden soll und der dritte gibt an wie viele Gerätenummern insgesamt angelegt werden sollen. Bei einer Gerätedatei genügt eine Gerätenummer, womit die genutzte Minor Nummer gleich null zu setzen ist. Im vierten und letzten Parameter steht der Name des Treibers.

```
 9 static int cdev_setup(struct msmst_dev *msmst) {
10   int ret = 0;
11   ret = alloc_chrdev_region(&msmst->devt, 0, 1, DRV_NAME);
12   if(ret) {
13     printk(KERN_DEBUG DRV_NAME":unable to alloc chrdev region\n");
14     return ret;
15   }
```

`cdev_init()` initialisiert die *char device* Struktur `cdev` und verknüpft diese mit den Dateioperationsfunktionen, die in der `file_operations`-Struktur `msmst_fops` notiert sind.

```
16   cdev_init(&msmst->cdev, &msmst_fops);
```

Danach kann das Gerät mittels `cdev_add()` zum System hinzugefügt werden, wodurch es sofort nutzbar ist. Die ersten beiden Parameter wurden schon erklärt und beim letzten handelt es sich um die Anzahl vergebener Minor Nummern.

```
17   ret = cdev_add(&msmst->cdev, msmst->devt, 1);
```

```
18    if(ret) /* (ret < 0) := error  */
19      return ret;
```

Das *char device* ist an dieser Stelle zwar bereits nutzbar, jedoch wurde noch keine Gerätedatei angelegt über die auf dieses zugegriffen werden kann. In den frühen Jahren von Linux wurden die Gerätedateien über die API von `devfs`, einem kernelinternen Gerätemanager, erzeugt. Mittlerweile ist man allerdings dazu übergegangen die Geräteerstellung in den *User Space* zu verlagern. Es bedurfte dadurch eines neuen Gerätemanagers, der in Anlehnung an ***U**ser Space* `udev` heißt. Um ihn nutzen zu können, wird eine `class`-Struktur erstellt und ihre Speicheradresse dem in Zeile 8 deklarierten Zeiger zugewiesen.

```
20    msmst_class = class_create(THIS_MODULE, DRV_NAME);
21    if(IS_ERR(msmst_class)) {
22      printk(KERN_DEBUG DRV_NAME":no␣udev␣support\n");
23      return -1;
24    }
```

Mit den Informationen aus der `class`-Struktur wird im letzten Schritt die Gerätedatei */dev/msmst* und Einträge im `sysfs` angelegt. Bei letzterem handelt es sich um ein virtuelles Dateisystem über das auf Geräteinformationen zugegriffen werden kann.

```
25    msmst_dev = device_create(msmst_class, NULL, \
26                  msmst->devt, NULL, "%s", DRV_NAME);
27    printk(KERN_DEBUG DRV_NAME":char␣device␣created␣"\
28               "(%d,%d).\n", MAJOR(msmst->devt),\
29                MINOR(msmst->devt));
```

In früheren Tagen von Linux wurden die Geräteinformationen unter */proc* zugänglich gemacht. Allerdings sind anfangs keine Konventionen vereinbart worden und so hat sich über die Zeit ein unübersichtliches Dateienwirrwarr ergeben, das nicht mehr glattgezogen werden konnte. Als Ausweg hat man `sysfs` definiert und hier sind die Treiberdaten hierarchisch strukturiert unter */sys* zu finden.

Mit der Rückgabe von null wird der aufrufenden `probe()`-Funktion ein erfolgreicher Durchlauf signalisiert.

```
30    return 0;
31  }
```

Nachdem erklärt ist, wie die Gerätedatei angelegt wird, können die Funktionen besprochen werden, die einen Zugriff auf die Messdaten ermöglichen. Die Öffnen-Methode soll im Allgemeinen den Zugriff auf die Hardware und die Daten vorbereiten. Ihr werden zwei Strukturzeiger an die Hand gegeben. Über den ersten erhält man Zugriff auf eine `inode`-Struktur, die Informationen über eine Datei

(*/dev/msmst*) hält, und über den zweiten greift man auf eine `file`-Struktur zu. Diese wird vom Kernel beim Öffnen erzeugt und wird bis zum Schließen an jede Funktion übergeben, die auf der Datei operiert. Es bietet sich daher an in ihr einen Verweis auf die Gerätedaten zu hinterlegen.

```
40 int cdev_open(struct inode *inode, struct file *filp) {
41   struct msmst_dev *msmst;
```

Die Adresse der Messdaten ist in dieser Funktion jedoch nicht bekannt, weil die sie haltende Struktur, nicht im Sichtbarkeitsbereich der aktuellen Funktion liegt. Allerdings kann sie über das Makro `container_of()` in Erfahrung gebracht werden. Der Code des Makros ist schwer zu verstehen, jedoch kann die Arbeitsweise leicht nachvollzogen werden. In Abbildung 5.2 ist gezeigt, wie die an das Makro

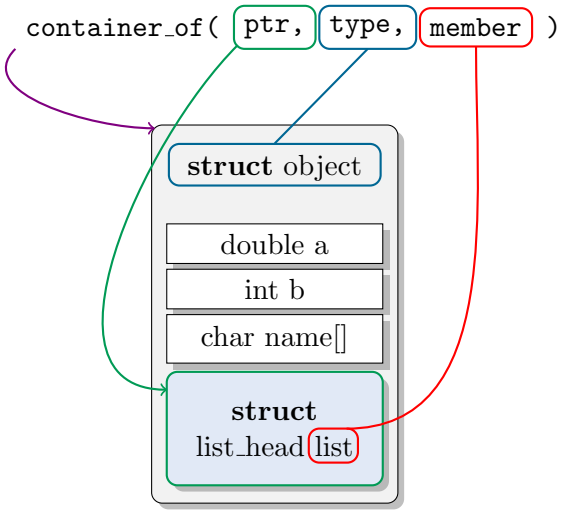



Abb. 5.2: container_of(ptr, type, member) [3]

übergebenen Parameter zueinander in Beziehung stehen müssen. An zweiter Position findet sich eine Strukturdefinition, in der wiederum eine weitere Struktur definiert ist und an erster findet sich die Speicheradresse der initialisierten „Unterstruktur“. Der dritte Parameter ist quasi als Name zu verstehen, über den die „Unterstruktur“ angesprochen werden kann. Zurückgegeben wird die Adresse an der sich die Gesamtstruktur befindet.

Die gewünschte Speicheradresse wird nun folgendermaßen beschafft. Anhand des ersten Parameters ist bekannt wo sich die Gesamtstruktur in etwa im Speicher befindet. Mit dem Namen der „Unterstruktur“ und dem Aufbau der Gesamtstruktur lässt sich der Abstand zwischen der „Unterstruktur“ und dem Kopf der Gesamtstruktur bestimmen. Das Makro lässt sich somit in ein Dekrementieren des ersten Parameters auflösen.

```
42   msmst = container_of(inode->i_cdev, struct msmst_dev, cdev);
```

Sollte dabei etwas schief gegangen sein, wird das mit der Überprüfung des Zeigers `msmst` auf einen Nullzeiger festgestellt und das Öffnen der Datei abgebrochen.

```
43   if(msmst == NULL)
44     return -ENODEV;
```

Wenn die Gerätedatei bereits von einer Anwendung geöffnet wurde, wird ebenso ein Fehlercode zurückgegeben.

```
45   if(msmst->dev_opened)
46     return -EAGAIN;
```

Erst danach wird die Adresse an den Dateizeiger `*filp` (*file pointer*) übergeben und durch Rückgabe einer Null das erfolgreiche Öffnen quittiert.

```
47   filp->private_data = msmst;
48   return 0;
49 }
```

Nach dem Öffnen folgt für gewöhnlich entweder ein Lese- oder ein Schreibzugriff. Im Grunde genommen handelt es sich in beiden Fällen um Kopiervorgänge. Beim Lesen einer Datei werden Daten von dieser in den Arbeitsspeicher kopiert und beim Schreiben dreht sich Vorgang entsprechend um. Dazu muss jeweils angegeben werden wie die Zieladresse für die Daten lautet und wie viel dorthin kopiert werden sollen. Damit nicht nur absolute Adressangaben gemacht werden müssen, kann mit dem vierten Parameter ein Offset übergeben werden.

```
60 ssize_t cdev_read(struct file *filp, char __user *buffer,\
61                           size_t count, loff_t *f_pos) {
62   unsigned long not_copied, to_copy;
```

In `cdev_open()` wurde die Speicheradresse, an der sich die Treiberstruktur befindet, dem Dateizeiger übergeben und in einem der ersten Schritte wird sie aus diesem wieder extrahiert.

```
63   struct msmst_dev *msmst = filp->private_data;
```

Im nächsten Schritt werden die Messwerte von der Peripherie einzeln ausgelesen. Dies könnte auch vom DMA-Controller übernommen werden, jedoch sind die Geschwindigkeitsvorteile bei der geringen Datenmenge nicht erwähnenswert.

```
64   msmst->values.s1q1 = ioread16(msmst->bar[0]+OFF_S1Q1);
65   msmst->values.s1q2 = ioread16(msmst->bar[0]+OFF_S1Q2);
66   ..
67   msmst->values.s4q4 = ioread16(msmst->bar[0]+OFF_S4Q4);
```

Bei älteren und weniger komplexen CPUs konnte garantiert werden, dass die Operationen in der Reihenfolge ausgeführt werden, wie sie im Code stehen. In moderneren Systemen ist dies hingegen nicht immer erfüllt, da die Befehlsreihenfolge aus Gründen der Performanz neu angeordnet worden sein kann. Soll nun garantiert werden, dass die bis hier her in der Funktion stehenden Funktionsaufrufe ausgeführt wurden, muss eine Speicherbarriere (*memory barrier*) gesetzt werden.

```
68    rmb();
```

Da nun mit Sicherheit neue Messdaten ausgelesen wurden, werden diese mit `copy_to_user()` vom *Kernel Space* in den *User Space* kopiert. Im Anschluss wird zurückgegeben wie viele Bytes nicht kopiert werden konnten, damit die Applikation z.B. einen weiteren Lesevorgang unternimmt.

```
69    to_copy = sizeof(msmst->values);
70    not_copied = copy_to_user(buffer, &msmst->values, to_copy);
71    return (ssize_t) (to_copy - not_copied);
72  }
```

Dem aufmerksamen Leser dürfte aufgefallen sein, dass in der Lesefunktion die Parameter `count` und `*f_pos` keine Verwendung finden und stattdessen immer 16 16 Bit-Werte kopiert werden. Das ist kein gutes Vorgehen, denn selbst wenn die Applikation nach nur bspw. vier Bytes fragt, werden dennoch 16 Bytes kopiert. Während Speicherzugriffsfehler in Anwendungen abgefangen werden, ist das im Kernel-Kontext nicht der Fall und so ist es denkbar, dass die zu viel kopierten 12 Bytes Schaden anrichten können. Dies wurde jedoch in der Anwendungsentwicklung bedacht.

Die Schreibfunktion ist im Aufruf ähnlich gestaltet wie die Lesefunktion, einzig der Adresszeiger `*buffer` ist jetzt unveränderlich.

```
80  ssize_t cdev_write(struct file *filp, const char __user \
81                 *buffer, size_t count, loff_t *f_pos) {
82    int ret = 0;
83    u8 syscfg_user[5];
84    u32 sysflags = 0;
85    struct msmst_dev *msmst = filp->private_data;
```

Nach der Initialisierung einiger Variablen und der Extraktion der Treiberdaten, wird überprüft ob zu viele Bytes übertragen werden sollen. Trifft dies zu wird ein Fehlercode zurückgegeben.

```
86    if(count > 5)
87      return -EINVAL;
```

Ansonsten werden die Daten in den *Kernel Space* kopiert, um mit ihnen arbeiten zu können.

```
88   ret = copy_from_user(&syscfg_user[0], buffer, count);
```

Wenn alle Bytes kopiert wurden, ist die Anzahl der noch zu kopierenden gleich null und der Kopiervorgang somit erfolgreich gewesen. An die Daten wird die in Tabelle 5.1 gezeigte Schablone angelegt. Mit den ersten 19 Bits werden die in

| 39 32 | 31 24 | 23 19 | 18 0 |
|---|---|---|---|
| MCP | TLV | X | SYS |

Tab. 5.1: Erwartetes Datenformat für Schreibvorgänge

Tabelle 3.2 definierten Zustände im FPGA gesteuert und mit den letzten beiden optionalen Bytes wird die Ausgangsspannung der DACs (TLV) bzw. der Wert der Potentiometer (MCP) eingestellt. Sie werden dazu mittels `iowrite8()` in bestimmte Speicherbereiche der PCIe-Einheit geschrieben. Es sei nochmals darauf hingewiesen, dass sich mit dem Schreiben der Bytes die Abläufe im FPGA nicht ändern. Erst wenn das SYS-Modul den Inhalt der Speicherbereiche abruft, werden die Anweisungen ausgeführt.

```
89   if(ret == 0) {
90     if(count == 5) {
91       msmst->mcp = syscfg_user[4];
92       iowrite8(msmst->mcp, msmst->bar[2]+OFF_RV1_5);
93       ..
94       iowrite8(msmst->mcp, msmst->bar[2]+OFF_RV2_10);
95     }
96     if(count >= 4) {
97       msmst->tlv = syscfg_user[3];
98       iowrite8(msmst->tlv, msmst->bar[2]+OFF_VA1);
99       ..
100      iowrite8(msmst->tlv, msmst->bar[2]+OFF_VB2);
101    }
```

Solange nur einzelne Bytes übertragen werden, treten keine Schwierigkeiten auf. Wenn jedoch die 19 Bit übertragen werden sollen, muss die Byte-Reihenfolge beachtet werden, da die x86 Architektur *little endian* und die PCIe-Schnittstelle *big endian* ist! Hier werden die Bits händisch verschoben, es hätte aber auch das Makro `cpu_to_be32` genutzt werden können.

```
102    sysflags = ( ((u32)(syscfg_user[2])) | \
103                 ((u32)(syscfg_user[1] << 8)) | \
104                 ((u32)(syscfg_user[0] << 16)) );
```

Für die Übertragung muss `iowrite32()` verwendet werden, da nur 8, 16 oder 32 Bit übertragen werden können. Sieht man genau hin, so fällt auf, dass der TLV-

Wert, wenn er vorhanden ist, bei dieser Operation mit übertragen wird. Im SOPC-Builder wurde jedoch die Speicherstelle auf 19 Bit beschränkt und die Bits 20 bis 31 sind somit gar nicht vorhanden. Im Anschluss wird durch eine Speicherbarriere die Ausführung der Funktionsaufrufe sichergestellt.

```
105       iowrite32(sysflags, msmst->bar[2]+OFF_SYSCONFIG);
106       rmb();
```

Sollte das Kopieren der Daten von der Benutzerumgebung in den Kernel-Kontext fehlgeschlagen sein, wird dies wieder mit einem Fehlercode kommuniziert.

```
107     } else
108         return -EINVAL;
```

Ansonsten wird an die Anwendung zurückgemeldet wie viele Bytes geschrieben wurden, das sind hier `count`. Auf diese Weise können Applikationen erkennen, ob der Schreibvorgang erfolgreich war oder ob er nochmals gestartet werden muss.

```
109     return (ssize_t) count;
110 }
```

Soll nicht länger vom Gerät gelesen bzw. auf dieses geschrieben werden, kann der Dateizugriff durch schließen der geöffneten Datei beendetwerden. Dabei kann nichts schief gehen. Die für den Dateizugriff verantwortliche Variable `dev_opened` wird auf null zurückgesetzt und die Funktion wird mit Rückgabe von null verlassen.

```
120 int cdev_close(struct inode *inode, struct file *filp) {
121   struct msmst_dev *msmst;
122   msmst = container_of(inode->i_cdev, struct msmst_dev, cdev);
123   if(msmst->dev_opened)
124     msmst->dev_opened = 0;
125   return 0;
126 }
```

Ohne Zugriffsbeschränkung würde der Inhalt von `cdev_close()`

```
130 return 0;
```

lauten.

# KAPITEL 6

## Applikation

Da nun die Messdaten ihren Weg in den Computer gefunden haben, bedarf es nur noch einer adäquaten Darstellung. Zuerst wurde ein Hintergrundprozess programmiert, der periodisch die Messdaten von */dev/msmst* liest und diese über das Netzwerk verteilt. Der Vorteil besteht in der damit verbundenen Auslagerung der Winkelberechnung, was die CPU-Last senkt, und für den Fall, dass die Hardware zu Messzwecken herangezogen wird, die ein großes Datenaufkommen zur Folge haben, setzt das System keine Schranken.

Darauf aufbauend wurde eine grafische Anwendung unter Verwendung der QT-Bibliotheken geschrieben, die ihre Daten über das Netzwerk von dem erwähnten Hintergrundprozess, dessen Quellcode über die nebenstehende Büroklammer bezogen werden kann, bezieht. Die QT-Bibliotheken sind zwar für jedes gängige Betriebssystem verfügbar, jedoch wurde letztlich eine einfachere Lösung präferiert.

Sie besteht aus einem Webserver und einem mit aktuellen Webtechnologien entworfenem Interface. Der Demonstrator lässt sich somit über einen normalen Browser bedienen und die Plattformunabhängigkeit ist garantiert. Einen Nachteil bringt diese Technik allerdings mit sich mit. Der Datenaustausch erfolgt asynchron, d.h. für zeitkritische Anwendungsbereiche steht dieser Ansatz nicht zur Verfügung. Strikte, zeitliche Hürden werden jedoch von den wenigsten Anwendungen gesetzt.

Das Gesamtpaket ist über die nebenstehende Büroklammer erhältlich. Wie es sich im einzelnen zusammensetzt, wird in den kommenden Abschnitten behandelt.

## 6.1 Webserver

Eine Webanwendung benötigt sinnigerweise einen Webserver, der die Webseiten ausliefert. Für diese Aufgabe gibt es eine Reihe von Kandidaten, jedoch bedurfte es beim Webserver `apache`[1] keiner längeren Einarbeitungszeit, da mit diesem schon Erfahrungen gesammelt wurden. Nach der Installation musste lediglich eine Konfigurationsdatei angelegt werden, deren Inhalt Listing 6.1 zeigt.

```
<VirtualHost *:80>
        ServerName msmst
        ServerAlias sixport
        DocumentRoot "/var/www/msmst/htdocs"
        <Directory "/var/www/msmst/htdocs">
                Options +ExecCGI
                AddHandler cgi-script .cgi
                AddType application/x-httpd-cgi .cgi
                AllowOverride All
                Order allow,deny
                Allow from all
        </Directory>
</VirtualHost>
```

Listing 6.1: /etc/apache2/vhosts.d/msmst.conf

Der Server nimmt wegen `*:80` Anfragen aus jeder Adressregion entgegen und fühlt sich angesprochen, wenn nach `msmst` oder `sixport` gefragt wird. Dafür muss in die *hosts*-Datei des eigenen Computers die IP-Adresse des Sechstor-Demonstrators gefolgt von „msmst“ eingetragen werden. Unter Linux liegt diese im Verzeichnis */etc* und bei Windows findet man sie unter *%SystemRoot%\system32\drivers\etc*. Die auszuliefernden Daten liegen im Verzeichnis `/var/www/msmst/htdocs/` und in diesem dürfen CGI-Skripte (***C****ommon* ***G****ateway* ***I****nterface*) ausgeführt werden.

Der Zugriff auf Dateien, die nicht im `DocumentRoot` liegen, kann aus Sicherheitsgründen nicht direkt aus der Webseite heraus erfolgen. Es muss ein Zwischenschritt über ein Skript gegangen werden. Den hierfür geschriebenen Perl-Skripten werden zwei bis drei Parameter geschickt, die diese an die jeweiligen Programme übergeben. Haben die Anwendungen eine Ausgabe, wird diese wiederum an die Webanwendung weitergereicht.

[1] http://www.apache.org/

## 6.2 Web Applikation

Bei Webseiten aus den Anfangszeiten des Internets handelte es sich i.d.R. um statische Konstrukte mit denen keine Interaktion möglich war. Das änderte sich bekanntermaßen Stück für Stück. Erst werteten auf dem Server laufende Skripte Benutzeranfragen aus und passten die auszuliefernde HTML-Datei an, ehe mit Einführung der im Webbrowser laufenden Skriptsprache JavaScript (JS) ist eine Änderung der Webseite auch auf Seite des Clients möglich wurde.

In dieser Sprache wurde die Bibliothek jQuery[2] entworfen mit der Inhalte einer Webseite komfortabel gestaltet und manipuliert werden können. Mit ihrer Hilfe wird die die Kommunikation mit den Perl-Skripten und das Timing gehandhabt. Neben jQuery wurde noch eine weitere JavaScript-Bibliothek genutzt namens Raphaël[3]. Durch sie ist es möglich veränderliche, skalierbare Vektorgrafiken (***S**calable **V**ector **G**raphics*) zu erzeugen. Genutzt wird diese Funktion bei der Animation der Tachonadel, mit der der gemessene Winkel illustriert wird.

Die Benutzeroberfläche ist in Abbildung 6.1 gezeigt und intuitiv gestaltet. Es werden zwei Betriebsmodi angeboten, „live“ und „track“, und über „config“ können Einstellungen vorgenommen werden. Wenn der Live-Modus gestartet ist, werden die Daten mit dem eingestellten Intervall abgerufen, ausgewertet und die grafische Anzeige bei Bedarf angepasst. Für den Track-Modus wird vorausgesetzt, dass sich die Sechstoranordnung auf der Rotationsplattform befindet. Denn in diesem Modus werden die Antennen durch Ansteuern der Motoren in die Richtung der Sendeanlage gebracht.

Die zur Motoransteuerung geschriebene Bibliothek, die unter Linux wie Windows genutzt werden kann, wird über die nebenstehende Büroklammer angeboten und enthält für beide Betriebssystem eine Beispielapplikation. Der Vollständigkeit halber sei erwähnt, dass es sich um Präzisionsrotationsmotoren PRS-110 der Firma PI miCos handelt.

[2] http://www.jquery.com/
[3] http://www.raphaeljs.com/

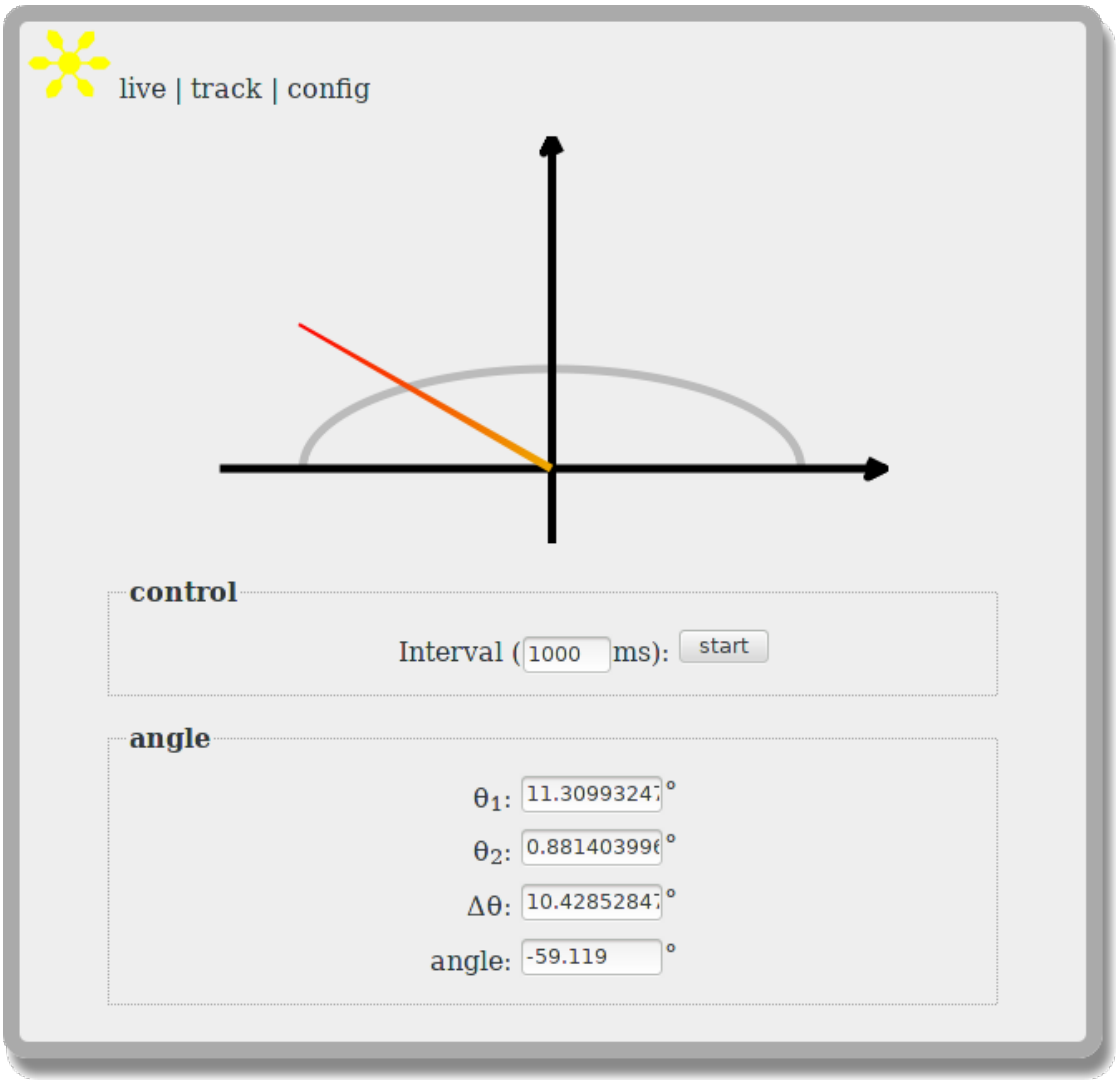



Abb. 6.1: Oberfläche der Applikation

# KAPITEL 7

# Ausblick

Wie in der Einleitung erwähnt, konnte das Vorhaben einen eigenständigen Demonstrator für die Winkeldetektion zu entwickeln, aufgrund des Hardwaredefekts nicht zum Abschluss gebracht werden. Die Hardware ist mittlerweile in Reparatur und dürfte nach Rücksendung unter zu Hilfenahme dieser Arbeit zügig in Betrieb genommen werden können.

Bisher konnte die Winkelmessung nur in einer Ebene durchgeführt werden, da es keine HF-Hardware gibt mit den beide Ebenen erfasst werden. Die grundlegenden Voraussetzungen, um die Messwerte aus der zweiten Ebene aufnehmen zu können, werden jedoch vom aktuellen Hardwareaufbau erfüllt. Es bedarf nur eines weiteren Steckers, der die in Tabelle A.2 notierte Pinbelegung abdeckt. Der Treiber liest standardmäßig die Messwerte von beiden Ebenen und auch die Applikation fragt nach allen Werten. Nur bei Darstellung müssten noch Anpassungen erfolgen.

Wenn man schon dabei ist die Anwendung anzupassen, kann sie gleich noch um die Möglichkeit ergänzt werden Daten aufzeichnen zu können. Beim Programmieren des Webinterfaces wurde dies vorgesehen, allerdings aus Zeitmangel zurückgestellt. Ein weiteres Feature, über dessen Implementierung laut nachgedacht werden sollte, ist eine Erweiterung der grafischen Benutzerschnittstelle um Skripte zur Automatisierung von Abläufen.

Zwischenzeitlich wurde noch mit dem Gedanken gespielt eine FFT (***F**ast **F**ourier **T**ransform*) in den FPGA einzuprogrammieren, da diese mit der eigentlichen Aufgabenstellung jedoch wenig gemein hat, blieb es bei einem Gedankenspiel.

Die Entwicklung des Demonstrators ist – wie durch die aufgezählten Ideen gezeigt wird – noch lange nicht abgeschlossen. Jetzt gilt es allerdings erst einmal zu hoffen, dass der Hersteller die Plattform reparieren kann.

# ANHANG A

## Pinbelegung

| | | | |
|---|---|---|---|
| 5V | 2 | 1 | GND |
| gpio_rxn(1) | 4 | 3 | gpio_rxp(1) |
| gpio_rxn(2) | 6 | 5 | gpio_rxp(2) |
| gpio_rxn(3) | 8 | 7 | gpio_rxp(3) |
| gpio_rxn(4) | 10 | 9 | gpio_rxp(4) |
| 3.3 V | 12 | 11 | GND |
| gpio_rxn(5) | 14 | 13 | gpio_rxp(5) |
| gpio_rxn(6) | 16 | 15 | gpio_rxp(6) |
| gpio_rxn(7) | 18 | 17 | gpio_rxp(7) |
| gpio_rxn(8) | 20 | 19 | gpio_rxp(8) |
| 3.3 V | 22 | 21 | GND |
| gpio_txn(1) | 24 | 23 | gpio_txp(1) |
| gpio_txn(2) | 26 | 25 | gpio_txp(2) |
| gpio_txn(3) | 28 | 27 | gpio_txp(3) |
| gpio_txn(4) | 30 | 29 | gpio_txp(4) |
| 3.3 V | 32 | 31 | GND |
| gpio_txn(5) | 34 | 33 | gpio_txp(5) |
| gpio_txn(6) | 36 | 35 | gpio_txp(6) |
| gpio_txn(7) | 38 | 37 | gpio_txp(7) |
| gpio_txn(8) | 40 | 39 | gpio_txp(8) |
| 3.3 V | 42 | 41 | GND |
| gpio_clkoutn | 44 | 43 | gpio_clkoutp |

Tab. A.1: GPIO Pinbenennung, *bottom view*

| DATA_IN | ADC1 | | | | J2 | ADC2 | ADC3 | | | | J3 | ADC4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | hsmc_clkin2n | 1 | 2 | hsmc_rxn(16) | | | hsmc_clkin1n | 1 | 2 | hsmc_rxn(7) | |
| | 1 | hsmc_clkin2n | 3 | 4 | hsmc_rxp(16) | | | hsmc_clkin1n | 3 | 4 | hsmc_rxp(7) | |
| | 2 | hsmc_txn(16) | 5 | 6 | hsmc_rxn(15) | | | hsmc_txn(7) | 5 | 6 | hsmc_rxn(6) | |
| | 3 | hsmc_txp(16) | 7 | 8 | hsmc_rxp(15) | | | hsmc_txp(7) | 7 | 8 | hsmc_rxp(6) | |
| | 4 | hsmc_txn(15) | 9 | 10 | hsmc_rxn(14) | | | hsmc_txn(6) | 9 | 10 | hsmc_rxn(5) | |
| | | 5 V | 11 | 12 | GND | | | 5 V | 11 | 12 | GND | |
| | 5 | hsmc_txp(15) | 13 | 14 | hsmc_rxp(14) | | | hsmc_txp(6) | 13 | 14 | hsmc_rxp(5) | |
| | 6 | hsmc_txn(14) | 15 | 16 | hsmc_rxn(13) | | | hsmc_txn(5) | 15 | 16 | hsmc_rxn(4) | |
| | 7 | hsmc_txp(14) | 17 | 18 | hsmc_rxp(13) | | | hsmc_txp(5) | 17 | 18 | hsmc_rxp(4) | |
| | 8 | hsmc_clkout2n | 19 | 20 | hsmc_rxn(12) | | | hsmc_clkout1n | 19 | 20 | hsmc_rxn(3) | |
| | 9 | hsmc_clkout2p | 21 | 22 | hsmc_rxp(12) | | | hsmc_clkout1p | 21 | 22 | hsmc_rxp(3) | |
| | 10 | hsmc_txn(13) | 23 | 24 | hsmc_rxn(11) | | | hsmc_txn(4) | 23 | 24 | hsmc_rxn(2) | |
| | 11 | hsmc_txp(13) | 25 | 26 | hsmc_rxp(11) | | | hsmc_txp(4) | 25 | 26 | hsmc_rxp(2) | |
| | EOC | hsmc_txn(12) | 27 | 28 | hsmc_rxn(10) | | | hsmc_txn(3) | 27 | 28 | hsmc_rxn(1) | |
| | | 3.3 V | 29 | 30 | GND | | | 3.3 V | 29 | 30 | GND | |
| | EOLC | hsmc_txp(12) | 31 | 32 | hsmc_rxp(10) | | | hsmc_txp(3) | 31 | 32 | hsmc_rxp(1) | |
| | RD | hsmc_txn(11) | 33 | 34 | hsmc_rxn(9) | | | hsmc_txn(2) | 33 | 34 | hsmc_rxn(0) | |
| | WR | hsmc_txp(11) | 35 | 36 | hsmc_rxp(9) | | | hsmc_txp(2) | 35 | 36 | hsmc_rxp(0) | |
| | CONVST | hsmc_txn(10) | 37 | 38 | hsmc_txn(9) | | | hsmc_txn(1) | 37 | 38 | hsmc_txn(0) | |
| | SCLK | hsmc_txp(10) | 39 | 40 | hsmc_txp(9) | | | hsmc_txp(1) | 39 | 40 | hsmc_txp(0) | |

Tab. A.2: THDB-HTG Pinbenennung, *top view*

# ANHANG B

## CD-Inhalt

```
|-- Ausarbeitung
|-- FPGA
|   |-- Code
|    -- Flashscript
|-- VHDL
|   |-- adc
|   |-- counter
|   |-- mcp
|   |-- meanvalue
|   |-- samplingclockmux
|   |-- switch
|   |-- sys
|   |-- tlv
|    -- updater
|-- KernelModul
|    -- TimerModul
|-- PRScom
|-- SysConfig
 -- WebInterface
```

In den meisten Verzeichnissen ist eine `README` Datei anzutreffen, die Informationen und Anweisungen enthält. Bei allen Archiven handelt es sich um `tar.gz`-Archive, die mit dem Aufruf `gzip -zxf archiv.tar.gz` entpackt werden können.

# Literaturverzeichnis


[1] Interfacing between LVPECL, VML, CML, and LVDS Levels. http://www.ti.com/lit/an/slla120/slla120.pdf. [Online; accessed 11-April-2013].

[2] Introduction to LVDS, PECL, and CML. http://pdfserv.maximintegrated.com/en/an/AN291.pdf. [Online; accessed 11-April-2013].

[3] Magical container_of() macro. http://linuxwell.com/2012/11/10/magical-container_of-macro/. [Online; accessed 14-April-2013].

[4] PCI Express payload required to be Big-Endian by specification. http://forums.xilinx.com/t5/PCI-Express/PCI-Express-payload-required-to-be-Big-Endian-by-specification/td-p/285551. [Online; accessed 09-April-2013].

[5] Prefetchable and Non-prefetchable memories. http://forums.xilinx.com/t5/PCI-Express/Prefetchable-and-Non-prefetchable-memories/td-p/30045. [Online; accessed 05-April-2013].

[6] Altera. *DMA Controller Core, Quartus II 9.1 Handbook*. [Online; accessed 11-April-2013].

[7] Altera. *IP Compiler for PCI Express - User Guide*.

[8] James E.J. Bottomley. Integrating DMA into the Generic Device Model. *Ottawa Linux Symposium*. [Online; accessed 11-April-2013].

[9] Brad Congdon. *Pci Express System Architecture* -. Addison-Wesley Professional, Boston, 2004.

[10] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers* -. O'Reilly Media, Inc., Sebastopol, CA, 3 edition, 2009.

[11] Erik.vikinger et al. Peripheral Component Interconnect. http://www.lowlevel.eu/wiki/Peripheral_Component_Interconnect. [Online; accessed 04-April-2013].

[12] Alexander Kölpin. *Der erweiterte Sechstor-Empfänger - Ein systemübergreifender Ansatz für Kommunikations- und Messaufgaben.* Südwestdeutscher Verlag, Saarbrücken, 2010.

[13] Stefan Lindner. Entwurf und Aufbau eines erweiterten Sechstor-Empfänger-Systems. Diplomarbeit, Friedrich-Alexander Universität Erlangen-Nürnberg, 10 2011.

[14] Sebastian Mann. 2-D Einfallswinkelbestimmung mit Hilfe von Sechstor-Technologie. Diplomarbeit, Friedrich-Alexander Universität Erlangen-Nürnberg, 07 2012.

[15] Jean P. Nicolle. PCI Express - Connector. http://www.fpga4fun.com/PCI-Express1.html. [Online; accessed 11-April-2013].

[16] PCI-SIG. PCI Family History. http://www.pcisig.com/specifications/PCI_Family_History.pdf, 2006. [Online; accessed 03-April-2013].

[17] Jürgen Quade and Eva-Katharina Kunst. *Linux-Treiber entwickeln - Gerätetreiber für Kernel 2.6 systematisch eingeführt ; [eine systematische Einführung in Gerätetreiber für den Kernel 2.6].* DPUNKT VERLAG, Heidelberg, 3. aktualis. und erw. a. edition, 2011.

[18] Darmawan Salihun. Building a „Kernel“ in PCI Expansion ROM. https://sites.google.com/site/pinczakko/building-a-kernel-in-pci-expansion-rom. [Online; accessed 05-April-2013].

[19] G. Vinci, F. Barbon, R. Weigel, and A. Koelpin. A novel, wide angle, high resolution direction-of-arrival detector. In *Radar Conference (EuRAD), 2011 European*, pages 265–268, 2011.

[20] Gabor Vinci. *Six-Port based direction finding and ranging - An innovative approach for accurate direction finding and distance measurement.* PhD thesis, Friedrich-Alexander Universität Erlangen-Nürnberg, 2013.

# Index