Bachelorarbeit

Entwicklung eines Windows Kernel Treibers für PCIex/PXIe Datenerfassungskarten

zur Erlangung des akademischen Grades eines
Bachelor of Science

angefertigt von
Lukas Wagenlehner

Betreuer:

Erstprüfer	Prof. Dr. Ulrich Margull
Zweitprüfer	Prof. Dr. Franz Regensburger

ausgegeben am 06. Mai 2021
abgegeben am 15. August 2021

Erklärung

Hiermit erkläre ich, Lukas Wagenlehner, dass ich die vorliegende Bachelorarbeit bis auf die offizielle Betreuung selbstständig und ohne die Hilfe von Dritten verfasst habe. Die Arbeit wurde noch nicht für anderweitige Prüfungszwecke vorgelegt. Die verwendeten Quellen sowie die verwendeten Hilfsmittel sind vollständig angegeben. Wörtlich übernommene Textteile und übernommene Bilder und Zeichnungen sind in jedem Einzelfall kenntlich gemacht.

Ingolstadt, 15. August 2021

Lukas Wagenlehner

Danksagung

Peter Omlor für die großartige Zusammenarbeit und Leitung des Projektes. Ohne ihn wäre das Projekt nicht zustande gekommen.

Dem **ALLDAQ Team** für die Unterstützung im Bereich Materialbeschaffung und Beantworten der vielen Fragen.

Ulrich Margull für die hervorragende Betreuung der Arbeit von Seiten der Technischen Hochschule Ingolstadt.

Franz Regensburger für seine Arbeit als Zweitprüfer.

Meiner Familie für die Unterstützung während des gesamten Studiums und dem Korrekturlesen der Arbeit.

Abkürzungsverzeichnis

ADC	Analog-To-Digital Converter
API	Application Programming Interface
AXI	Advanced EXtensible Interface
BIN	Binary File
BIOS	Basic Input/Output System
BSoD	Blue Screen Of Death
CPCI	CompactPCI
DLL	Dynamic-Link Library
DMA	Direct Memory Access
DPC	Deferred Procedure Call
EEPROM	Electrically Erasable Programmable Read-Only Memory
EOL	End-Of-Life
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
GUID	Globally Unique Identifier
HAL	Hardware Abstraction Layer
I2C	Inter-Integrated Circuit
IDE	Integrated Development Enviroment
IOCTL	Input/Output Control
IP	Intellectual Property
IP	Internet Protocol
IRQL	Interrupt Request Level

ISR	Interrupt Service Routine
KMDF	Kernel-Mode Driver Framework
LAN	Local Area Network
NIC	Network Interface Controller
PCI	Peripheral Component Interconnect
PDB	Program Database
PnP	Plug And Play
PXI	PCI eXtensions for Instrumentation
SCSI	Small Computer System Interface
SDK	Software Development Kit
SPI	Serial Peripheral Interface
TTL	Transistor-Transistor Logic
UMDF	User-Mode Driver Framework
USB	Universal Serial Bus
WBEM	Web-Based Enterprise Management
WDF	Windows Driver Framework
WDK	Windows Driver Kit
WDM	Windows Driver Model
WMI	Windows Management Instrumentation
WPF	Windows Presentation Foundation
WPP	Windows Software Trace Preprocessor

Abbildungsverzeichnis

Inhaltsverzeichnis

1 Abstrakt

In der Bachelor Arbeit wird die Entwicklung eines Windows Kernel Treibers beschrieben. Der Treiber soll hierbei die ADQ-22 PXIe Karte ansteuern, welche von der Firma ALLDAQ entwickelt worden ist. Es handelt sich dabei um eine Digitale Ein- und Ausgabe Karte. Zur Implementierung des Treibers wird das WDF-Framework verwendet und die Programmiersprache C/C++. Zum Schluss wird noch auf das Debuggen und Tracing der entwickelten Software eingegangen.

2 Einleitung

Die Firma ALLDAQ wurde Anfang 2014 als Tochterfirma von ALLNET GmbH Computersysteme gegründet. Das „DAQ“ steht hier für „Data Acquisition“. Die Marke ALLDAQ steht für hochwertige, innovative PC-Messtechnik für den Einsatz in Labor und Industrie.
Sie bietet hochperformante Mess- und Steuerkarten aus eigener Entwicklung mit verschiedenen Schnittstellen, wie PCIexpress/PXI/PXIe und USB an. Die Kunden profitieren hier vor allem davon, dass sie Unterstützung und Support direkt von den Entwicklern der Technik bekommen. Zudem wird auf Kundenwunsch eigene Hardware entwickelt (ALLDAQ 2021c).
Im Jahr 2020 wurde die Entwicklung einer neuen Generation von Datenerfassungskarten für den PCIex/PXIe Bus beschlossen. Die ADQ-22 PXIe ist hierbei einer der ersten Prototypen.
In den weiteren Unterkapiteln der Einleitung soll die Notwendigkeit des Projektes erläutert werden. Zudem soll geklärt werden, welche Aufgaben ein Treiber hat. Des Weiteren wird noch das Windows Driver Model (WDM) von Microsoft vorgestellt und besprochen, was genau eine PCIex/PXIe Schnittstelle ist.

2.1 Problemstellung

Zu Beginn der Arbeit stellt sich die Frage, warum überhaupt die Entwicklung der ADQ-22 PXIe notwendig ist. Das wichtigste Argument ist der End-Of-Life (EOL) Zyklus des aktuell verbauten PCI 9056 Chips von dem Unternehmen Broadcom Inc. und einiger anderer Bauelemente.
Die Firma Broadcom entwickelt die meisten Chips für einen Zeitraum von 10-12 Jahren. Danach wird die Produktion eingestellt und meist durch eine neue Generation ersetzt. Bei beliebten Chips wird der Preis gegen Ende immer höher, bis er sich ungefähr verdoppelt hat. Nachdem ein Bauelement den EOL Status erreicht hat, kann er nur noch überteuert aus Restbeständen eingekauft werden. Der PCI 9056 erreicht sein EOL 2022, was die Entscheidung zur Neuentwicklung angestoßen hat (Broadcom 2021, vgl.).
Da eine Neuentwicklung viel an Zeit und Geld kostet, möchte ALLDAQ natürlich ihr Produkt auch gleichzeitig besser gestalten. Früher waren für die grundlegenden Funktionen der Karten zwei Chips notwendig. Der PCI 9056 für das PCI Interface und das Field Programmable Gate Array (FPGA) Spartan 6 von Xilinx für die Logikfunktion der Karten. Mit dem neu ausgewählten FPGA Artix 7, ebenfalls von Xilinx, lassen sich beide Funktionalitäten in einem Chip unterbringen. Diese Einsparung vereinfacht das Design auf der Platine.

Das bedeutet eine Kosteneinsparung bei der Produktion der Karte und dem Einkauf von Bauteilen. Ebenso ist der Footprint auf der Platine geringer, wodurch mehr Platz für das Frontend entsteht. Zuletzt besitzt der neue Artix 7 vor allem mehr Leistung. In der größten Ausführung hat der Spartan 7 102.400 Logik Zellen, mit 215.360 besitzt der Artix 7 mehr als doppelte so viele. Auch die anderen technischen Spezifikationen sind besser. (XILINX 2018).
Der Umstieg hat aber auch Nachteile. Neben dem erforderlichen Neudesign von Platine und Programmierung der Software, fehlt im FPGA die PCIex Schnittstellen Komponente. Technisch gesehen kann der Artix 7 die PCIex Schnittstelle zwar ansprechen, jedoch stellt die Firma XILINX Inc. den dafür notwendigen Firmware Part nicht zur Verfügung. Das Unternehmen Smart Logic GmbH bietet hier ihren "Multichannel DMA Flex IP Core for PCI-Express" an. Der IP Core stellt die Funktionen einer PCIex Schnittstelle incl. Direct Memory Access (DMA) zur Verfügung.
Zu guter Letzt kann eine modernere Architektur bei der Firmwareprogrammierung gewählt werden. Anstatt häufig aufkommende Komponenten selbst zu implementieren, bietet XILINX die Lösung in Form von Modulen an. Alle Module der Firmware werden dann untereinander im Chip mit einem Bus verbunden. Dadurch kann die Komplexität des Designs verringert werden und Fehler vermieden werden. Die verwendeten Module werden in Kapitel 3.2 näher erläutert.

2.2 Was ist ein Treiber

Bei einem Betriebssystem ist es unverzichtbar, Peripherie Geräte wie Maus, Tastatur, Grafikkarte, Monitor, etc. einzubinden. Da es viele Hersteller dieser Geräte gibt, die unterschiedlichste Features in ihre Hardware einbauen, ist es nicht möglich mit einer Software alle dieser Geräte zu steuern. Deshalb muss hierfür individuelle Software entwickelt werden, welche man als Treiber bezeichnet.
Der Autor Eduard Glatz definiert Treiber folgendermaßen: "Als Treiber (driver) bezeichnet man diejenigen Softwareteile, welche die eigentliche Verbindung zwischen den Anwenderprozessen und den Geräten bzw. deren Peripheriecontroller schaffen (Glatz 2019, S. 332)". Der Treiber kommuniziert also zwischen Anwendersoftware und Hardware.
Dies erreicht der Treiber, indem er Register der Hardware ausliest und beschreibt. So kann er den Zustand abfragen sowie Befehle und Daten übermitteln. Darüber hinaus kann mit Interrupts gearbeitet werden. Ein Interrupt unterbricht den aktuellen Programmablauf und erzwingt eine sofortige Abarbeitung. In Windows kann nur ein Kernel Treiber einen Interrupt bearbeiten,

da nur Kernel Treiber uneingeschränkten Zugriff auf alle Ressourcen haben (vgl. Glatz 2019, S.332f). Programme die im User Mode laufen, haben nur über Schnittstellen vom Betriebssystem und Treibern Zugriff auf die Hardware Ressourcen.
Für große und besonders schnelle Datenübertragungen steht dem Treiber außerdem noch DMA zur Verfügung. Nach der Initialisierung, kann hier die Peripherie direkt in den zugewiesenen Arbeitsspeicher schreiben. Andersherum kann der Computer auch direkt Daten auf das Gerät schreiben (vgl. Silberschatz, Galvin und Gagne 2019, S. 15).
Treiber sind im Windows Betriebssystem als Stack organisiert. Ganz unten im Stack sitzen die physischen Treiber, diese implementieren die Systemschnittstellen, beispielsweise den Standard für Peripheral Component Interconnect (PCI) oder Universal Serial Bus (USB). Die logischen Treiber, welche individuelle Hardware beschreiben, können dann auf den physischen Treibern aufbauen (vgl. Glatz 2019, S. 334). So kann man auch bei unterschiedlichen Mainboards z.B. auf die PCI Schnittstelle vom Mainboard Treiber eigene Software aufsetzen.
Anders als normale Programme, werden die meisten Treiber mit dem Systemstart geladen, da ohne sie keine Interaktion mit der Hardware möglich wäre. Das dynamische Laden und Entladen von Treibern ist grundsätzlich möglich, jedoch vom Betriebssystem und von der Art des Treibers abhängig. Damit zuletzt die Interaktion zwischen Treiber und Anwenderprogramm möglich ist, muss der Treiber sich und das Gerät gegenüber dem Betriebssystem definieren (vgl. Glatz 2019, S. 335). Das passiert unter Windows in der INF Datei oder Registry. Die INF Datei enthält wichtige Informationen zur Installation, wie Name von Treiber und Gerät, den Ort der Installation, Version und Registry Informationen (vgl. Hudek u. a. 2021b).

2.3 Windows Treiber Modell

Mit dem WDM lässt sich die generische Treiberschnittstelle von Windows darstellen. In der Abbildung 1 kann man sich einen bildlichen Überblick vom WDM machen. Ganz unten befindet sich der Hardware Abstraction Layer (HAL). Er stellt eine generische Schnittstelle zu der im System individuell verbauten Hardware dar. So kann jeder Treiber, der auf einen Bus zugreift, auf DMA Controller, Interrupt Controller, System Timer und Speicher Controller zugreifen (vgl. Stalling 2018, S. 101f).
In Windows wird zwischen drei verschiedenen Arten von Treibern unterschieden. Bus Treiber kommunizieren direkt mit der Hardware. Sie steuern die Schnittstellen und implementieren den jeweiligen Standard, wie PCI, Small Computer System Interface (SCSI) oder USB. Ihre Aufgabe ist es alle am

Bus angeschlossenen Geräte zu nummerieren sowie sich um das Plug And Play (PnP) und Power Management zu kümmern. Daneben, falls auf dem Bus unterstützt, multiplexen sie den Zugriff der einzelnen Komponenten auf dem Bus. Sie administrieren also die Bus Teilnehmer.
Die Funktionstreiber sitzen von den Bus Treibern aus gesehen eine Stufe höher. Diese Treiber implementieren die eigentlichen Peripherie Geräte. Der Treiber für die Datenerfassungskarte ADQ-22 PXIe dieser Arbeit ist ein solcher Funktionstreiber.
Zuletzt gibt es noch Filtertreiber. Sie sitzen zwischen Bus-, Funktionstreiber und Applikation. Wie der Name schon sagt, filtern sie die Kommunikation in der Treiberhierarchie und können diese auch manipulieren. Dafür gibt es ein großes Anwendungsgebiet, wie Monitoring, Verschlüsselung, Zugriffsberechtigung und so weiter. Filtertreiber können beliebig kaskadiert werden (vgl. Hudek, Viviano und Sherer 2021). In der Abbildung 1 sitzen die Treiber über dem HAL Layer.
Danach kommt die Kernkomponente von WDM, der I/O Manager oder auch Geräteverwaltung genannt. Er nimmt Ein-/Ausgabe Aufträge von den Applikationen entgegen und leitet sie an die zugehörigen Treiber weiter.
Die nächste Komponente im I/O Systems ist der PnP Manager. Dieser wird von den Bustreibern angesprochen, wenn für ein angeschlossenes Gerät kein Treiber existiert und der Anwender benachrichtigt werden muss. Der Power Manager steuert die verschiedenen Energie Modi von Windows. Wenn sich der Modus verändert, schickt er unterstützenden Treibern die entsprechenden Kommandos zur Aktivierung oder Abschaltung der angeschlossenen Hardware zwecks Energieeinsparung. Er findet vor allem bei Laptops, aber auch bei leistungsstarken Grafikkarten seinen Einsatz.
Die Windows Management Instrumentation (WMI) Module stellen die Implementierung für Web-Based Enterprise Management (WBEM) bereit. WBEM beschreibt der Autor Glatz als "[...] Basis für eine unternehmensweite Datensammlung und Datenverwaltung [...]"(Glatz 2019, S. 347).
Rechts in der Abbildung sieht man die schon angesprochenen INF Dateien und Registry. Dazu kommen noch die CAT Dateien, welche für verifizierte Software Komponenten eine Digitale Signatur von Windows bereitstellten.
Im User Mode befindet sich die Anwender Software, aber auch ein User Mode PnP Manager. Dieser Manager ist es, welcher vom Kernel äquivalent angesprochen wird, wenn kein Treiber für ein Gerät existiert (vgl. Glatz 2019, S. 348).
Für die vereinfache Treiberprogrammierung gibt es von Microsoft das Windows Driver Framework (WDF), welches auf dem Modell von WDM aufbaut. Grundlegend können hier zwei Arten von Treibern entwickelt werden, Kernel- und User- Mode Treiber.

User Mode Treiber können nur über einen speziellen Kernel-Filtertreiber indirekt im Windows Kernel Aufgaben erledigen. Sie haben keinen direkten Zugriff auf die Hardware und eignen sich daher nur für z.B. Netzwerktreiber oder USB Treiber für Kameras, Fingerabdrucksensor oder Headsets. Kernel Mode Treiber haben im Gegenzug uneingeschränkte Privilegien im Windows Kernel (vgl. Glatz 2019, S.348). Dadurch sind sie extrem mächtig. Sie können durch einen Fehler aber auch schnell zu einem Blue Screen Of Death (BSoD) führen oder sogar Daten und Betriebssystem modifizieren bzw. löschen.
WDF ist objektorientiert aufgebaut, es besteht aus den sogenannten WDF-Objekten. "WDF-Objekte besitzen Methoden (methods), Eigenschaften (properties) und Ereignisse (events)." (Glatz 2019, S. 349). Die WDF-Objekte sind in Wirklichkeit C++ Strukturen, mit denen als Funktionsparameter Funktionen aufgerufen werden. Die WDF-Objekte enthalten auch Callback Funktionspointer. Mit diesen Funktionspointern können bestimmte Events im Framework abonniert werden. Das einfachste Event hierbei ist das *EvtDriverDeviceAdd* Ereignis, es wird vom Framework ausgelöst, sobald ein neues Gerät für den Treiber gefunden worden ist.

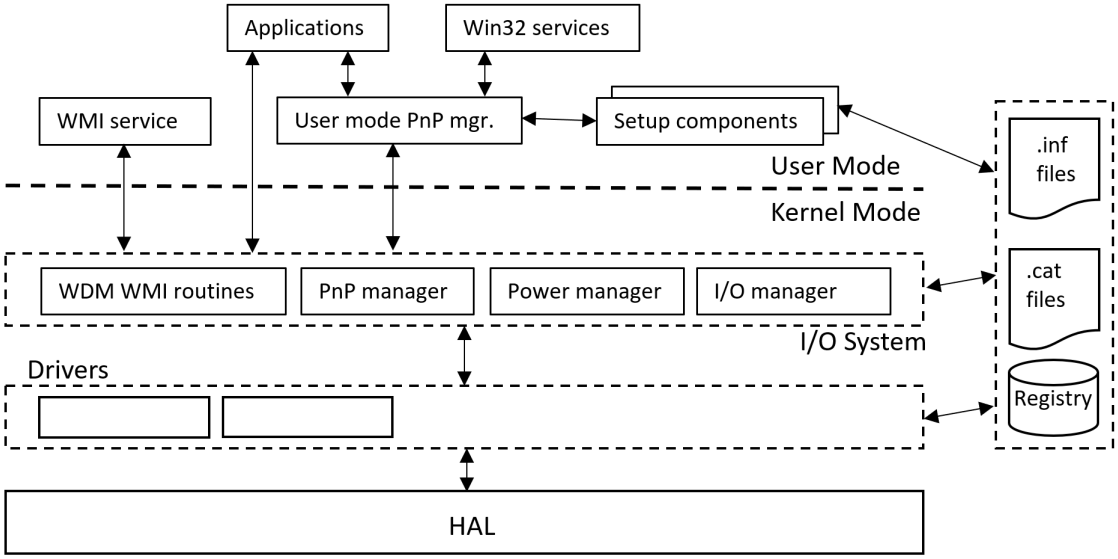


Abbildung 1: WDM Übersichtsbild (Glatz 2019, S. 348)

2.4 PCIex/PXIe Schnittstellen Technologie

Der PCI Bus ist ein "Paralleler Bus zur Ankopplung von Peripheriegeräten (Drucker, Scanner) an ein Computersystem." (Dembowski 2013, S. 4). Die

PCI Spezifikation wird von PCI-SIG (*https://pcisig.com/*) herausgegeben.
Da der PCI Bus bei einer Transferrate von maximal 132 MByte/s mit der Zeit zu langsam wurde, wurde der PCI Express (PCIe) Standard eingeführt. Obwohl vom PCIe Bus gesprochen wird, handelt es sich technisch gesehen jedoch um eine Punkt zu Punkt Verbindung. Deshalb sind die beiden Standards elektronisch auch nicht mehr kompatibel. Die Neuerungen beeinflussen die Software allerdings nicht, sodass ältere Programme weiterhin lauffähig bleiben.
PCIe arbeitet mit den sogenannten Link Verbindungen. Jede Link Verbindung ermöglicht eine eigene isolierte Verbindung von zwei Punkten. Dadurch können über mehrere Links gleichzeitig Datenpakete übertragen werden. Daher fallen die Zugriffszeiten auf den Bus weg. Ein Link verwendet eine oder mehrere Lanes. Jede Lane besitzt wiederum vier Signalleitungen, welche sich in zwei differentielle Leitungen für Hin- und Rückrichtung unterteilen. Auf einer einzelnen Leitung lassen sich die Daten auch nicht mehr parallel übertragen, sondern nur noch seriell. Die Lanes werden folgendermaßen gekennzeichnet: x1, x2, x4, x8, x12, x16 und x32. Die Zahl nach dem "x" gibt dabei die Anzahl der Lanes an.
Mit einer einzelnen Lane lässt sich so eine Datentransferrate von 250 MByte/s in PCIe-Version 1.X erreichen. Nach und nach wurden immer höhere Geschwindigkeiten und größere Datenmengen verlangt. Erreicht worden ist dies z.B. mit einer Erhöhung der Taktrate von PCIe-Version 1.X auf 2.X. In diesem konkreten Fall wurde der Takt verdoppelt, was eine Übertragungsgeschwindigkeit von 500 MByte/s ermöglichte (vgl. Dembowski 2013, S. 295ff). Die aktuelle PCIe-Version 5.0 kann bei x1 3.94 GByte/s und bei x16 63 GByte/s übermitteln (PCI-SIG 2021). Die angegebenen Datenraten sind nicht die reinen Nutzdaten, sondern beinhalten noch einen Overhead für das Protokoll.
Auf der Abbildung 2 sieht man ein Asus Prime Z370-P II Mainboard mit 6 PCIe Slots. Die zwei größeren Anschlüsse sind PCIe-x16 Slots und die vier kleineren sind PCIe-x1 Slots.

Der PCI eXtensions for Instrumentation (PXI) Standard ist speziell für die Automatisierungstechnik aus dem CompactPCI (CPCI) weiterentwickelt worden. PXI Karten werden auf einer Backplane verbaut, welche sich in einem Chassis befindet (siehe: Abbildung 5 & Abbildung 3). Durch die Backplane wird die Latenz extrem verringert. Die Auswirkung ist eine bis zu hundertfache Verbesserung der Performance zwischen mehreren kommunizierenden Karten auf dem Bus. Da auch PXI auf dem Standard PCI aufbaut, muss bei der Softwareentwicklung keine Unterscheidung gemacht werden (Alliance 2021, vgl.).

Der Konfigurationsbereich (Configuration Space) ist bei allen PCI Stan-

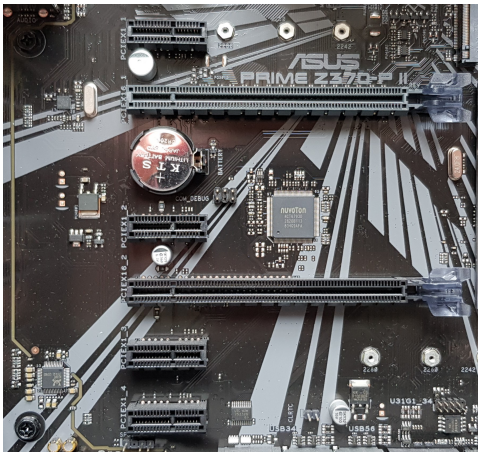


Abbildung 2: Mainboard mit PCIe Slots

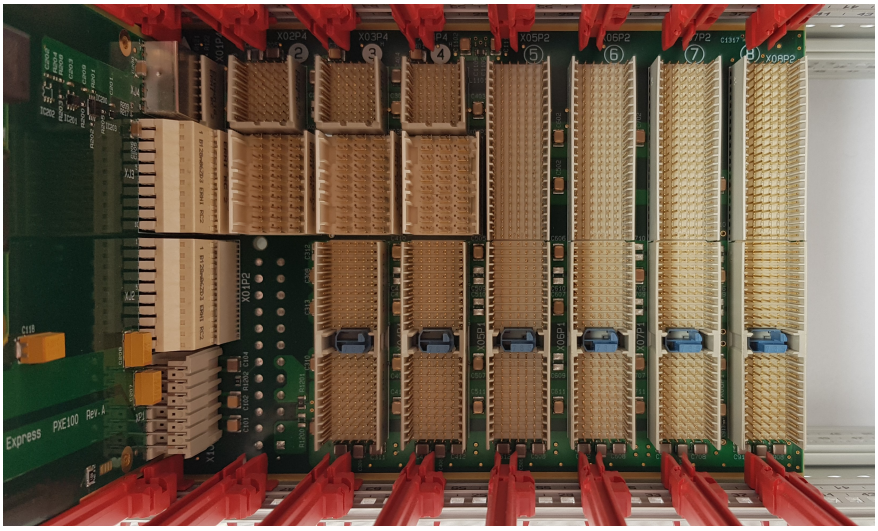


Abbildung 3: Backplane mit PXIe Slots

dards gleich. Bei den neueren Versionen wurde er jedoch abwärtskompatibel um neue Features erweitert. Beim Bootvorgang des Systems wird mit diesen Daten die PCI Einheit automatisch konfiguriert. Geladen werden die Daten meistens aus einem Electrically Erasable Programmable Read-Only Memory (EEPROM) oder FPGA, welche auf der PCI Karte verbaut sind.
Im PCI-Header sind die Device ID, Vendor ID, Status Register, Command Register, Class Code, Base Address Register 0-5, Interrupt Pin/Line und Vieles mehr enthalten. Die Device ID und Vendor ID geben den Hersteller und die Gerätekennung an. Das Status Register informiert über Fehler, Timings und Weiteres. Im Command Register wird das Verhalten der Karte

beschrieben z.B. ob sie ein Bus Master ist. Mittels dem Class Code Register kann dem Device eine Funktion, wie etwa Speicher Controller oder Netzwerk Controller zugeordnet werden. In den sechs Base Address Registern stehen die Pointer für die Basisadressen der I/O- und/oder Memory-Bereiche. Dadurch werden die sogenannten Spaces definiert. Durch die Spaces wird der Zugriff auf die Register im PCI Gerät ermöglicht. Interrupt Pin und Line geben an, über welchen Anschluss der PCI-Interrupt abgebildet wird und welcher PCI-Interrupt-Kanal verwendet wird.
Nach dem Header folgt der Device Space. Der Device Space ist geräteabhängig und kann vom Karten Hersteller frei definiert werden. (vgl. Dembowski 2013, S. 273-279)

3 Anforderungs Analyse

In diesem Kapitel soll darauf eingegangen werden, wie die Hardware technisch aufgebaut ist und was dadurch für Anforderungen an die Software gestellt werden.

3.1 Hardware Analyse

Die ADQ-22 PXIe lässt sich in 2 Teile, das Front- und Backend unterteilen. Das Backend (rechte Seite auf Abbildung 4) beinhaltet hierbei den Anteil an Chips und Bauteilen, der auf jeder neuen Datenerfassungskarte von ALLDAQ als Grundlage vorhanden sein muss. Durch die Unterteilung kann bei einer neuen Entwicklung einer PXIe Karte das Backend kopiert werden.
Im Backend ist unter anderem der Artix 7 XC7A100T-1FGG484C von Xilinx verwendet worden. Der Artix 7 ist ein FPGA. Bei einem FPGA handelt sich es um einen programmierbaren Logikbaustein, in den sich ohne Veränderung der Hardware neue Logik per Software einprogrammieren lässt (vgl. Dembowski 2013, S. 75). Das Programmieren eines FPGAs wird oft auch Flashen genannt.
Der FPGA ist der wichtigste Chip auf der Datenerfassungskarte, denn er enthält die komplette Logik der Karte. Darüber hinaus stellt er das nötige Interface zum PCIe Bus zur Verfügung. Zusätzlich sind ein Flash (AT25DF041A), EEPROM (S25FL128S) und der PXIe Stecker, sowie sämtliche Bauteile zur Anbindung oder Stromversorgung der genannten Chips im Backend enthalten.
Vom Frontend (linke Seite auf Abbildung 4) wird gesprochen, wenn es sich um den individuellen Anteil der Karte handelt. Bei der ADQ-22 PXIe sind es die 32 isolierten Digital-Eingänge (2*16 bit Ports) mit programmierbaren

Filtern und 32 isolierte Digital-Ausgänge (2*16 bit Ports), 16 bidirektionale TTL-Digital-I/Os (2*8 bit Port) und ein ADQ-LINK Module (vgl. ALLDAQ 2021b, S. 13).
ADQ-LINK ist hier ein Bezeichnung von ALLDAQ, im Grunde handelt es sich um einen I2C Bus für längere Leitungen. Erreicht wird dies mit dem Chip LTC4331, dieser wandelt den I2C Bus um und überträgt ihn differentiell über ein Twisted Pair Kabel. Um den Bus nun auf einer anderen Baugruppe zu verwenden, wird der gleiche Chip (LTC4331) auch auf der Gegenseite benötigt. Es handelt sich hierbei um eine Punkt zu Punkt Verbindung.
Um die ADQ-22 PXIe zu betreiben wird ein PCIeX4 Expand System benötigt, zu sehen in Abbildung 5. Dieses erlaubt es dem Anwender bis zu 17 Karten mit der maximalen Ausbaustufe in einem System zu betreiben. Es ist auch möglich durch Reihenschaltung weiterer solcher Systeme noch mehr Karten zu betreiben.
In Abbildung 6 sieht man die Brückenkarte, welche in den Host PC eingebaut wird. Mit dem abgebildetem PCIe X4 Verbindungskabel von Molex können dann Host- und Endsystem verbunden werden (ALLDAQ 2021a).

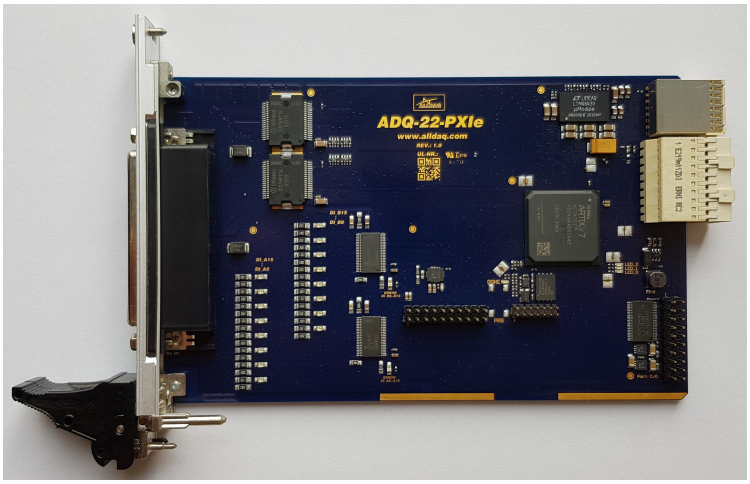


Abbildung 4: ADQ 22 PXIe

3.2 Anforderungen an die Software

Im Folgenden werden die Anforderungen von ALLDAQ auf die Software konkretisiert. Zu Beginn hierfür stellt sich die Frage, was für Software überhaupt benötigt wird. Windows 10 als grundlegendes Betriebssystem ist von ALLDAQ vorgegeben, da dies von allen Kunden eingesetzt wird.

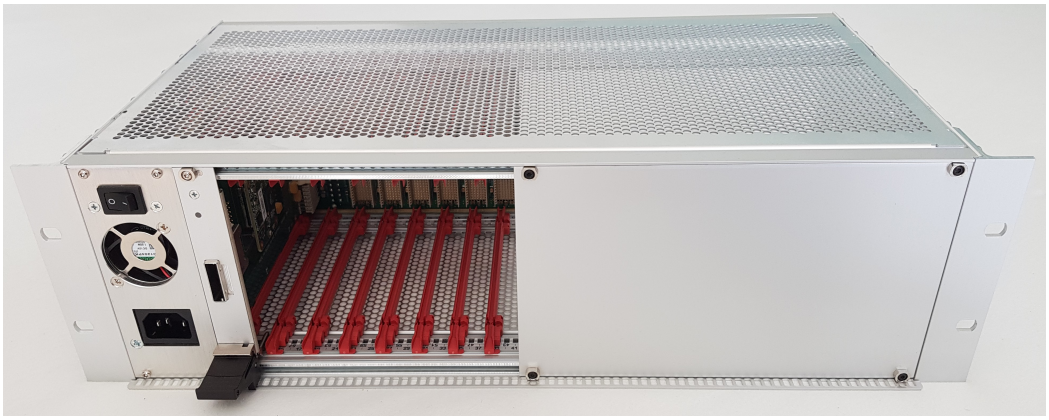


Abbildung 5: PCIe Expand System Hybrid

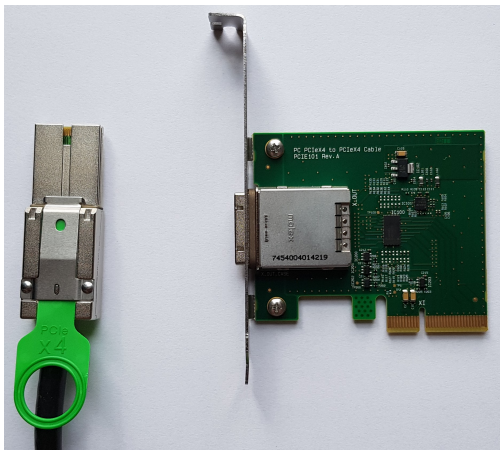


Abbildung 6: PCIe Verbindungskarte und Kabel

Eine normale Java Anwendung reicht hier trotzdem nicht aus, da solche Software nicht die nötigen Zugriffs Rechte für den PCI Bus besitzen kann. Um unter Windows Zugang zum PCI Bus zu erlangen, muss ein Treiber geschrieben werden. Windows bietet hier zwei Möglichkeiten an, User-Mode Driver Framework (UMDF) oder Kernel-Mode Driver Framework (KMDF). UMDF Treiber laufen im User Mode, welcher z.B. für File System Treiber oder einfache I/O Treiber geeignet ist (vgl. Hudek, Viviano und Sasouvanh 2021). Die andere Option ist ein KMDF Treiber, dieser läuft im Kernel Mode und hat unbeschränkte Privilegien. Dadurch hat er Zugriff auf Interrupt Signale und direkten Hardwarezugriff (vgl. Glatz 2019, S.349). Da direkter Zugriff auf den PCI Bus für das Projekt unumgänglich ist, wird ein KMDF Treiber entwickelt.
Der Treiber alleine ist jedoch nicht ausreichend, später müssen ja die Anwen-

der die ADQ-22 PXIe steuern können. Zu diesem Zweck muss eine Dynamic-Link Library (DLL) mit Application Programming Interface (API) oder ein User Mode Programm geschrieben werden. In dieser Arbeit wird die Entwicklung der DLL nicht näher besprochen, da hier kein wesentlicher Unterschied zu anderen DLLs besteht. Lediglich das Einbinden anderer Header ist nötig. Auf die Aufrufe in der User Mode Applikation wird in einem folgendem Kapitel genauer eingegangen. Die dort vorgestellten Aufrufe lassen sich dann ohne großen Aufwand in eine DLL übertragen.
Als Programmiersprache für den Treiber sind C/C++ hier die uneingeschränkten Favoriten, denn damit ist WDF entwickelt worden. Für die Anwenderprogramme ist die Wahl der Programmiersprache nicht so relevant, da diese lediglich mit dem Treiber kommunizieren müssen, dies unterstützen fast alle modernen Sprachen. Im Rahmen der Arbeit sind alle Programmauszüge in C/C++ geschrieben.
Im Weiteren wird analysiert und definiert was die Software eigentlich leisten bzw. steuern muss. Grundlegend soll der Treiber alle Module im FPGA steuern. Darüber hinaus soll dem Anwender eine Möglichkeit geboten, werden via eigener Software im User Mode die ADQ-22 PXIe zu bedienen.
Zu den Modulen im FPGA gehören ein Intellectual Property (IP) Core, GPIO-, XADC, 2*Quad-SPI-Modul und eine individuelle Registerstruktur mit dazugehöriger Logik für das Frontend. Die Module im FPGA lassen sich alle über die sechs zur Verfügung stehenden Spaces ansteuern. Dies geschieht, indem Werte in die Register über den zugehörigen Space und Offset geschrieben werden, so werden interne State Machines gestartet oder Funktionen ausgelöst.
Obendrein muss der Treiber alle auftretende Interrupts bearbeiten. Die ADQ-22 PXIe kann beispielsweise einen Interrupt auslösen, wenn einer der Eingangsports auf einen Bit-Change programmiert worden ist. Nicht nur das Frontend kann Interrupts auslösen, sondern auch die Module (I2C, SPI, ADC, Counter) im FPGA, etwa bei einem Fehler.
Zuletzt soll noch für spätere Entwicklungen ein DMA Handling in den Treiber eingebaut werden. Auch wenn das bei der aktuellen Karte noch nicht notwendig ist, wird es beispielsweise bei einer analogen Messkarte mit erheblich höheren Datendurchsatz gebraucht. Durch die Aufteilung in Front- und Backend macht eine solche vorgezogene Entwicklung durchaus Sinn. Bei zukünftigen Karten der ALLDAQ soll dasselbe Backend verwendet werden und große Teile des Treibers können übernommen werden.
WDM bietet auch ein Power Management für PCI Karten an. Dieses wird für ALLDAQ Karten jedoch nicht implementiert. Im Industriellen Einsatz werden die Karten permanent verwendet und nicht ausgeschaltet.

4 Entwicklung

In diesem Kapitel soll die Entwicklung der Software erläutert werden. Einige Punkte wie Debuggen und vor allem die Programmierung der User Mode Anwendung finden parallel statt. Die User Mode Anwendung ist bei der Entwicklung besonders hervorzuheben, denn ohne sie lässt sich der Kernel Treiber Code nur sehr eingeschränkt ausführen. Die Driverentry Funktion kann z.B. mit WinDbg getestet werden. Hintergrund ist hierbei, dass die Driverentry Funktion schon bei vorhandener Hardware und Neuinstallation des Treibers oder Neustarts des Systems ausgeführt wird. Input/Output Control (IOCTL) Aufrufe und Interrupts müssen gesondert behandeln werden. Diese können nur mit einer entsprechenden User Mode Applikation ausgeführt werden. Das direkte Testen bzw. Ausführen von neu geschriebenem Code ist jedoch essentiell, da so Fehler deutlich früher auffallen. Außerdem können Fehler in kürzeren Codeabschnitten schneller behoben werden.
Die im Folgenden verwendeten Funktionen sind alle von Microsoft auf *https://docs.microsoft.com//...//wdf* beschrieben.
Als Code Beispiel hat dabei das Projekt PLX9x5x auf *https://github.com/.../PLX9x5x* gedient. Es handelt sich hierbei um ein Unterprojekt aus einer großen Sammlung an Treiber Beispielen von Microsoft. In dem genanten Projekt wird ein PCI Treiber für den PLX9x5x Chip gezeigt.
Bei den gezeigten Code-Abbildungen handelt es sich hier nur um Ausschnitte. In ihnen wurden die meisten Kommentare, Fehler Behandlung/Überprüfung, Compiler Anweisungen, Variablen Definition, sowie Tracing entfernt. Die Code Ausschnitte sind nur als Orientierungshilfe gedacht und lassen sich in diesem Zustand nicht Compilieren.

4.1 Modellentwicklung für den Aufbau der Firmware

Eine Firmware ist ein Computer Programm, welches direkt auf einen Mikrocontroller oder Chip geflasht wird.
Bei der ADQ-22 PXIe wird die Firmware auf den FPGA geflasht. Beim ersten Mal muss dies mit externer Hardware erfolgen. In der Abbildung 4 sieht man unter dem FPGA eine 14 polige Steckerleiste, hier wird das Platform Cable USB Model DLC9LP von Xilinx zum Flashen angeschlossen. Nun kann mit der Xilinx Entwicklungsumgebung Vivado die Firmware als Binary File (BIN) direkt in den dazu vorgesehenen Flashspeicher geladen werden.
Beim Systemstart (power on) wird aus dem Flash dann die Firmware jedes Mal neu in den FPGA geschrieben. Da es jedoch nach Auslieferung der Karte an den Kunden neue Features oder Bug fixes in der Firmware geben kann,

muss diese änderbar sein. Für diesen Zweck muss der Flash vom Nutzer neu programmiert werden können. Die spätere Programmierung des Flash Chips ist deshalb auch über den geflashten FPGA möglich.

Zum Erstellen der Firmware wurde die Entwicklungsumgebung Vivado De-

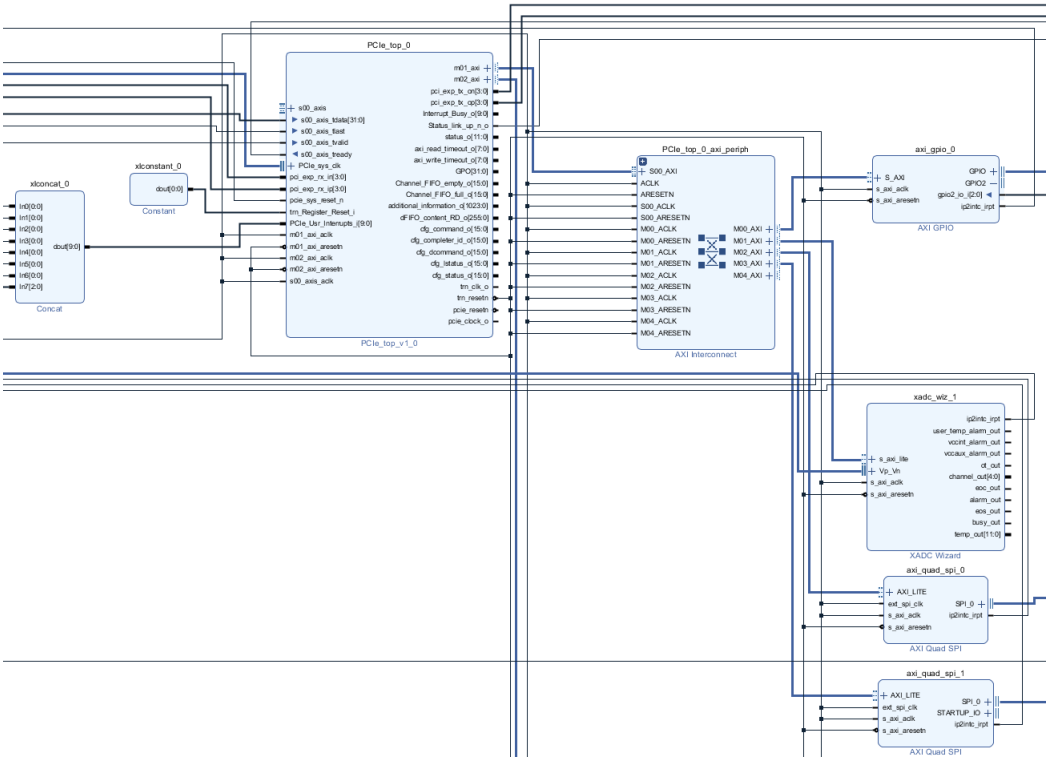


Abbildung 7: Firmware Blockschaltbild in Vivado

sign Suite verwendet. In dem Blockdiagramm aus Abbildung 7 sieht man die einzelnen Module der Firmware. Alle Module haben das Advanced EXtensible Interface (AXI) und sind somit über einen zentralen Bus im FPGA untereinander verbunden.

Der Multichannel DMA Flex IP Core for PCI-Express (kurz IP Core) von der Firma Smartlogic stellt das PCI-SIG kompatible Interface zur Verfügung. Des Weiteren stellt er acht AXI Master, DMA Handling, Interrupt Support und Einiges mehr für die Firmware Programmierung zur Verfügung. (vgl. Smartlogic 2020, S. 6). Der IP Core wird über den Space 0 angesprochen. Das GPIO-, XADC- und die zwei Quad-SPI-Module werden mit Space 1 angesprochen. Die von ALLDAQ entwickelte Logik der Datenerfassungskarte ist auf den Space 2 gelegt. Die anderen Spaces sind nicht in Verwendung. Grund für diese Aufteilung ist, dass bei neu entwickelten Karten Space 0 und 1 übernommen werden können. So müssen Treiber und Firmware bei

neuen Entwicklungen in diesem Bereich nicht verändert werden.
Das General Purpose Input/Output (GPIO)-Module steuert im aktuellen Prototypen LEDs und einige Ports auf den Steckleisten der ADQ-22 PXIe an. Es wird für den späteren Betrieb nicht mehr gebraucht. Während der Entwicklung dienten die LED's zum visuellen Testen interner digitaler Signale aus dem FPGA.
Mit Hilfe von dem XADC-Module lassen sich interne Chip Daten auslesen. Darunter fallen die Chip-Temperatur, Versorgung-Spannungen für den FPGA, Referenz Spannungen im Chip und externe analoge Input Signale. Die externen analogen Inputs können verwendet werden um diverse Spannungen im Frontend zu prüfen. Durch das Messen der unterschiedlichen Spannungen kann schnell festgestellt werden, ob die ADQ-22 PXIe einen elektrischen Fehler hat oder in Ordnung ist.
Die beiden Serial Peripheral Interface (SPI)-Module sind je für das EEPROM und für das Flash zuständig. Das EEPROM beinhaltet Daten, wie Seriennummer und Kalibriereinstellung. Die beiden SPI Bausteine werden im Master Mode betrieben. Sie schicken Kommandos und Daten an die beiden Speicherchips. Durch diesen Mechanismus lässt sich das Flash auch per Software mit neuer Firmware beschreiben. Das Laden der Firmware vom Flash in den FPGA wird über eigens dafür vorgesehene Leitungen gemacht.
Die von ALLDAQ entwickelte Logik im Space 2 besteht vor allem aus Registern und unterschiedlichen State Machines. Jede dieser State Machines ist für die Steuerung eines Chips im Frontend verantwortlich oder stellt eine Funktion der ADQ-22 PXIe dar. Die Entwicklung der Logik in diesem Bereich ist eine der Kompetenzen von ALLDAQ, die später über die Karte an den Kunden verkauft wird. Deswegen wird in dieser Arbeit nicht weiter im Detail darauf eingegangen. Die im Treiber hierfür verwendeten Befehle zur Steuerung der Register und des Interrupt Handlings sind hierbei dieselben, wie bei den bereits beschriebenen Modulen/Chips. Die entsprechenden Handbücher der verwendeten Chips und internen Modul in der Firmware liegen der Arbeit in digitaler Form bei.

4.2 Systemeinrichtung

Bevor mit der eigentlichen Softwareprogrammierung begonnen werden kann, müssen einige Programme installiert werden. Zuallererst wird ein Integrated Development Enviroment (IDE) benötigt. Hierfür kommt Visual Studio 2019 zum Einsatz. Über den Visual Studio Installer müssen noch einige Software Pakete zusätzlich heruntergeladen werden. Darunter sollte die neuste Version von "MSVC (v142-VS2019 C++-x64/x86 Buildtolls (v14.20))" sein, einem C++ Compiler von Microsoft. Zusätzlich sollte das ".Net-Destopentwicklungs

Paket" oder alternativ "Destopentwicklung mit C++ Paket" installiert werden. Empfehlenswert ist auch der Help Viewer, er ermöglicht es sämtliche Dokumentationen von Windows Funktionen, Definitionen, Macros, etc. nachzuschlagen.
Speziell für die Windows Treiber Entwicklung muss dann noch das Windows 10 Software Development Kit (SDK) (10.0.18362.0) Paket und das Windows Driver Kit (WDK) (Version 1803, 10.0.17134) installiert werden. Das Windows SDK Paket enthält Header, Bibliotheken, Metadaten und Tools zum Erstellen von Apps für Windows 10. Das WDK ist vor allem ein Toolset von Microsoft. Es enthält Dokumentationen, Beispiele, Tools zum Entwickeln und Debugging Programme für die Treiberentwicklung. Standardmäßig ist Git als Versions-Kontroll-System schon dabei, muss jedoch beim Projekt manuell aktiviert werden.
Nachdem erfolgreichen Installieren kann Visual Studio geöffnet werden. Bei der Projekt Erstellung kann nun das KMDF Template ausgewählt werden. Wichtig bei dem Projektnamen ist hier, dass dieser nur 32 Zeichen besitzen darf. Dieser Fakt ist in dem Header *wdfglobals.h* vorgeschrieben. Nach dem erfolgreichen Erstellen des Projektes hat man schon einmal eine gewisse Projektstruktur vorliegen. In dem Template existiert etwa Driver.c, in dieser Datei ist die Driverentry Funktion enthalten.
Auffallend ist, dass Dateien im Projekt mit sogenannten Filtern sortiert sind. So existiert ein Filter für Header Dateien, C Dateien und Treiber Dateien. In Wahrheit liegen alle Dateien jedoch im gleichen Ordner und werden nur in Visual Studio kategorisiert dargestellt. An dieser Stelle lässt sich auch nur empfehlen, mit dem System der IDE zu arbeiten und nicht dagegen. Es ist durchaus möglich Dateien in Ordnern anders zu organisieren, jedoch müssen alle diese Änderungen sorgfältig in den Projekteigenschaften konfiguriert werden. Das kann sehr schnell zu ungewollten Komplikationen führen.
Standardmäßig ist beim Erstellen dieser Arbeit noch C++ 14 eingestellt. Das kann jedoch in den Eigenschaften auf C++ 17 umgestellt werden.
Im Filter *Driver Files* befindet sich die INF Datei. Die INF Datei muss zuallererst ausgefüllt werden. Sie enthält Setup Informationen für die spätere Treiberinstallation sowie grundlegende Informationen über den Treiber. Begonnen wird mit der ClassGuid. Ein Globally Unique Identifier (GUID) kann direkt in Visual Studio generiert werden (Tools > Create GUID).
Die GUID gibt dem Treiber die Möglichkeit sich global im Windows System eindeutig zu identifizieren. Die GUID wird später auch von der User Mode Software verwendet um ein Handle für die Kommunikation mit dem Treiber zu öffnen.
Mit dem *Stampinf* Tool, welches über die Projekt Eigenschaftsseite konfiguriert werden kann, werden Felder wie Datum und Version automatisch

verwaltet.
Als nächstes muss dem Treiber eine *Class* zugeordnet werden. Dies ist später der *Class*-Name, unter dem die einzelnen Devices im Device Manager von Windows aufgelistet werden.
Der wichtigste Teil in der INF Datei ist die Zuordnung von Device- und Vendor-ID der Karten. In der Abbildung 8 kann man Visual Studio mit geöffneter INF Datei sehen. Unter *https://pcisig.com/membership/* können die einzelnen Vendor-IDs nachgeschlagen werden.
Wenn eine Firma PCI Karten entwickeln möchte muss eine solche ID gekauft werden. Im Falle von ALLDAQ ist das die Vendor-ID 0x1B49. Die Device-ID wird im Unternehmen zur Unterscheidung unterschiedlicher PCI Karten frei vergeben. Die ADQ-22 PXIe besitzt die ID 0x7222. Es können auch mehrere IDs an dieser Stelle eingetragen werden, solange die dazugehörigen Karten von dem Treiber unterstützt werden. Zuletzt können noch einige Felder wie z.B. Herstellername ausgefüllt werden. Diese Daten lassen sich später über den Windows Device Manager auslesen.
Nach der Einrichtung wird ein erster Test Build gemacht. Der entstandene Treiber kann dann sogleich installiert werden und sollte bei eingebauter Hardware im Windows Device Manager angezeigt werden.
Zum Testen der programmierten Treiberfunktionen wird außerdem noch ein User Mode Projekt als Client erstellt. Hier gibt es nun mehrere Optionen.
Ein C++ Konsolen Programm ist die einfachste und direkteste Lösung zur Kommunikation mit dem Treiber.
Ein C++ DLL Projekt ist flexibler und kann später auch vom Kunden eingesetzt werden. Ergänzend dazu wird hier jedoch noch eine Graphical User Interface (GUI) Applikation zur Bedienung benötigt. Die GUI kann dann in beliebiger Sprache und Framework geschrieben werden. Bei ALLDAQ wird hier Windows Presentation Foundation (WPF) als Framework verwendet, welches mit C# programmiert wird.
Beide Lösungen wurden für das Projekt umgesetzt.

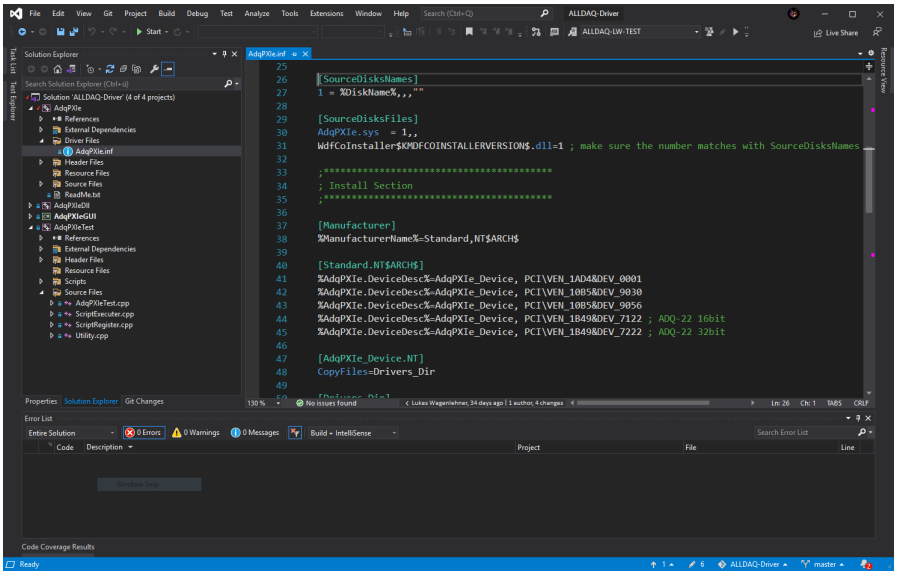


Abbildung 8: Entwicklungsumgebung Visual Studio

4.3 Driverentry und EvtDeviceAdd

Die *Driverentry* Funktion ist die *main* Funktion eines Treibers. Aufgerufen wird sie nach dem Laden des Treibers von WDF.
In Abbildung 9 sieht man eine mögliche Implementierung. Der Parameter *DriverObject* repräsentiert den Treiber als WDF-Objekt. Das Objekt enthält den Treibernamen, Startadresse, Größe, Type, etc.. Mit dem zweiten Parameter *RegistryPath* erhält der Treiber Zugriff auf die Registry zum Laden von Konfigurationsparametern.
Anmerkung zur Formatierung der Parameternamen: WDF verwendet die Ungarische Notation. So steht z.B. *P* für einen Pointer und *H* für ein Handle. Das oft nachfolgende kürzel *Evt* steht für Event.

Die *Driverentry* Funktion muss nun den Treiber initialisieren. Im Codeauszug wird hier etwa das Event *AdqPXIeEvtDeviceAdd* für den PnP Manager registriert. Wenn Speicher allokiert werden soll, muss zusätzlich auch eine *EvtDriverContextCleanup* Routine registriert werden. In *EvtDriverContextCleanup* muss dann der Speicher wieder freigegeben werden. Jeder nicht freigegebene Speicherblock, nicht geschlossenes Handle, etc. kann ansonsten nur beim Neustart von Windows gelöscht oder geschlossen werden.
Sobald der PnP Manager nun ein passendes Gerät erkannt hat, wird *AdqPXIeEvtDeviceAdd* vom Framework aufgerufen. Hier können Device spezifische Initialisierungen getätigt werden. Im Folgenden werden die in späteren

Kapiteln angesprochenen Funktionen oder/und Events beim WDF Framework angemeldet.
Darunter fällt unter anderem die *DeviceContext* Struktur (Abbildung 10). Mit dem Macro *WDF_DECLARE_CONTEXT_TYPE_WITH_NAME* wird darauf eine Zugriffsfunktion *DeviceGetContext* definiert. Über diese kann überall im Treiber auf die individuellen Daten des aktuellen Devices zugegriffen werden. Die Initialisierung der Werte, wie sie im Beispiel zu sehen sind, findet aber erst im nächsten Kapitel statt.
Ähnlich wie zuvor werden jetzt auch die beiden Funktionen für *PrepareHardware* und *ReleaseHardware* beim PnP Manager registriert. Da sie für jedes Gerät neu aufgerufen werden, ist es nicht möglich ihre Anmeldung vorzuziehen.
Die im Kapitel 4.5 verwendete *DeviceIoQueue* wird als nächstes erstellt. Zur Konfiguration wird die *WDF_IO_QUEUE_CONFIG* Struktur verwendet. Damit die Queue IOCTL Codes bekommt, muss der Funktionspointer *EvtIoDeviceControl* in der Struktur belegt werden. Danach kann mit *WdfIoQueueCreate* die Queue erstellt werden.
Zuletzt werden noch die Interrupt Events im Framework angemeldet. Diese sind *InterruptEnable*, *InterruptDisable*, *InterruptIsr* und *InterruptDpc*. Mit *WdfInterruptCreate* kann dann die Interrupt Einrichtung abgeschlossen werden.
Damit es einer User Applikation möglich ist mit dem Device zu sprechen, muss die GUID aus dem Setup noch eingebunden werden. Das erfolgt durch die Erstellung eines Device Interfaces mit dem Aufruf *WdfDeviceCreateDeviceInterface*.
An diesem Punkt lässt sich der Treiber kompilieren. Wird er installiert, werden die angeschlossenen PXIe Karten im Windows Device Manager angezeigt.

```
NTSTATUS DriverEntry(
    _In_ PDRIVER_OBJECT  DriverObject,
    _In_ PUNICODE_STRING RegistryPath
)
{

    WDF_DRIVER_CONFIG config;
    NTSTATUS status;
    WDF_OBJECT_ATTRIBUTES attributes;

    WDF_OBJECT_ATTRIBUTES_INIT(&attributes);
    WDF_DRIVER_CONFIG_INIT(&config, AdqPXIeEvtDeviceAdd);
    status = WdfDriverCreate(DriverObject, RegistryPath, &attributes, &config, WDF_NO_HANDLE);

    return status;
}
```

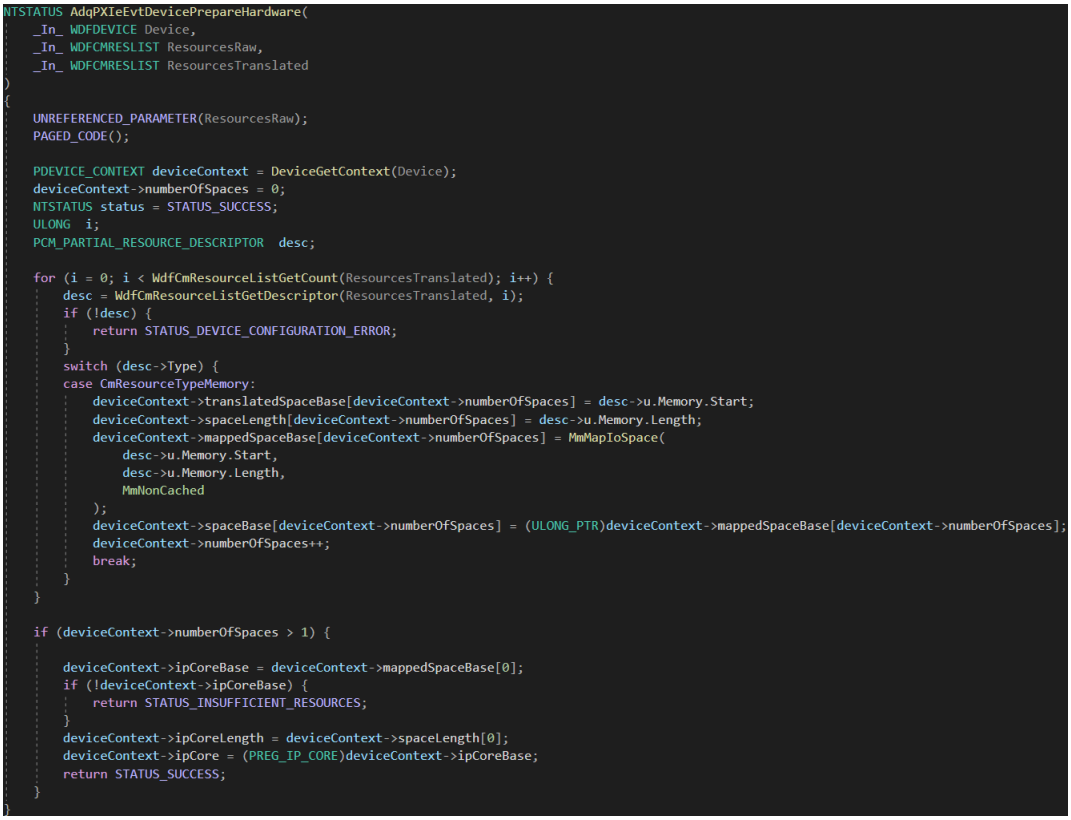
Abbildung 9: Driver Entry Funktion

```
typedef struct _DEVICE_CONTEXT
{
    UINT32              numberOfSpaces;             // Number of Spaces
    PHYSICAL_ADDRESS    translatedSpaceBase[6];     // Space base address, system address
    PVOID               mappedSpaceBase[6];         // Space base, mapped address
    ULONG               spaceLength[6];             // Space length
    ULONG_PTR           spaceBase[6];               // Space base

    USHORT              vendorID;                   // vendor ID
    USHORT              deviceID;                   // device ID

    // HW Resources
    PREG_IP_CORE        ipCore;                     // registers address of ip core
    PUCHAR              ipCoreBase;                 // registers base address
    ULONG               ipCoreLength;               // registers base length
} DEVICE_CONTEXT, *PDEVICE_CONTEXT;

WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(DEVICE_CONTEXT, DeviceGetContext)
```

Abbildung 10: Device Context Declaration

4.4 Prepare- und Release- Hardware

In dem *AdqPXIeEvtDevicePrepareHardware* Event wird die PXIe-22 für den Treiber eingerichtet. Der wichtigste Punkt beim Einrichten ist das Zuordnen der Speicherblöcke (Spaces). Erst danach kann in den Speicher der Karte geschrieben bzw. gelesen werden.
Erstmals ist es dann auch möglich Chips auf der Karte anzusteuern. Bei ALLDAQ werden hier z.B die IP Core Version, EEPROM ID und Flash ID überprüft. Wenn eine falsche Version oder eine falsche Bestückung festgestellt wird, beendet der Treiber hier mit einem Fehlercode. Die Verwendung der Karte ist dann nicht mehr möglich.
In der Abbildung 11 ist das “Memory mapping” in der *AdqPXIeEvtDevi-*



```
NTSTATUS AdqPXIeEvtDevicePrepareHardware(
    _In_ WDFDEVICE Device,
    _In_ WDFCMRESLIST ResourcesRaw,
    _In_ WDFCMRESLIST ResourcesTranslated
)
{
    UNREFERENCED_PARAMETER(ResourcesRaw);
    PAGED_CODE();

    PDEVICE_CONTEXT deviceContext = DeviceGetContext(Device);
    deviceContext->numberOfSpaces = 0;
    NTSTATUS status = STATUS_SUCCESS;
    ULONG  i;
    PCM_PARTIAL_RESOURCE_DESCRIPTOR  desc;

    for (i = 0; i < WdfCmResourceListGetCount(ResourcesTranslated); i++) {
        desc = WdfCmResourceListGetDescriptor(ResourcesTranslated, i);
        if (!desc) {
            return STATUS_DEVICE_CONFIGURATION_ERROR;
        }
        switch (desc->Type) {
        case CmResourceTypeMemory:
            deviceContext->translatedSpaceBase[deviceContext->numberOfSpaces] = desc->u.Memory.Start;
            deviceContext->spaceLength[deviceContext->numberOfSpaces] = desc->u.Memory.Length;
            deviceContext->mappedSpaceBase[deviceContext->numberOfSpaces] = MmMapIoSpace(
                desc->u.Memory.Start,
                desc->u.Memory.Length,
                MmNonCached
            );
            deviceContext->spaceBase[deviceContext->numberOfSpaces] = (ULONG_PTR)deviceContext->mappedSpaceBase[deviceContext->numberOfSpaces];
            deviceContext->numberOfSpaces++;
            break;
        }
    }

    if (deviceContext->numberOfSpaces > 1) {

        deviceContext->ipCoreBase = deviceContext->mappedSpaceBase[0];
        if (!deviceContext->ipCoreBase) {
            return STATUS_INSUFFICIENT_RESOURCES;
        }
        deviceContext->ipCoreLength = deviceContext->spaceLength[0];
        deviceContext->ipCore = (PREG_IP_CORE)deviceContext->ipCoreBase;
        return STATUS_SUCCESS;
    }
}
```

Abbildung 11: Prepare Hardware Funktion

cePrepareHardware zu sehen. Dem *PrepareHardware* Event wird eine Liste von physischen Ressourcen (*ResourcesTranslated*) als Parameter mitgegeben. Durch diese Liste wird im gezeigten Programmausschnitt iteriert.
Wenn es sich bei dem *Type* der Ressource um *CmResourceTypeMemory* handelt, wird dieser Space weiterverarbeitet.
Ein Space ist eine Memory Ressource, er besteht aus Registern die beschrieben und gelesen werden können. Andere Typen sind beispielsweise Ports oder DMA Kanäle.
Die Startadresse und die Speichergröße werden jetzt im *deviceContext* gespeichert. Die gespeicherten Werte werden später für das Unmapping in *ReleaseHardware* benötigt.
Mit *MmMapIoSpace* wird die physische Adresse auf eine virtuelle Adresse gemappt. Bei dem Mapping muss unbedingt *MmNonCached* verwendet werden. Hierdurch wird Cashing vom Prozessor verhindert. Falls Cashing aktiv, wäre könnte es vorkommen, dass zwei aufeinanderfolgende Lesebefehle je den selben Wert liefern. Der Prozessor kann nicht wissen, ob sich ein Wert auf

der ADQ-22 PXIe geändert hat ohne diesen frisch aus der Karte auszulesen. Die Pointer auf die virtuellen Speicherblöcke werden ebenfalls im *deviceContext* abgespeichert.
Im Anschluss muss überprüft werden, ob die Speicherblöcke die erwartete Größe besitzen und ob die richtige Anzahl gefunden worden ist (nicht im Codeausschnitt enthalten).
In der Abbildung 12 sieht man die Definition der ersten vier Register im IP

```
typedef struct _IP_CORE_INTTERRUPT {
    unsigned int dma_write                  : 1;    // bit 0
    unsigned int dma_read                   : 1;    // bit 1
    unsigned int all_eofs_seen              : 1;    // bit 2
    unsigned int line_interrupt             : 1;    // bit 3
    unsigned int parameter_change_interrupt : 1;    // bit 4
    unsigned int reserved_1                 : 15;   // bit 5-19
    unsigned int test_interrupt             : 1;    // bit 20
    unsigned int reserved_2                 : 1;    // bit 21
    unsigned int user_interrupt             : 10;   // bit 22-31
} IP_CORE_INTERRUPT;

typedef struct _IP_CORE_INTERRUPT_CONTROL_2 {
    unsigned int invoke_test_interrupt          : 1;    // bit 0, 1: invokes a test interrupt
    unsigned int suspend_interrupt_transmission : 1;    // bit 1
    unsigned int reserved                       : 30;   // bit 2-31
} IP_CORE_INTERRUPT_CONTROL_2;

typedef struct _REG_IP_CORE {
    unsigned int versionRegister;                               // 0x0
    volatile IP_CORE_INTERRUPT interrupt_control;               // 0x4
    volatile IP_CORE_INTERRUPT interrupt_status;                // 0x8
    volatile IP_CORE_INTERRUPT_CONTROL_2 interrupt_control_2;   // 0xC
} REG_IP_CORE, *PREG_IP_CORE;
```

Abbildung 12: Register Declaration von IP Core

Core. Damit später leserlicher Code entsteht, wird der Strukturpointer *ipCore* auf die virtuelle Startadresse gesetzt. Soll z.B. das *versionRegister* aus *REG_IP_CORE* ausgelesen werden, lässt sich folgende Code Zeile schreiben:

ULONG version = READ_REGISTER_ULONG((PULONG)&deviceContext->ipCore->versionRegister)

Äquivalent gibt es zum Lesen der Register auch den Schreib Aufruf:

*void WRITE_REGISTER_ULONG(volatile ULONG *Register, ULONG Value)*

Beim Entladen des Treibers wird das *AdqPXIeEvtDeviceReleaseHardware* Event aufgerufen. Alles was in *PrepareHardware* gemacht worden ist, muss

hier wieder rückgängig gemacht werden. Das Mapping der Spaces wird mit *MmUnmapIoSpace* bereinigt. Falls der Treiber unerwartet entladen wird, sollten auch alle laufenden Aktionen auf der Karte gestoppt werden. Realisiert wird dies mit einem Reset oder Stop Befehl. Der verwendete Befehl muss dazu natürlich im FPGA der Karte vorgesehen sein.

4.5 DeviceIoQueue und IOCTL Codes

Ein IOCTL Code ist ein Befehl, welcher von einem Anwenderprogramm oder einem Treiber verschickt wird. Der IOCTL Code wird dann mit seinen dazugehörigen Datenpointern in eine WDF-Queue gespeichert. WDF löst für jedes Element in der Queue nach und nach das *AdqPXIeEvtIoDeviceControl* Event aus. In diesem Event wird der IOCTL Code verarbeitet. Anschließend wird dem Sender eine Rückmeldung gegeben.
In Abbildung 14 ist der Quellcode für die Verarbeitung von zwei unterschiedlichen IOCTL Codes beschrieben.
Zuerst müssen aber erst einmal die entsprechenden Codes definiert werden. In Abbildung 13 werden beispielhaft zwei IOCTL Codes definiert. Die Codes befinden sich in *public.h*. Dieser Header wird sowohl im Kernel, als auch in die User Applikation, eingebunden.
Der IOCTL Code ist ein 32 Bit Wert. Darum wird zur Erstellung das Macro *CTL_CODE(DeviceType, Function, Method, Access)* verwendet.
Der *DeviceType* darf ab 0x8000 frei gewählt werden. Werte darunter sind von Microsoft reserviert.
Die *Function* gibt an welcher Befehl später ausgeführt werden soll. Alle Werte unter 0x800 sind reserviert.
Der Parameter *Method* sagt aus, wie und ob Speicherpuffer bei dem Code verwendet werden. Im Beispiel wird einmal *METHOD_NEITHER* (keine Daten) und *METHOD_BUFFERED* (Puffer für kleine Datenmenge) verwendet. Zuletzt werden mit *Access* Lese- und/oder Schreibrechte auf den Speicherpuffer gegeben.
Wenn WDF nun das *AdqPXIeEvtIoDeviceControl* Event aufruft, beginnt

```
#define IOCTL_ADQPXIE_TEST CTL_CODE(0x8000,0x800,METHOD_NEITHER,FILE_ANY_ACCESS)
#define IOCTL_ADQPXIE_VENDOR_DEVICE CTL_CODE(0x8000, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

Abbildung 13: IOCTL Code Definition

die Bearbeitung des IOCTL Codes. Mit einem Switch Statement und dem *IoControlCode* wird zum passenden Codesegment gesprungen. Bei *IOCTL_ADQPXIE_TEST* muss lediglich der Request mit *WdfRequestComplete* abgeschlossen werden.

Standardmäßig wird allerdings anders vorgegangen. Zuerst werden alle Parameter auf Plausibilität, wie z.B. die Puffergröße geprüft. Das *ASSERT* Macro ist nur in der Entwicklungsphase aktiv. Später wird es mit der Kommando Option */DNDEBUG* abgestellt.
Im zweiten Schritt wird der Puffer aus dem *WDFMEMORY* auf eine in *public.h* deklarierte Struktur umgesetzt. Die meisten IOCTL Calls verwenden Strukturen aus *public.h* zum Datenaustausch. Im Fall von
IOCTL_ADQPXIE_VENDOR_DEVICE werden jetzt die IDs in den dafür vorgesehenen Speicher geschrieben.
Bei anderen IOCTL Codes wird z.B. ein Register auf der ADQ-22 PXIe gelesen oder beschrieben. Die Pointer für den Aufruf von etwa *READ_REGISTER* sind in *deviceContext* zu finden.
Zum Flashen der Firmware wurde ein eigenes Kommando implementiert. Die Firmware wird indirekt an den Treiber übermittelt. Mithilfe einer *UNICODE_STRING* Struktur wird lediglich der Speicherort übergeben.
Damit der Sendeprozess seine Antwort erhält und weiterlaufen kann, muss am Schluss noch der Request beendet werden.
Bei den unterschiedlichen *WdfRequestComplete* Funktionen sollte auch immer im Fehlerfall ein aussagekräftiger Fehlercode übergeben werden. Auf *https://docs.microsoft.com/.../NTSTATUS_Values* kann eine Übersicht über die etlichen Fehlercodes und deren Bedeutung erlangt werden.

```
VOID AdqPXIeEvtIoDeviceControl(
    _In_ WDFQUEUE Queue,
    _In_ WDFREQUEST Request,
    _In_ size_t OutputBufferLength,
    _In_ size_t InputBufferLength,
    _In_ ULONG IoControlCode
)
{
    NTSTATUS status = STATUS_SUCCESS;
    WDFMEMORY memory;
    WDFDEVICE device;
    PDEVICE_CONTEXT deviceContext;
    switch (IoControlCode)
    {
    case IOCTL_ADQPXIE_TEST:
    {
        WdfRequestComplete(Request, STATUS_SUCCESS);
        break;
    }
    case IOCTL_ADQPXIE_VENDOR_DEVICE:
    {
        ASSERT((IoControlCode & 0x3) == METHOD_BUFFERED);
        if (OutputBufferLength != sizeof(VendorDeviceID)) {
            WdfRequestComplete(Request, STATUS_INVALID_DEVICE_REQUEST);
            break;
        }
        status = WdfRequestRetrieveOutputMemory(Request, &memory);
        if (!NT_SUCCESS(status)) {
            WdfRequestComplete(Request, status);
            break;
        }
        VendorDeviceID* id = (VendorDeviceID*)WdfMemoryGetBuffer(memory, NULL);
        device = WdfIoQueueGetDevice(Queue);
        deviceContext = DeviceGetContext(device);
        id->deviceID = deviceContext->deviceID;
        id->vendorID = deviceContext->vendorID;
        WdfRequestCompleteWithInformation(Request, STATUS_SUCCESS, sizeof(VendorDeviceID));
        break;
    }
    default:
        WdfRequestComplete(Request, STATUS_SUCCESS);
        break;
    }
    return;
}
```

Abbildung 14: Device IO Control Funktion

4.6 User Mode Applikation

Damit eine User Mode Applikation mit einem Device kommunizieren kann braucht sie ein Handle. Mit dem Handle und der *DeviceIoControl* Funktion können dann Befehle an das Device geschickt werden.
Nach dem Erstellen des Projektes ist es notwendig die *puplic.h* als externe Ressource hinzuzufügen.

Danach muss für jedes Device des gewünschten Treibers ein Handle geöffnet werden. In Abbildung 15 ist der Programmcode zum Öffnen der Handles zu sehen.

Mit der *SetupDiGetClassDevs* Funktion wird ein Handle auf ein *Device In-*

```
int openHandle() {
    SP_DEVICE_INTERFACE_DATA deviceInterfaceData;
    ULONG predictedLength = 0;
    ULONG requiredLength = 0;
    DWORD deviceInterfaceIndex = 0;
    _numberOfDevices = 0;
    deviceInfos = NULL;
    // 1. SetupDiGetClassDevs() flag DIGCF_DEVICEINTERFACE -> list of HDEVINFO
    hardwareDeviceInfo = SetupDiGetClassDevs((LPGUID)&GUID_DEVINTERFACE_AdqPXIe, NULL, NULL, (DIGCF_DEVICEINTERFACE | DIGCF_PRESENT));
    // 2. SetupDiEnumDeviceInterfaces()
    while (true) {
        deviceInterfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);
        BOOL error = SetupDiEnumDeviceInterfaces(hardwareDeviceInfo, 0, (LPGUID)&GUID_DEVINTERFACE_AdqPXIe, deviceInterfaceIndex, &deviceInterfaceData);
        if (GetLastError() == ERROR_NO_MORE_ITEMS) {
            break;
        }
        if (!error) {
            SetupDiDestroyDeviceInfoList(hardwareDeviceInfo);
            return EXIT_FAILURE;
        }
        else {
            deviceInterfaceIndex++;
            _numberOfDevices++;
        }
    }
    // 3. SetupDiGetDeviceInterfaceDetail() -> device path
    deviceInfos = new DeviceInfo[_numberOfDevices];
    for (DWORD i = 0; i < _numberOfDevices; i++) {
        BOOL error = SetupDiEnumDeviceInterfaces(hardwareDeviceInfo, 0, (LPGUID)&GUID_DEVINTERFACE_AdqPXIe, i, &deviceInterfaceData);
        if (error) {
            SetupDiGetDeviceInterfaceDetail(hardwareDeviceInfo, &deviceInterfaceData, NULL, 0, &requiredLength, NULL);
            predictedLength = requiredLength;
            deviceInfos[i].deviceInterfaceData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, predictedLength);
            deviceInfos[i].deviceInterfaceData->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);

            error = SetupDiGetDeviceInterfaceDetail(hardwareDeviceInfo, &deviceInterfaceData, deviceInfos[i].deviceInterfaceData, predictedLength, &requiredLength, NULL);

            deviceInfos[i].hDevice = CreateFile(deviceInfos[i].deviceInterfaceData->DevicePath, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, NULL);
        }
        else {
            return EXIT_FAILURE;
        }
    }
    return EXIT_SUCCESS;
}
```

Abbildung 15: Open Handle Funktion

formation Set erhalten. Diese und folgende Funktionen haben den Präfix *Setup*, weil sie im *setupapi.h* Header zu finden sind. Das Set enthält dabei alle installierten Devices, auf welche die Parameter zutreffen. Mit der GUID aus dem Setup kann sichergestellt werden, dass nur ADQ-22 PXIe Karten im Set enthalten sind.

Im Anschluss wird im zweiten Schritt die Anzahl der Devices Ermittelt. Dafür muss mit *SetupDiEnumDeviceInterfaces* einmal durch das Komplette Set iteriert werden. Der Zähler *numberOfDevices* hält dabei die Anzahl der Devices fest. Nach Beendigung von Schritt Zwei, wird ein Array allokiert. Dieses enthält später unter anderem die Device Handles. In der nächsten Schleife wird die *SP_DEVICE_INTERFACE_DETAIL_DATA* Struktur für jedes Device ermittelt. In dieser Struktur findet sich der *DevicePath*. Als Letztes wird mit *CreateFile* und dem *devicePath* zusammen ein Handle für jedes Device geöffnet. Das Handle wird zuletzt noch in dem dafür vorgesehenen Array gespeichert. Bevor die Applikation beendet wird, müssen alle Handles wieder geschlossen werden.

Mithilfe der *DeviceIoControl* Funktion kann jetzt mit den Devices kommuniziert werden. Abbildung 16 zeigt zwei Beispiele für die Verwendung von

DeviceIoControl.
Bei dem Testaufruf ist nicht viel zu beachten. Es muss nur ein Device Handle

```
void test() {
    bool error = DeviceIoControl(hDevice, IOCTL_ADQPXIE_TEST, NULL, 0, NULL, 0, NULL, NULL);
    std::cout << "IOCTL_ADQPXIE_TEST" << std::endl;
}


void id() {
    VendorDeviceID id;
    DWORD bytesReturned;
    bool error = DeviceIoControl(hDevice, IOCTL_ADQPXIE_VENDOR_DEVICE, NULL, 0, &id, sizeof(VendorDeviceID), &bytesReturned, NULL);
    std::cout << "IOCTL_ADQPXIE_VENDOR_DEVICE" << std::endl;
    std::cout << "Device ID: " << std::hex << id.deviceID << std::endl;
    std::cout << "Vendor ID: " << id.vendorID << std::endl << std::dec;
}
```

Abbildung 16: Usermode IOCTL Aufruf

und der gewünschte IOCTL Code eingesetzt werden.
Für das Ermitteln der IDs muss die *VendorDeviceID* Struktur initialisiert werden. Die Definition befindet sich in *public.h*. Danach müssen zusätzlich zum vorherigen Funktionsaufruf die Adresse von *VendorDeviceID* und deren Größe mitgegeben werden. Damit der Aufruf klappt, wird noch die Adresse von *bytesReturned* erwartet. Wenn alles erfolgreich abgelaufen, ist werden die Device ID und Vendor ID auf der Kommandozeile ausgegeben.
In Abbildung 17 sieht man das PXIe Development Tool. In dem Tool ist der Space-Tab geöffnet. Hier können die einzelnen Register der PXIe-Karten ausgelesen oder beschrieben werden. So lassen sich viele Funktionen der Hardware schnell ausprobieren.
Dieses Tool greift über eine DLL auf alle implementierten Funktionen des Treibers zu. Beim Laden der DLL werden die passenden Device Handles geöffnet. Über die API der DLL können dann die PXIe Karten gesteuert werden.

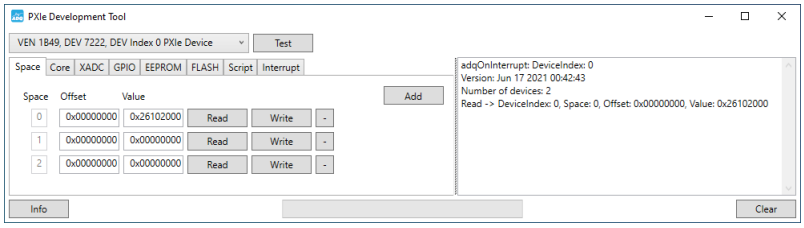


Abbildung 17: GUI des Usermode Testprogramms

4.7 Interrupt Handling

Die Events für das Interrupt Handling sind bereits in dem *AdqPXIeEvtDeviceAdd* Event registriert worden. Nach *PrepareHardware* wird zuerst *AdqP-*

XIeInterruptEnable aufgerufen. Die Funktion ist in Abbildung 18 zu sehen. In der Enable Funktion müssen die Bits z.B. im IP Core gesetzt werden, welche die Interrupts auf der Hardware maskieren. In der *AdqPXIeInterruptEnable* wird der im IP Core eingebaute Test-Interrupt aktiviert.
Entsprechend gibt es auch eine Disable Funktion. Die Disable Funktion wird

```
NTSTATUS AdqPXIeInterruptEnable(
    _In_ WDFINTERRUPT Interrupt,
    _In_ WDFDEVICE Device
)
{
    UNREFERENCED_PARAMETER(Device);
    PDEVICE_CONTEXT deviceContext = DeviceGetContext(WdfInterruptGetDevice(Interrupt));
    union {
        IP_CORE_INTERRUPT bits;
        ULONG ulong;
    } interruptControl;
    interruptControl.ulong = ADQ_READ_REGISTER_ULONG((PULONG)&deviceContext->ipCore->interrupt_control);
    interruptControl.bits.test_interrupt = TRUE;
    ADQ_WRITE_REGISTER_ULONG((PULONG)&deviceContext->ipCore->interrupt_control, interruptControl.ulong);
    return STATUS_SUCCESS;
}
```

Abbildung 18: Enable Interrupt Funktion

```
BOOLEAN AdqPXIeInterruptIsr(
    _In_ WDFINTERRUPT Interrupt,
    _In_ ULONG MessageID
)
{
    UNREFERENCED_PARAMETER(MessageID);
    PDEVICE_CONTEXT deviceContext = DeviceGetContext(WdfInterruptGetDevice(Interrupt));
    BOOLEAN isRecognized = FALSE;
    union {
        IP_CORE_INTERRUPT bits;
        ULONG ulong;
    }interruptStatus;
    union {
        IP_CORE_INTERRUPT_CONTROL_2 bits;
        ULONG ulong;
    } interruptControl_2;
    // reading the status register clears the interrupt
    interruptStatus.ulong = ADQ_READ_REGISTER_ULONG((PULONG)&deviceContext->ipCore->interrupt_status);
    if (interruptStatus.bits.test_interrupt == 0) {
        isRecognized = TRUE;
        interruptControl_2.bits.invoke_test_interrupt = FALSE;
        ADQ_WRITE_REGISTER_ULONG((PULONG)&deviceContext->ipCore->interrupt_control_2, interruptControl_2.ulong);
    }
    if (isRecognized) {
        WdfInterruptQueueDpcForIsr(deviceContext->Interrupt);
    }
    return isRecognized;
}
```

Abbildung 19: Interrupt ISR Funktion

vor *RemoveHardware* aufgerufen. Andersherum ist es nicht möglich, da *RemoveHardware* das Mapping der virtuellen Adressen aufhebt.
Beim Auftreten eines Interrupts wird die Callback Funktion Interrupt Service Routine (ISR) (*AdqPXIeInterruptIsr*) aufgerufen. Eine Implementation ist in Abbildung 19 zu sehen.
In dieser Funktion muss geprüft, werden ob der Interrupt vom unterstützten Device kommt. Wenn der Interrupt von einem fremden Gerät kommt, muss *FALSE* zurückgegeben werden, andernfalls ist die Routine verpflichtet den Interrupt zu bearbeiten und mit *TRUE* zu beantworten. Eine dritte Option

ist es, den Interrupt an den nächsthöheren Treiber im Treiberstack weiter zu kaskadieren. Im letztem Schritt wird mit *WdfInterruptQueueDpcForIsr* ein Deferred Procedure Call (DPC) erstellt.
Die *AdqPXIeInterruptIsr* Funktion sollte sich so schnell wie möglich beenden, denn sie läuft im *DISPATCH_LEVEL*. In ihr werden im Normalfall nur zeitkritische Daten erfasst, z.B. Inhalte von Registern. Erst in der *InterruptDpc* Routine werden die Daten verarbeitet.
Hintergrund ist, dass *AdqPXIeInterruptIsr* im IRQ-*DISPATCH_LEVEL* Interrupt Request Level (IRQL) läuft. Auf niedrigen IRQL's sollten grundlegend nur minimale Zeitspannen verbracht werden (vgl. Hudek u. a. 2021a). In der *InterruptDpc* Funktion wird jeder Interrupt für das PXIe Development Tool gezählt. Mit dem Inverted Call Model wird auch der User Mode bei einem Interrupt benachrichtigt. Das Model wurde von OSR (*https://www.osr.com/inverted-call-model-kmdf/*) vorgestellt.
In Abbildung 20 sieht man die GUI mit dem Tab für die Interrupts. Die User

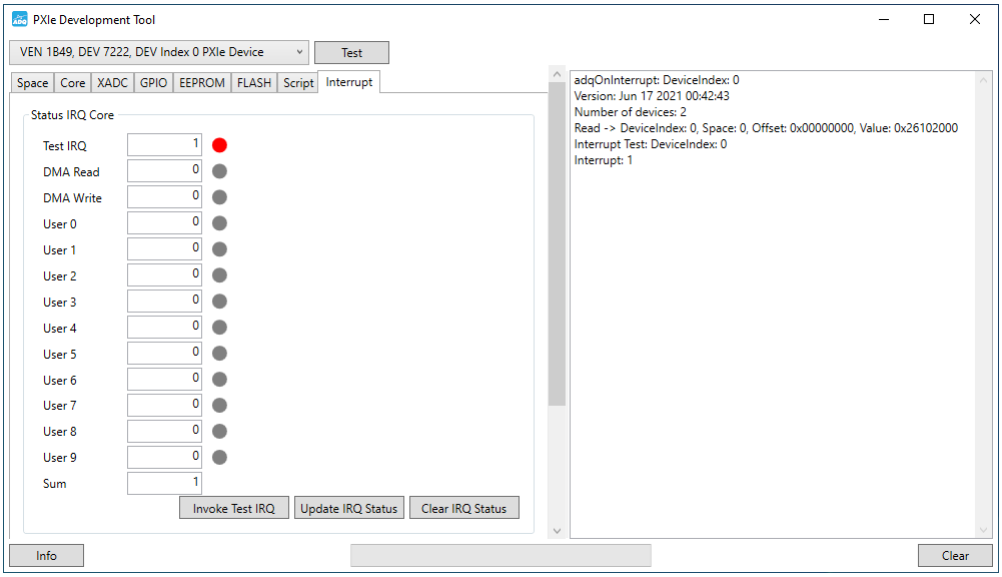


Abbildung 20: GUI für Interrupt Zähler

Interrupts 0-9 sind die 10 Interrupt Leitungen vom IP Core. Der rote Punkt zeigt den letzten Interrupt an. Daneben befindet sich jeweils ein Zähler für die Interrupts.

4.8 Deployment

Allem voran sollte ein neuer Build via Visual Studio gemacht werden. Bei allen erstellten Dateien muss unbedingt auf Datum und Uhrzeit geachtet

werden. Bedauerlicherweise geschehen genau hier die meisten Fehler. Visual Studio versucht extrem viel Optimierung durch Cashing, Precompiled Header oder Ähnliches zu erreichen. Einen behobenen Fehler in alter Software zu haben, kann Einiges an Zeit bei der Fehlersuche kosten.
Das Installieren des Treiber und dem dazu gehörigen User Mode Programm ist etwas aufwendiger als bei einer C++ Konsolen Anwendung. Die Installation auf dem Entwicklungs-Rechner ist zwar prinzipiell möglich, jedoch mit einigen Problemen verbunden. Bei einem Fehler im Treiber ist ein BSoD sehr wahrscheinlich. Je nach aufgetretenem Fehler kann es sein das sich Windows nicht mehr starten lässt, weil Systemdateien oder anderes beschädigt worden sind.
Um dies zu vermeiden, wird der Treiber auf einem zweiten Rechner installiert. Dieser Prozess kann mit Visual Studio automatisiert werden. Dabei müssen sich der Test- und Entwicklungs-Rechner im selben Netzwerk befinden. Dies war ursprünglich auch erfolgreich eingerichtet. Leider hat ein Visual Studio Update diese Automatisierung zunichtegemacht. Seitdem war eine neue Konfiguration dieses Setups nicht mehr erfolgreich. Deshalb werden jetzt per Post-Build Script alle nötigen Dateien auf ein Netzwerklaufwerk kopiert. So können Hardwareentwickler und auch der Testcomputer den neusten Treiber darüber beziehen.
Daraufhin muss der Treiber auf dem Target-Rechner installiert werden. Der Treiber lässt sich mit *devcon install <INF file> <hardware ID>* über die Kommandozeile als Administrator installieren. Alternativ kann auch der Windows Device Manager dafür verwendet werden. Windows zeigt hier zusätzlich auch unbekannte PCI Hardwarekomponenten an, über einen Rechtsklick auf das Device lässt sich dann auch hier der Treiber via GUI installieren.
Wenn der Treiber fertig entwickelt ist und an den Kunden ausgeliefert werden soll, muss er zuvor noch vom Windows Hardware Developer Center Dashboard Portal signiert werden (vgl. Hudek, Coulter und Cymoki 2021). Ohne diese Signatur lässt sich der Treiber nur mit deaktiviertem Secure Boot im Basic Input/Output System (BIOS) installieren.

5 Testing

Im letzten Teil der Arbeit soll das aufwendigere Setup bzw. Einrichten eines Kernel Debuggers zum Testen erläutert werden. Danach wird noch auf die Verwendung des Debuggers eingegangen. Zuletzt wird erklärt, wie Logging im Kernel mithilfe von Windows Software Trace Preprocessor (WPP) realisiert werden kann.

5.1 Kernel Mode Debugging

Um den Windows Kernel zu debuggen, werden zwei Rechner benötigt, je einen Host- und einen Target-Computer. Hintergrund dafür ist:
Sobald ein gesetzter Brakepoint den Kernel stoppt, friert der Computer ein. Es ist keine Interaktion mehr möglich(vgl. Yosifovich 2019, S. 36f).
Die beiden PCs müssen zu Beginn über Ethernet miteinander verbunden werden. Am besten wird dies mit einem LAN Kabel durchgeführt. Eine möglichst direkte Verbindung ist hier vorteilhaft, weil damit Latenzen sehr gering bleiben und andere Netzwerkteilnehmer nicht beeinträchtigt werden.
Beide Computer müssen Teilnehmer im selben IP-Subnetz sein. Die korrekte Verbindung der beiden PC's kann im DOS Fenster mit Hilfe des
ping <IP Adresse> Befehls überprüft werden.
Im weiteren Verlauf werden *kdnet* und *WinDbg* verwendet. Beide Tools sollten bereits durch die Installation von WDK oder dem Windows 10 SDK Paket auf dem Rechner zu finden sein. Auf dem Target Computer müssen zuvor noch folgende Dateien kopiert werden *kdnet.exe*, *VerifiedNICList.xml*. Beide Dateien befinden sich Standardmäßig unter
C:\Program Files (x86)\Windows Kits\10\Debuggers\x64
auf dem Host Computer. Gespeichert werden sie beim Target Rechner unter *C:\KDNET*.
Auf dem Target-Computer muss jetzt folgender Befehl mit Admin Rechten ausgeführt werden:
kdnet.exe
Dieser Befehl sollte bei Erfolg den zugehörigen Network Interface Controller (NIC) anzeigen. Weiter folgt:
kdnet.exe <HostComputerIPAddress> <DebugPort>
Der DebugPort ist in der Abbildung 21 Port 50005. Es kann aber auch ein anderer unbelegter Port verwendet werden.
Dem letzten Befehl folgte ein Key als Ausgabe. Mithilfe des Keys kann auf dem Host PC das Kommando:
windbg -k -d net:port=<DebugPort>,key=<Key>
gestartet werden. Das Argument *-k* gibt hierbei an, dass es sich um eine Kernel Mode Debugging Session handelt. Das Argument *-d* führt zu einen automatischen break wenn der Target-Rechner neu startet.
Durch den Befehl:
shutdown -r -t 0
auf dem Target-Computer, kann dieser neu-gestartet werden. Auf dem Host-PC sollte jetzt die Debbuging Sitzung starten (vgl. Marshall u. a. 2020b).
Damit WinDbg wie in Abbildung 21 Programmcode anzeigt, müssen noch die Symboldateien geladen werden. Visual Studio erstellt diesen automatisch

für die Debug Build Version. Es handelt sich hier um die Program Database (PDB) Dateien, welche im Projektordner im *bin* Verzeichnis zu finden sind.
Das Kommando:
*.sympath cache*C:\MySymbols*
gibt dem Debugger das Verzeichnis an, in dem nach PDB Dateien gesucht werden soll. Final werden die Symbole mit:
.reload
geladen (vgl. Marshall u. a. 2020a).
Mit dem Befehl:
*lm m TreiberModule**
lassen sich Status und Speicherort des Treibers herausfinden. Wenn nichts angezeigt wird, wurde der Treiber nicht geladen.
Um sich die Funktionen des Treibers anzeigen zu lassen, wird das Kommando:
x /D TreiberModule!Symbol
verwendet. Der letzte Parameter *Symbol* kann etwa ein Funktionsname oder WDF-Objektname sein. Der Parameter */D* erzeugt einen anklickbaren Link hinter der Ausgabe in der Konsole.
Nach dem dieser angeklickt worden ist, wird der Quellcode der entsprechenden Funktion angezeigt (vgl. Marshall u. a. 2021).
Das Setzen von Breakpoints und das anschließende Navigieren ist jetzt auch via GUI möglich.
Ist ein Breakpoint gesetzt, kann mit *go* fortgefahren werden. Währenddessen die Konsole aktiv ist, befindet sich nämlich der Target Computer im Break Mode.
Nun kann erstmals wieder zum Test Rechner gewechselt werden. An diesem wird über die programmierte Konsolen Anwendung oder GUI ein Befehl an den Treiber geschickt. In der Abbildung 21 ist das die Abfrage von Device ID und Vendor ID. Nach dem Auslösen des Befehls friert der Target Computer aufgrund des Breakpoints ein. Am Host Rechner kann nun der Code Schritt für Schritt durchgegangen werden.
Dabei werden die Werte von Variablen ebenfalls angezeigt. Oft können die Variablen hinter den Pointern nicht direkt besichtigt werden. Die Pointer zeigen eventuell auf ein Objekt, welches nicht im Arbeitsspeicher liegt. Beispielsweise könnte das Objekt auf der Festplatte liegen. Da sich der Target-Computer aber im Break Modus befindet, können nur bereits geladene Werte angezeigt werden. Mit der Konsole lassen sich solche Werte auch manuell laden.

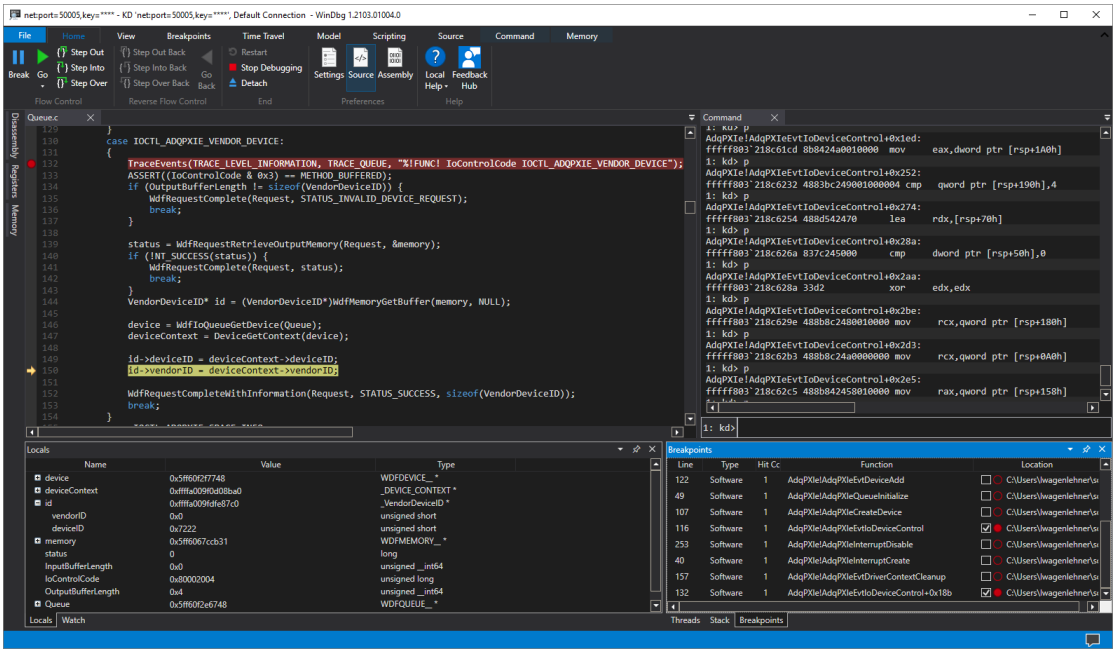


Abbildung 21: Debugging mit WinDbg

5.2 Tracing mithilfe von WPP

Wenn ein Kunde Probleme mit dem Treiber hat, ist das Debuggen oft keine Option. Die Fehlerbeschreibung vom Anwender sind oft sehr ungenau. Aus diesem Grund ist es sinnvoll, Tracing mit in die Software mit einzubauen. So kann ein Anwender bei einem Problem einen Trace erstellen. Dieser Trace enthält dann genaue Informationen, etwa welche Funktionen aufgerufen worden sind, was für Events aufgetreten sind und den Inhalt ausgewählter Variablen. Mit dem WPP lässt sich eine solche Lösung für Kernel Treiber realisieren.
Initial muss WPP in den Projekteigenschaften vom Treiber aktiviert werden. Die *Driverentry* Funktion muss jetzt um folgende Zeile erweitert werden:

WPP_INIT_TRACING(DriverObject, RegistryPath)

Auch *ContextCleanup* muss erweitert werden:

WPP_CLEANUP(WdfDriverWdmGetDriverObject((WDFDRIVER)DriverObject));

Beim Aktivieren von WPP Tracing ist die *Trace.h* generiert worden. In den Header muss eine neue GUID (Tracing GUID) eingefügt werden. Des Weiteren können hier im *WPP_DEFINE_CONTROL_GUID* eigene Tracing Flags erstellt werden. Diese

helfen, den Trace übersichtlicher zu gestalten.
Zuletzt muss in allen Dateien des Treibers, in denen das Tracing verwendet werden soll, ein Header mit *<dateiname>.tmh* angelegt werden (vgl. Hudek, Coulter und Varma 2017). WPP erstellt dann beim Build automatisch den nötigen Code in den entsprechenden TMH Dateien.
Ein Trace Event kann z.B. folgendermaßen aussehen:

TraceEvents(TRACE_LEVEL_ERROR, CUSTOM_BIT, "Message");

Die *Message* ist ähnlich zu *printf* und unterstützt auch verschiedene Parameter.
Mit dem Tool *Logman* und der Tracing GUID kann jetzt ein Trace erstellt werden. Dabei wird dann eine ETL Datei erzeugt.
Zur graphischen Auswertung kann hierfür dann der *Windows Performance Analyzer* verwendet werden.
Zuletzt noch eine Bemerkung:
In Abbildung 18 sieht man den Präfix *ADQ_* vor dem *READ_REGISTER_ULONG*. Dabei handelt es sich um eine Wrapper Funktion. Diese traced alle Register Befehle im Debug Build.

6 Fazit und Ausblick

In den entwickelten Treiber wurden alle geforderten Funktionen erfolgreich eingebaut. Vereinbarungsgemäß wird DMA erst im nächsten Schritt bei der Programmierung des Treibers implementiert. Zusätzlich zum Treiber wurde für Testzwecke ein PXIe Development Tool programmiert. Dies war sowohl bei der Entwicklung der Software als auch beim Testen der Hardware eine große Hilfe.
Das Projekt war ein großer Erfolg. ALLDAQ hat deshalb beschlossen, alle Zukünftigen Karten mit dem hier programmierten Backend auszustatten. Also wird der Treiber in seiner Grundfunktion auch mit späteren PXIe Karten zur Anwendung kommen.

Literaturverzeichnis

[All21] PXI Systems Alliance. *PXI Architecture*. 12. Juli 2021. URL: http://www.pxisa.org/ (besucht am 12. 07. 2021).

[ALL21a] ALLDAQ. *ALLDAQ x4-PCIexpress Expand System "Hybrid"*. 12. Juli 2021. URL: https://shop.alldaq.com/Messen-Steuern-Regeln/PXI-und-Hybrid-Komplettsysteme/ALLDAQ-x4-PCIexpress-Expand-System-Hybrid::184197.html (besucht am 12. 07. 2021).

[ALL21b] ALLDAQ. *Handbuch ADQ-22/23-Serie Rev. 1.1*. 2021.

[ALL21c] ALLDAQ. *Unternehmen*. 12. Juli 2021. URL: https://www.alldaq.com/company/ (besucht am 12. 07. 2021).

[Bro21] Broadcom. *PCI 9056*. 12. Juli 2021. URL: https://www.broadcom.com/products/pcie-switches-bridges/usb-pci/io-accelerators/pci9056 (besucht am 12. 07. 2021).

[Dem13] Klaus Dembowski. *Computerschnittstellen und Bussysteme für PC, Tablets, Smartphones und Embedded-Systeme*. 3. Aufl. VDE Verlag, 2013.

[Gla19] Eduard Glatz. *Betriebssysteme Gundlagen, Konzepte, Systemprogrammierung*. 4. Aufl. dpunkt.verlag, 2019.

[HCC21] Ted Hudek, D. Coulter und Cymoki. *Driver Signing*. 13. Juli 2021. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing (besucht am 13. 07. 2021).

[HCV17] Ted Hudek, D. Coulter und S. Varma. *Using WPP Software Tracing in KMDF Drivers*. 20. Apr. 2017. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/wdf/using-wpp-software-tracing-in-kmdf-drivers (besucht am 12. 07. 2021).

[Hud+21a] Ted Hudek u. a. *Managing Hardware Priorities*. 22. Juli 2021. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/managing-hardware-priorities (besucht am 12. 07. 2021).

[Hud+21b] Ted Hudek u. a. *Overview of INF Filesl*. 12. Juli 2021. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/overview-of-inf-files (besucht am 12. 07. 2021).

[HVS21a] Ted Hudek, A. Viviano und M. Sasouvanh. *Overview of UMDF*. 13. Juli 2021. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/wdf/overview-of-the-umdf (besucht am 13. 07. 2021).

[HVS21b] Ted Hudek, A. Viviano und T. Sherer. *Types of WDM Drivers*. 12. Juli 2021. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/types-of-wdm-drivers (besucht am 12. 07. 2021).

[Mar+20a] Don Marshall u. a. *Getting Started with WinDbg (Kernel-Mode)*. 2. Juni 2020. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/getting-started-with-windbg--kernel-mode- (besucht am 12. 07. 2021).

[Mar+20b] Don Marshall u. a. *Setting Up KDNET Network Kernel Debugging Automatically*. 11. Nov. 2020. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-a-network-debugging-connection-automatically (besucht am 12. 07. 2021).

[Mar+21] Don Marshall u. a. *Getting Started with WinDbg (Kernel-Mode)*. 5. Apr. 2021. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debug-universal-drivers---step-by-step-lab--echo-kernel-mode-#section-5-use-windbg-to-display-information-about-the-driver (besucht am 12. 07. 2021).

[PS21] PCI-SIG. *Specifications*. 12. Juli 2021. URL: https://pcisig.com/ (besucht am 12. 07. 2021).

[SGG19] Abraham Silberschatz, P. Baer Galvin und G. Gagne. *Operating System Concepts*. 10. Aufl. Wiley, 2019.

[Sma20] Smartlogic. *Multichannel DMA Flex IP Core for PCI-Express User Guide*. 2020.

[Sta18] William Stalling. *Operating Systems Internals and Design Principles*. 2018.

[XIL18] XILINX. *All Programmable 7 Series Product Selection Guide*. 2018.

[Yos19] Pavel Yosifovich. *Windows Kernel Programming*. Leanpub, 2019.