Institut für Grundlagen der Elektrotechnik und Messtechnik
Fachgebiet Sensorik + Messtechnik

Leibniz Universität Hannover

Bachelorarbeit

Entwicklung und Optimierung von Software für eine mikrocontrollerbasierte Steuerung einer Messeinheit mit High-Speed USB-Schnittstelle

von

Niklas Rother
Matrikelnummer: 2942200

Betreuer:
M.Sc. Alexander Bohnhorst

Prüfer:
1. Prof. Dr.-Ing. Holger Blume
2. Prof. Dr.-Ing. Stefan Zimmermann

August 2015

Bachelorarbeit von Herrn Niklas Rother

Matrikelnummer: 2942200

Studiengang: Informatik

Thema: **Entwicklung und Optimierung von Software für eine mikrocontrollerbasierte Steuerung einer Messeinheit mit High-Speed USB-Schnittstelle**

Aufgabenstellung:

Im Rahmen einer Bachelorarbeit erhält Herr Niklas Rother die Aufgabe die Software für die Steuerung und Kommunikation einer auf einem 32-bit Mikrokontroller basierende Messeinheit zu entwickeln. Im ersten Teil der Arbeit soll Herr Rother durch eine geeignete Treiberschnittstelle sicherstellen, dass die neue Messeinheit zu der bisher verwendeten in LabView geschriebenen Kontrollsoftware kompatibel ist, gleichzeitig jedoch eine höhere Performance bietet und den Funktionsumfang der Messeinheit unterstützt.

Als Herzstück des am Institut vorhandenen Einschubsystems ist der von Herrn Rother programmierte Mikrokontroller für die Auswertung der Messdaten und Steuerung des gesamten Messsystems verantwortlich. Die Software muss daher sicherstellen, dass Spannungspulse mit einer Pulslänge von weniger als 100 ns erzeugt und synchron dazu mittels eines 24-bit Analog-Digital-Wandler die Messdaten erfasst werden können. Dabei liegt aufgrund der zeitlich kritischen Anwendung das Hauptaugenmerk darauf, ein Konzept zu entwickeln und umzusetzen, bei dem es, etwa durch die Kommunikation mit dem Messcomputer oder andere Störeinflüsse, nicht zu Fehlern in den erfassten Messdaten kommen kann. Ferner ist hinsichtlich der anhaltenden Entwicklung des Messsystems eine umfangreiche Dokumentation des Programmcodes nötig, sodass eventuelle Änderungen auch von Dritten schnell eingebunden werden können.

Erstprüfer:	Prof. Dr.-Ing. Holger Blume
Zweitprüfer:	Prof. Dr.-Ing. Stefan Zimmermann
Betreuer:	M.Sc. Alexander Bohnhorst
Beginn der Arbeit:	14.04.2015
Ende der Arbeit:	14.08.2015

Prof. Dr.-Ing. Holger Blume

Eigenständigkeitserklärung

Ich versichere hiermit, dass ich die vorstehende Bachelorarbeit mit dem Titel:

Entwicklung und Optimierung von Software für eine mikrocontrollerbasierte Steuerung einer Messeinheit mit High-Speed USB-Schnittstelle

selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich in jedem einzelnen Fall durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

Ort, Datum	Unterschrift

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungen

ADC Analog Digital Converter

DMA Direct Memory Access

EEPROM Electrically Erasable Programmable Read-Only Memory

GPIO General Purpose Input/Output

GPT General Purpose Timer

HID Human Interface Device

I²C Inter-Intergrated Circut

IDE Integrierte Entwicklungsumgebung

IMS Ionenmobilitätsspetroskopie

PWM Pulsbreitenmodulation

SPI Serial Peripherial Interface

UART Univeral Asynchronus Receiver/Transmitter

ULIP USB Low Pin Interface

USB Universal Serial Bus

USB IF USB Implementers Forum

WCID Windows Common Device Identification

ZLP Zero-Length-Package

1 Einleitung

Eine digitale Aufzeichnung und Verarbeitung von Messdaten ist aus der modernen Laborpraxis nicht mehr wegzudenken. Die Erfassung der Messwerte erfolgt dabei für gewöhnlich mit einem spezialisierten Gerät, die weitere Verarbeitung meistens auf einem verbundenen Computer, der über deutlich mehr Rechenleistung verfügt. Zur Verbindung zwischen Computer und Messgerät kam in der Vergangenheit oft eine serielle Schnittstelle nach RS-232 zum Einsatz, heutiger Standard ist jedoch – nicht nur für die Verbindung zu Messgeräten – eine Schnittstelle nach dem USB-Standard.

Am *Institut für Grundlagen der Elektrotechnik und Messtechnik*, an dem diese Arbeit entstanden ist, kommt ein selbst entwickeltes Bussystem zum Einsatz, das die flexible Kombination von Mess- und Steuergeräten in einem gemeinsamen Gehäuse ermöglicht. Alle verwendbaren Module sind dabei auf Platinen mit einem standardisierten Stecker untergebracht. Eine dieser Karten ist die *USB-Messkarte*, welche die Verbindung zwischen Bussystem und Computer über eine USB-Schnittstelle herstellt. Neben dieser Aufgabe ist eine wesentliche Funktion der Karte die Aufzeichnung von Spannungsverläufen, die als *Spektren* bezeichnet werden, sowie die Erzeugung von Spannungsimpulsen mit definierter Dauer, die von anderen Karten im System verarbeitet werden.

Die USB-Messkarte wird mit dem Mikrocontroller AT90USB1287 von Atmel betrieben, dessen Leistungsfähigkeit sich im Laufe der Zeit als Engpass des Systems herausstellte. Es wurde daher eine neue Karte unter dem Namen *Datenerfassungsmodul* entwickelt, die von einem Mikrocontroller des Typs TM4C1294NCPDT von Texas Instruments angetrieben wird, der deutlich mehr Leistungsreserven hat. Im Rahmen dieser Arbeit wurde die Software erstellt, mit der die neue Datenerfassungskarte betrieben wird. Die wichtigsten Anforderungen dabei waren:

Kompatibilität zur USB-Messkarte Die neue Karte sollte die gleichen Aufgaben wie die alte Version übernehmen, und daher in allen Punkten so kompatibel wie möglich sein.

Höhere Auflösung der Spektrenerfassung Sowohl die vertikale als auch die horizontale Auflösung der Spektren sollte erhöht werden. Zu diesem Zweck kommen auf dem Datenerfassungsmodul leistungsfähigere Analog-Digital-Wandler zum Einsatz (siehe Abschnitt 2.2.4).

Vorverarbeitung der Spektren auf der Karte Durch die erhöhte Auflösung der Spektren steigt die erfasste Datenmenge deutlich an. Um den Messcomputer zu entlasten, sollte es möglich sein, eine variable Anzahl der Spektren direkt auf der Karte aufzusummieren.

Höhere Auflösung der erzeugten Spannungspulse Die zeitliche Auflösung und die minimale Breite der Spannungsimpulse sollten verbessert werden. Ziel waren hier Spannungsimpulse von weniger als 100 ns Breite.

Synchronisation zwischen Pulsen und Spektrenerfassung Der zeitliche Abstand zwischen der Aussendung von Pulsen und Erfassung der Spektren muss in jedem Fall konstant sein, da ansonsten eine weitere Verarbeitung der Daten unmöglich ist. Eine Störung dieser Synchronisation muss in jedem Fall verhindert werden.

Übertragung der Daten per USB 2.0 Die USB-Messkarte verfügt nur über eine Schnittstelle nach dem USB 1.1 Standard, zudem stellt die Datenübertragungsgeschwindigkeit mit aktuell 666 kB/s einen Engpass dar. Das Datenerfassungsmodul verfügt über eine USB 2.0 Schnittstelle, die Übertragungsgeschwindigkeit sollte deutlich erhöht werden.

Einbindung in *LabView* Da die Auswertung der Messergebnisse in *LabView* erfolgt, sollte eine Einbindung entwickelt werden, die möglichst kompatibel mit der schon bestehenden Software ist.

2 Grundlagen zum Aufbau des Systems

In diesem Kapitel soll ein Überblick über den grundlegenden Aufbau der Software gegeben werden. Dabei sollen die wichtigsten Design-Entscheidungen dargelegt, sowie alle verwendeten Protokolle dokumentiert werden. Auf die konkrete Implementierung soll an dieser Stelle noch nicht eingegangen werden, Hinweise dazu sind in Kapitel 6 zu finden.

2.1 Die Architektur des Systems

2.1.1 Aufgaben der Datenerfassungskarte

Die Datenerfassungskarte, deren Software im Rahmen dieser Arbeit entwickelt wurde, muss eine Vielzahl von Aufgaben erfüllen. Zu den wichtigsten zählen dabei die Kommunikation mit dem Host-Rechner über den Universal Serial Bus (USB), die Erfassung von Spektren mit den auf der Karte vorhandenen Analog Digital Converter (ADC), die Erzeugung von Spannungspulsen und die Kommunikation per Inter-Intergrated Circut (I^2C) mit den weiteren, im Bussystem vorhandenen Karten. Neben der Software, die auf der Karte, bzw. dem dort verbauten Mikrocontroller, läuft, wurde im Rahmen dieser Arbeit auch noch Software für die Ansteuerung der Karte aus dem Programm *LabView* heraus erstellt. Die Schnittstelle zu *LabView* sowie die Anbindung an das Bussystem sollten soweit wie möglich kompatibel zu der bestehenden USB-Messkarte bleiben.

Die Anforderung nach Kompatibilität schließt auch das einheitliche Verfahren mit ein, in dem die Karten konfiguriert werden, die sogenannten „Sätze“[1]. Bei diesen handelt es sich um Register-artige Datenspeicher auf den einzelnen Karten, in denen alle vom Benutzer konfigurierbaren Einstellungen zusammen mit einigen Metadaten, wie etwa einer Beschreibung, gespeichert werden. Die Sätze der verschiedenen Karte können über ein entsprechendes Protokoll auf Basis von I^2C über den Bus von einer Master-Karte angesprochen werden.

Die Datenerfassungskarte muss zum einen also die Möglichkeit bieten, auf die Sätze der anderen Karte zuzugreifen und dabei zu den bestehenden *LabView*-Blöcken soweit wie möglich kompatibel bleiben, zum anderen aber auch selbst über solche Sätze konfiguriert werden[2].

[1] Im Quelltext auch englisch als „Sets“ bezeichnet. Vermutlich handelt es sich bei dieser Bezeichnung – die aus dem bestehenden Quellcode übernommen wurde – um eine Verkürzung des englischen *datasets*, wie die „Sätze“ vermutlich eine Verkürzung von *Datensätze* sind.

[2] Diese Anforderung resultierte in einer sehr hohen Zahl an Sätzen der Datenerfassungskarte von mehr als 40, die bestehenden Karten haben nur selten ein zweistellige Anzahl an Sätzen.

2.1.2 Grundlegende Architektur

Zur Planung und Implementierung einer solch umfangreichen Software ist es nötig, sie in handhabbare Teile zu zerlegen. Für die Software auf der Karte wurde ein Modulkonzept verwendet. Jedes der Module der Software ist größtenteils in sich abgeschlossen und dafür entworfen, auch außerhalb dieser Software, etwa auf einer anderen Karte die ähnliche Funktionen erfüllt, wiederverwendet zu werden. Ein grundlegender Überblick über den Aufbau der Software – die Architektur – ist in Abbildung 2.1 zu sehen.

Anhand dieser Abbildung soll der Datenfluss einiger beispielhafter Anfragen des Hostsystems dargestellt werden. Hierbei soll nur ein genereller Überblick über das System gegeben werden um eine bessere Einordnung der in den nachfolgenden Abschnitten dargestellten Beschreibungen zu ermöglichen. Auf eine ausführliche Beschreibung der Komponenten wird hier bewusst verzichtet. Zur Demonstration soll hier zunächst die Abfrage des Namens einer bestimmten externen Karte verwendet werden. Die Anfrage beginnt damit, dass das *LabView*-Programm auf dem Host-Computer eine entsprechende Anfrage stellt, indem der entsprechende vordefinierte Block aus der Block-Bibliothek aufgerufen wird. In dem Block wird eine Anfrage nach dem I^2C-Protokoll – im Folgenden auch als *BusControl* bezeichnet – zusammengesetzt. Sie enthält im Wesentlichen die Nummern von Satz und Parameter, die gelesen werden sollen. Dieses Paket wird nun noch innerhalb der *LabView*-Blöcke in ein Paket nach dem USB-Transportprotokoll eingepackt. In Abbildung 2.2 ist das *BusControl*-Paket in blau zu sehen, das umschließende Paket des USB-Transportprotokolls in rot. Dieser Datenblock wird nun an die Treibersoftware übergeben, welche die Daten mit Hilfe des USB-Stacks des Betriebssystems an den Mikrocontroller überträgt. Dabei werden die Daten noch in ein relativ einfaches, weiteres Protokoll eingefasst, das auf der Karte direkt beim Empfang wieder entfernt wird – in der Abbildung ist dieses Protokoll in grün dargestellt. Auf dem Controller wird nun zuerst das USB-Transportprotokoll entpackt und anhand der darin enthaltenen Daten entschieden, wie mit den inneren Daten verfahren werden soll. In diesem Fall sollen die Daten per I^2C an eine externe Karte übertragen werden, die Bus-Adresse der Karte ist dabei im USB-Transportprotokoll enthalten. Nach der Übertragung der Daten per I^2C wird die Antwort der externen Karte empfangen und als Antwort per USB an den Host zurückgeschickt. Die Daten werden dabei in der gleichen Weise wieder in mehreren Schichten ein- und ausgepackt. Die Übertragung per I^2C ist zusätzlich noch mit einer Checksumme abgesichert, die auf dem Datenerfassungsmodul generiert wird.

Als zweites Beispiel soll das Auslesen des Wertes des Satzes, der die aktuelle Chip-Temperatur der Karte enthält dienen. Die Anfrage nimmt zunächst den gleichen Weg wie in dem vorangegangen Beispiel und wird auch per I^2C auf den Bus übertragen. Da die Karte ihre eigene Bus-Adresse spezifiziert hat, wird die Übertragung aber auch von der Karte selbst empfangen. Wenn die Daten von dem Slave-Modul empfangen wurden, wird das interne Modul zur Verwaltung der Sätze verwendet, um den angefragten Wert zu ermitteln. Die Antwort nimmt nun wieder den umkehrten Weg über den I^2C-Bus und den USB bis zum Host-Computer.

Als drittes Beispiel soll die Erfassung eines einzelnen Spektrums dienen[3], der hier beschriebene Datenfluss ist auch mit Pfeilen in Abbildung 2.1 eingezeichnet. Um der Karte Befehle, wie „Erfasse ein Spektrum!“ zu erteilen, muss ein spezieller Wert in einen dafür reservierten Kommandosatz geschrieben werden. Das Paket nimmt den gleichen Weg wie im Beispiel zuvor, beim Empfang des Kommandos wird notiert, dass ein Spektrum erfasst werden soll. Die Erfassung eines Spektrum ist normalerweise mit der Erzeugung der Spannungspulse synchronisiert, daher wird nicht sofort mit der Erfassung begonnen, sondern auf ein Signal vom Puls-Modul gewartet[4]. Wenn dieses Signal eintrifft (in der Zeichnung als „Trigger“ bezeichnet), wird aufgrund der vorher gesetzten Markierung mit der Aufzeichnung eines Spektrums begonnen. Wenn das Spektrum erfolgreich aufgezeichnet wurde, werden die Daten per USB, in einem von *BusControl*-System getrennten Kanal, übertragen. Wurden die Daten vom Host vollständig empfangen, wird dies *LabView* durch die Treibersoftware mitgeteilt, wo die Daten dann weiterverarbeitet werden.

2.2 Entwurf der Software-Module

2.2.1 Das *BusControl*-System

Für die Kommunikation zwischen Datenerfassungskarte und den anderen Steckkarten des Bussystems wird ein auf I^2C aufbauendes Protokoll verwendet, über das es möglich ist, die Sätze der Karten zu lesen und zu schreiben. Um Verwirrung mit dem darunterliegenden I^2C-Hardware-Protokoll zu vermeiden, wurde dieses Protokoll *BusControl* getauft. Das Protokoll ist Teil der Spezifikation des Bussystems und wurde unverändert von der USB-Messkarte übernommen, um die Kompatibilität zu den schon bestehenden Karten zu wahren.

Die Implementierung besteht aus zwei einzelnen Modulen sowie einer Funktionssammlung, die von beiden Teilen genutzt wird. Das *Master*-Modul stellt einen I^2C-Master bereit, das *Slave*-Modul entsprechend einen I^2C-Slave. Die Slave-Adresse bekommt jede Karte durch ihre physische Position im Gehäuse zugeordnet, zu diesem Zweck sind sieben Pins des Bussteckers je nach Steckplatz unterschiedlich mit High- oder Low-Potenzial verbunden. Auf diese Weise wird sichergestellt, dass jede Karte in jedem Fall eine eindeutige Adresse zugeordnet bekommt, auch wenn etwa zwei ansonsten identische Karten verwendet werden. Alle *BusControl*-Transaktionen, auch solche, die die Datenerfassungskarte selbst betreffen, werden unabhängig davon immer auf den I^2C-Bus des Bussystems übertragen[5], und in dem Fall, dass die Karte selbst angesprochen ist, von dem Slave-Modul derselben Karte beantwortet. Diese Art der Implementierung hat einige Vorteile, unter anderem kann eine Datenerfassungskarte so auch als passive Karte im Bussystem verwendet werden,

[3] Im Normalbetrieb wird die Karte meistens so eingestellt, dass die Spektren kontinuierlich erfasst und übertragen werden.

[4] Ein IMS generiert nicht kontinuierlich, sondern periodisch Daten. Eine solche Periode beginnt mit der Aussendung von Pulsen. Es ist folglich nötig, Datenerfassung und Pulsgenerierung zu synchronisieren, um sinnvolle Daten zu erhalten.

[5] Im Gegensatz zu der USB-Messkarte, bei der Übertragungen, welche die Karte selbst betreffen, in einer höheren Schicht „abgezweigt“ wurden.

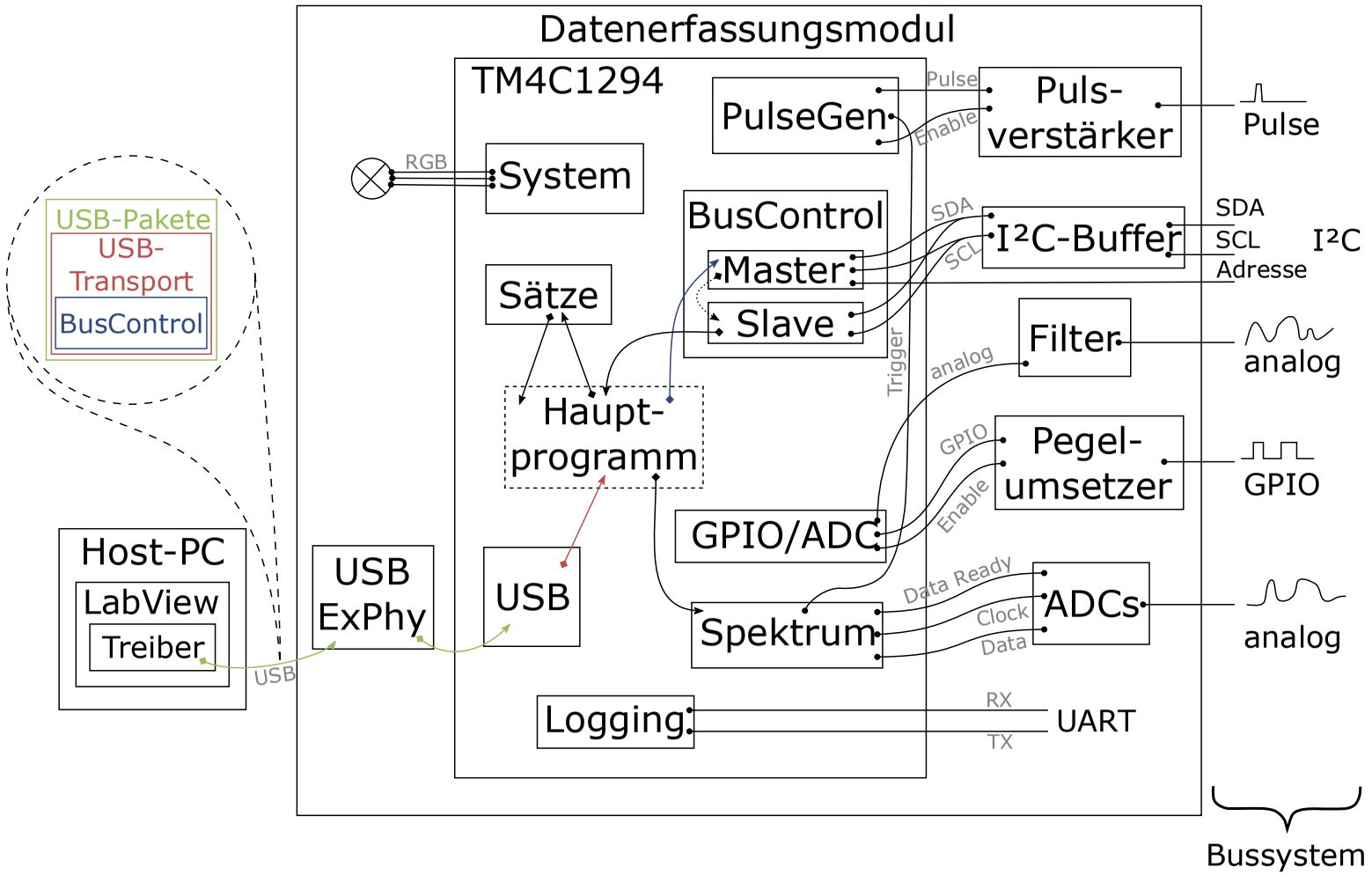


Abbildung 2.1: Übersicht über die grundlegende Architektur des Systems. Die Module des Systems sind als Kästen in dem TM4C129-Block dargestellt, die Linien stellen die wichtigsten Zusammenhänge dar. Der in Abschnitt 2.1.2 als drittes Beispiel beschriebene Datenfluss ist mit Pfeilen verdeutlicht

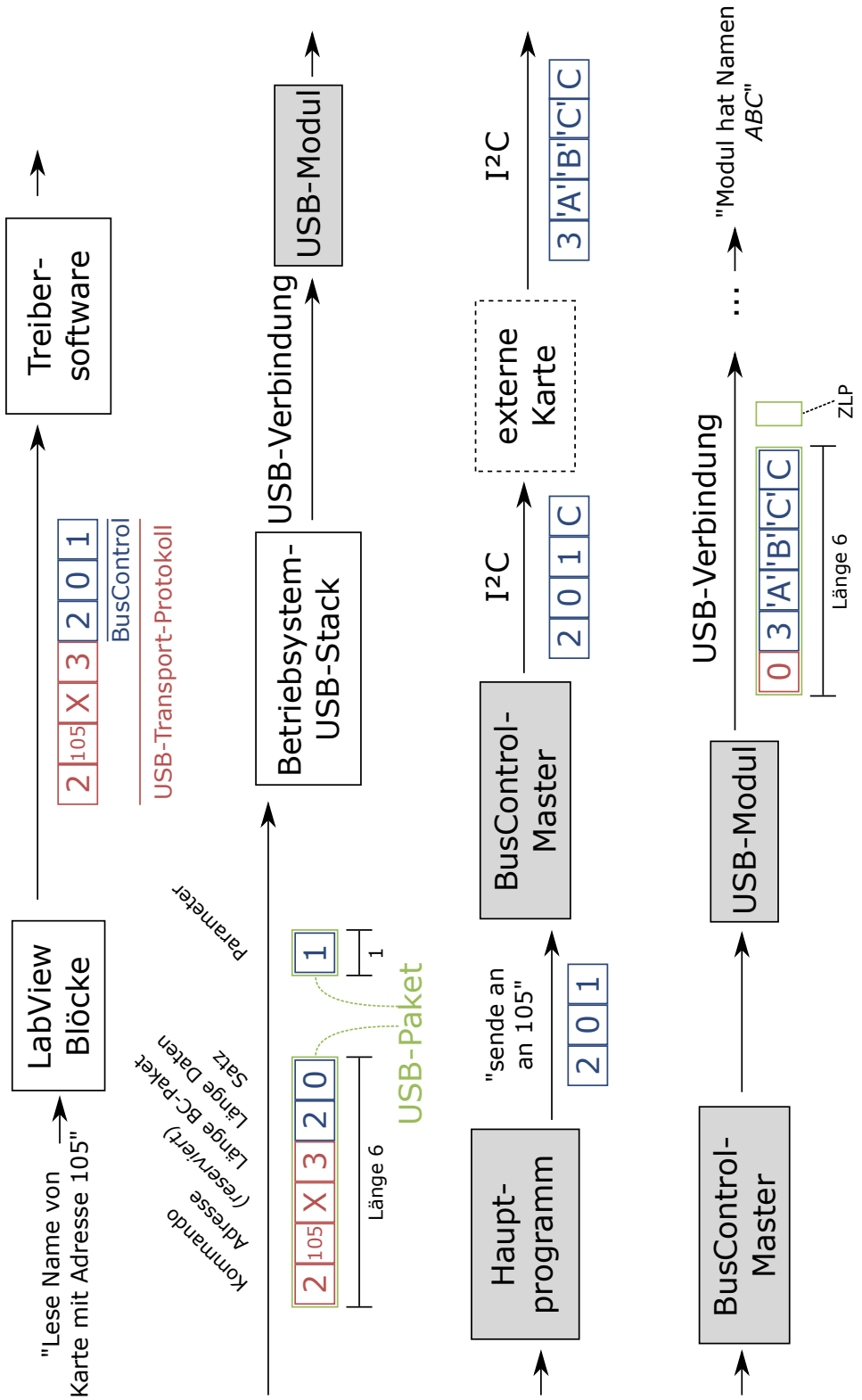


Abbildung 2.2: Beispiel des Datenflusses durch die verschiedenen Module der Software. Die hier dargestellte Transaktion entspricht dem ersten Beispiel in Abschnitt 2.1.2.

ohne etwas an der Software der Karte ändern zu müssen. Dies kann etwa nützlich sein, wenn mehr Spektren erfasst werden sollen, als mit einer Karte möglich ist[6]. Zusätzlich kann auf diese Weise das Slave-Modul ohne weitere Modifikationen in einer zukünftigen Karte, die nur als Slave arbeitet, verwendet werden.

Für den I^2C-Bus sind Pull-Up-Widerstände auf den beiden Leitungen, SCK und SDA, nötig. Im Bussystem sind diese Widerstände auf dem Netzteil untergebracht, einer speziellen Karte, die den Rest des Bussystems mit Strom versorgt. Während der Entwicklung der Datenerfassungskarte wurde diese oft getrennt vom Bussystem verwendet, sodass diese Widerstände nicht vorhanden waren und der I^2C-Bus, welcher ja auch zur internen Kommunikation der Karte verwendet werden muss, nicht funktioniert. Aus diesem Grund wird auf der Karte der I^2C-Buffer PCA9512ADP von NXP verwendet, welcher zum einen die Pegel des Busses auf die 5 V des Bussystems anpasst, zum anderen aber auf der 3.3 V Seite des Busses auf der Karte die nötigen Pull-Up-Widerstände zur Verfügung stellt.

Das *BusControl* Protokoll

Das *BusControl*-System dient ausschließlich dazu, Sätze von Karten zu lesen oder zu schreiben. Von daher ist sein Design eng mit dem des, in Abschnitt 2.2.3 beschriebenen, Satz-Systems verbunden. Die Daten einer Transaktion werden immer als ein durchgängiger Schreibzugriff per I^2C an eine Karte gesendet, wenn die Antwort von Interesse ist, muss sie in einem durchgehenden Lesezugriff gelesen werden. Ein mehrmaliges Lesen der Antwort ohne erneutes Senden einer Anfrage ist nicht spezifiziert[7]. Jede Übertragung beginnt immer mit einem Byte, welches die Länge der *nachfolgenden* Daten angibt. Danach folgt die Nummer des Satzes und des Parameters, der gelesen oder geschrieben werden soll. Wenn ein Schreibvorgang durchgeführt werden soll, folgen danach die Daten, die geschrieben werden sollen. Ein Lesezugriff hat entsprechend immer eine Längenangabe von 2 Byte, bei Schreibzugriffen ist die Längenangabe immer größer als 2 Byte. Als letztes Byte der Übertragung folgt eine CRC-8 Checksumme über die Datenbytes. Diese Checksumme ist nicht in der Längenangabe enthalten und die Checksumme wird auch nicht über die Länge gebildet. Die Checksumme wird dabei ausschließlich bei der Übertragung über I^2C verwendet und auf dem Datenerfassungsmodul generiert. Sie ist nicht Teil der USB-Übertragung. Tabelle 2.1 zeigt das Protokoll.

Als Antwort auf einen Lese- oder Schreibzugriff wird immer der aktuelle Wert des entsprechenden Parameters übertragen, er wird dabei auf die gleiche Weise von Längenbyte und Checksumme eingerahmt wie die restlichen *BusControl*-Transaktionen. Satz- und Parameternummer werden dabei nicht erneut übertragen. Sollte ein Fehler auftreten, wird ein spezielles Fehlerpaket als Antwort gesendet. Dieses hat als Längenangabe den Wert 0xFE[8], danach folgt ein Byte mit der genaueren Bestimmung des Fehlers als

[6] Es ist nicht einmal nötig, das Master-Modul abzuschalten, da dieses Modul sich passiv verhält, wenn es nicht verwendet wird.

[7] ... führt aber sowohl auf der Datenerfassungskarte, als auch auf der USB-Messkarte zum Lesen der Antwort des letzten Befehls.

[8] abgeleitet von dem Wort **Fe**hler

Offset	Länge (Byte)	Bedeutung
0	1	Länge des Datenteils (ohne Länge und Checksumme)
1	1	Satz-Nummer
2	1	Parameter-Nummer
3	n	Daten (optional, bei Schreibzugriff)
3+n	1	Checksumme (CRC-8) über den Datenteil

Tabelle 2.1: Übersicht über das *BusControl*-Protokoll. Die Checksumme wird dabei nur bei der Übertragung per I^2C angefügt.

Code	Konstante (ERROR_*)	Bedeutung
0	NO_ERROR	(Kein Fehler)
1	NO_ANSWER_FROM_SLAVE	Master kann Slave nicht erreichen
2	SLAVE_TIMEOUT	Timeout bei Kommunikation mit Slave
3	NOT_SUPPORTED	USB-Kommando nicht untersützt
4	BUS_NOT_FREE	I^2C-Bus aktuell belegt
5	ILLEGAL_START_STOP	I^2C-Fehler
6	CHECKSUM_SLAVE	Vom Slave gesendete Checksumme falsch
7	CHECKSUM_MASTER	Vom Slave empfangene Checksumme falsch
8	SET_NOT_FOUND	Satz existiert nicht

Tabelle 2.2: Übersicht über die Fehlercodes des *BusControl*- und USB-Transportprotokolls.

Fehlercode[9]. Auch dieses Paket wird mit einer Checksumme abgeschlossen. Tabelle 2.2 zeigt die möglichen Fehlercodes. Dort sind auch einige Fehler angegeben, die im Kontext des *BusControl*-Systems keinen Sinn ergeben, da die gleichen Fehlercodes auch im USB-Transportprotokoll verwendet werden.

2.2.2 Der Pulsgenerator

Für die Ionenmobilitätsspetroskopie (IMS) ist es nötig, mindestens zwei elektrische Impulse mit festgelegter Dauer und definiertem zeitlichen Versatz erzeugen zu können. Mit dem ersten Puls werden die Ionen erzeugt, mit dem zweiten dann in die Messapperatur gegeben. Um diese Pulse erzeugen zu können, stehen in dem Einschubsystem insgesamt 6 Pulskanäle zur Verfügung[10], die alle von der Datenerfassungskarte angesteuert werden sollen.

Im ursprünglichen Platinenentwurf waren zunächst sogenannte Pulsfilter angedacht, die von mehreren aufeinanderfolgenden Pulsen innerhalb einer gewissen Zeit nur den ersten weitergeben und die restlichen unterdrücken. Diese Schaltung war vorgesehen, weil es zunächst nicht sicher war, ob es mit dem gewählten Mikrocontroller möglich sein würde, sehr kurze Pulse wiederholungsfrei zu erzeugen[11]. Im Laufe dieser Arbeit konnte eine Lösung für dieses Problem gefunden werden, sodass die Pulsfilter entfernt werden konnten.

Das theoretische Minimum der Pulsbreite liegt bei 8.33 ns, der Dauer eines Taktzyklus des verwendeten Prozessors bei dem maximalen Takt von 120 MHz. Als erster Test der möglichen Pulsbreiten wurde zunächst ein einfaches Testprogramm geschrieben, das einen der General Purpose Input/Output (GPIO)-Pins des Prozessors in einer Schleife abwechselnd auf High- und Low-Potenzial zieht. Die Ergebnisse sind in Tabelle 2.3 zu sehen. Es wird deutlich, dass eine rein software-basierte Lösung deutlich zu langsam ist. Als Alternative wurden die Hardware-Timer des TM4C1294NCPDT im Pulsbreitenmodulation (PWM)-Modus verwendet. Die Timer verfügen – wie ausführlich in in Abschnitt 6.7 beschrieben – über 24 bit Auflösung, daher beträgt die maximale Verzögerung bei einem Takt von 120 MHz $2^{24}/_{120\,\mathrm{MHz}} \approx 140\,\mathrm{ms}$. Es ist also nicht möglich, Pulse mit einem Abstand von mehr als 140 ms zu erzeugen. Alternativ wäre es möglich gewesen, die Timer zu verketten, auf diese Weise hätte sich eine maximale Verzögerung von 140 ms zwischen *jeweils zwei* Pulsen ergeben. Diese Implementierung hätte es aber erfordert, dass alle Pulse immer in aufsteigender Reihenfolge hätten konfiguriert werden müssen, also Pulskanal 1 immer vor Pulskanal 2 ausgelöst wird etc. Da dies eine deutliche Einschränkung bei der Verwendung des Systems ergeben hätte und die maximale Verzögerung von 140 ms aktuell ausreichend ist, wurde die erste Variante implementiert.

Die Parameter der auszusendenden Pulse werden über Sätze konfiguriert, das eigentliche Aussenden wird über ein Kommando in Satz 1 (siehe Abschnitt 2.2.3) einmalig ausgelöst. Alternativ ist es über eine sogenannte „AutoFire“-Funktion möglich, die Pulse periodisch

[9] Die Angabe von 0, also NO_ERROR, ist hier unzulässig, wenn kein Fehler auftritt, wird, wie oben beschrieben, der aktuelle Wert gesendet.

[10] Es ist geplant, zwei IMS parallel zu betreiben und möglicherweise auch noch andere Aktionen mit den Pulskanälen auszulösen, daher wurden vorsorglich sechs Kanäle vorgesehen.

[11] Es war damals fraglich, ob es bei sehr kurzen Pulslängen möglich sein würde, die Pulserzeugung zu deaktivieren, bevor ein zweiter Impuls erzeugt würde.

Variante	min. Pulslänge (ns)
Software-Schleife, ohne Optimierung	315
Software-Schleife, mit Optimierung	165
Erzeugung mit PWM-Timer	8.33
Theoretisches Optimum	8.33

Tabelle 2.3: Mit verschiedenen Techniken erreichte minimale Pulsbreiten.

automatisch auszulösen, sodass die Messaperatur unabhängig vom Hostcomputer Messdaten erzeugt. Diese Funktion wird ebenfalls über einen Satz konfiguriert.

2.2.3 Das Sätze-System

Um mit dem bestehenden System kompatibel zu bleiben, muss auch die Konfiguration dieser Karte in Sätzen gespeichert werden. Ein solcher Satz speichert neben dem eigentlichen Wert auch Metadaten, wie etwa die Einheit des Wertes. Tabelle 2.5 zeigt alle 11 Parameter eines Satzes. Alle Parameter können von außen geändert werden. Die Software identifiziert die Sätze dabei ausschließlich über ihre Nummer, die Beschreibung etc. sind nur zur Verwendung durch den Benutzer. Eine besondere Rolle nimmt dabei Satz 0 ein, er enthält eine andere Aufteilung der Parameter, die in Tabelle 2.6 dargestellt ist, und beschreibt die Karte als Ganzes. Eine Sonderstellung hat auch Satz 1 inne. Sein aktueller Wert stellt Statusinformationen der Karte dar (siehe Abschnitt 6.1.1), ein Schreibzugriff auf das Modul wird von der Software verhindert und der Wert des Schreibzugriffes als Kommando interpretiert. Tabelle 2.4 zeigt eine Übersicht aller implementierten Kommandos.

Im praktischen Einsatz wird eine Vorlage für die Metadaten der Sätze in einer getrennten Datei zusammen mit der Software einer Karte gepflegt. Nachdem die Software auf eine neue Karte überspielt wurde, werden mit einem Skript die Metadaten der Sätze aus der Datei in den Speicher der Karte übertragen. Neben Einträgen wie etwa der Beschreibung, die auf allen Karten eines Typs gleich sind, verfügen die Sätze auch noch über die Felder *Offset* und *Slope*, in denen Versatz und Steigung einer Kalibriergeraden gespeichert werden können, wobei sich der genaue Verlauf der Geraden durch Bauteilstreuungen von Karte zu Karte unterscheidet. Diese Werte werden daher nach der Übertragung der restlichen Daten durch eine Kalibrierung ermittelt.

Die Kalibriergerade beschreibt den Zusammenhang zwischen dem Wert im *Value* Parameter und dem wirklichen, messbaren Wert. So kann etwa bei einer Spannungsversorgungskarte die zu erzeugende Spannung intern digital über einen Wert von 0 bis 1024 beschrieben werden, über die Kalibriergerade wird dieser Wertebereich aber auf 0 V bis 10 V abgebildet. Die *LabView*-Blöcke des Instituts beinhalten Funktionen um diese Abbildung vorzunehmen, sodass aus Sicht des Benutzers nur die Spannung in Volt eingegeben werden muss und diese automatisch in den passenden Steuerwert für den Controller umgewandelt wird. Dabei werden auch die Minimal- und Maximalwerte aus

Wert	Konstante (CONTROL_CMD_*)	Kommando
0	NOOP	(keine Aktion)
1	FIX_EEPROM_MAGIC	„Magische“ Werte in den EEPROM schreiben
2	CLEAR_EEPROM	EEPROM vollständig löschen
3	RESET_DEVICE	Software neustarten
4	FIRE_PULSES	Pulse auslösen
5	RESET_USB_PACKAGE_HANDLING	USB-Paketzähler zurücksetzen
6	AQUIRE_SPECTRUM	Einzelnes Spektrum aufnehmen
7	AUTO_AQUIRE_ON	Automatische Aufzeichnung von Spektren an
8	AUTO_AQUIRE_OFF	Automatische Aufzeichnung von Spektren aus
9	TRIGGER_SPECTRUM_NOW	Trigger an das Spektrum-Modul senden

Tabelle 2.4: Übersicht aller Kommandos, die an Satz 1 gesendet werden können. Diese Aufstellung ist auch in der Datei *command_codes.h* zu finden.

den Parametern *Value Min* und *Value Max* beachtet[12]. Diese Umwandlung wird innerhalb von *LabView* durchgeführt, da die nötigen Fließkommaberechnungen auf den, in den meisten Karten verwendeten, ATmega-Controllern sehr aufwendig sind.

Speicherung der Sätze

Zur permanenten Speicherung der Sätze sind mehrere Ansätze denkbar. Die Speicherdauer der Metadaten sollte eine Abschaltung des Systems überstehen, der flüchtige SRAM des Controllers scheidet daher aus. Als weitere Möglichkeiten bleiben der Flash und der Electrically Erasable Programmable Read-Only Memory (EEPROM) des TM4C1294-NCPDT. Der Flash verfügt mit 1 MB über eine ausreichende Menge an Speicher [1, Abs. 1.2], die Änderung der Werte ist aber sehr aufwendig, da der Flash immer nur seitenweise beschrieben werden kann. Zudem muss darauf geachtet werden, keine Teile des Flash zu verwenden, die schon vom Programmcode benutzt werden, welcher auch im Flash abgelegt ist. Im EEPROM steht mit 6 kB Speicher für etwa 75 Sätze zu Verfügung (s.u.), was ausreichend ist. Der EEPROM lässt sich in Wörtern von 32 bit Breite beschreiben, was die Programmierung vereinfacht. Aufgrund der einfacheren Programmierbarkeit wurde daher der EEPROM zur Speicherung der Sätze verwendet.

Die Erfahrung mit anderen Karten, bei denen die Sätze auch im EEPROM abgelegt wurden, zeigt jedoch, dass diese Art von Speicher anfällig für Fehler durch starke elektrische Felder ist. So konnte beobachtet werden, dass sich die Metadaten der im EEPROM ablegten Sätze immer wieder auf zufällige Werte änderten; als Verursacher hiervon wurden die

[12] Die sich auf den Wert auf der Karte, also nach der Umrechnung, beziehen, nicht etwa auf die Eingabegröße.

Nummer	Konstante (SET_PARAM_*)	Länge (Byte)	Bedeutung
0	CATEGORY	1	Kategorie des Satzes, zur automatischen Klassifizierung in LabView verwendet
1	UNIT	10	Einheit des Wertes als ASCII-String
2	DESCRIPTION	30	Beschreibung/Name als ASCII-String
3	VALUE	2	Aktueller Wert
4	DIRECTION	1	Soll- (1) oder Ist-Wert (0)
5	OFFSET	4	Versatz der Kalibriergeraden (4 Byte Fließkommazahl)
6	SLOPE	4	Steigung der Kalibriergeraden (4 Byte Fließkommazahl)
7	VALUE_MIN	2	Minimalwert
8	VALUE_MAX	2	Maximalwert
9	VALUE_DEFAULT	2	Standardwert
10	LINK	1	Nummer eines vernüpften Satzes (0 = deaktiviert)

Tabelle 2.5: Übersicht über die Parameter eines Satzes. Alle Parameter können über Schreibzugriffe geändert werden und werden im EEPROM dauerhaft gespeichert. Ausgenommen davon ist der Parameter *Value*, sein Wert wird nur im flüchtigen RAM gespeichert und beim Start des Programms mit dem Wert aus *Value Default* initialisiert. Der Satz 0 hat andere Parameter, siehe Tabelle 2.6

Nummer	SET0_PARAM_*	Länge	Bedeutung
0	SERIAL_NO	2	Seriennummer der Karte
1	DEV_NAME	50	Name der Karte als ASCII-String
2	BLINK	1	Löst Blinken der Karte zur Identifizierung aus
3	I2C_ERROR_COUNT	1	I^2C Fehler-Zähler
4	SOFTWARE_REV	10	Software-Revision als ASCII-String
5	BUS_ERROR	1	nicht verwendet

Tabelle 2.6: Die Paramter des Satz 0. Die Software-Revision ist in der Software fest eingespeichert und kann von außen nicht geändert werden. Die Karte blinkt, wenn der entsprechende Paramter mit einem Wert ungleich Null beschrieben ist.

Felder, die durch das Schalten von Hochspannung in benachbarten Karten entstanden waren, identifiziert. Es wurde daher eine Überprüfung des EEPROM eingebaut. Dazu werden in drei, ansonsten unbenutzten, Bytes jedes Satzes zufällige, „magische“ Werte gespeichert, die beim Start des Controllers überprüft werden. Sollte eine Veränderung dieser Werte festgestellt werden, wird eine Warnung ausgegeben. Bei einer neuen Karte ist es deshalb allerdings nötig, diese Werte einmalig über ein spezielles Kommando in den EEPROM zu schreiben.

Die Werte des EEPROM können immer nur in 32 bit Worten geändert werden, eine byte-weise Änderung ist nicht ohne einen Read-Modify-Write-Zyklus möglich[13]. Die einzelnen Parameter der Sätze sind nicht auf solche Wortgrenzen ausgerichtet, wie Tabelle 2.5 zeigt, es ist daher schwierig, einzelne Werte zu ändern, wenn die Daten in genau dieser Anordnung im EEPROM ablegt werden würden. Um die Programmierung zu vereinfachen wurden daher alle Parameter auf Wortgrenzen ausgerichtet. Insbesondere wurden dabei auch alle ein Byte großen Felder zu vier Bytes aufgerundet. Diese Variante benötigt also deutlich mehr Speicher als eine kompakte Speicherung, pro Satz sind so 80 B nötig. Da die erste 64 B-Seite des EEPROM aktuell nicht genutzt wird, können auf diese Weise $(1024 * 6 - 64)/80 = 76$ Sätze gespeichert werden. Bei einer kompakten Speicherung sind pro Satz nur 59 B nötig, sodass maximal (bei Benutzung des gesamten EEPROMs) $1024 * 6/59 \approx 104$ Sätze gespeichert werden könnten. Da aktuell weniger als 50 Sätze benötigt werden, wurde der einfachere Ansatz zu Speicherung gewählt. Gleichzeitig können dadurch im letzten Parameter, der ebenfalls von ein auf vier Byte erweitert wird, die drei „magischen“ Bytes zur Fehlererkennung gespeichert werden.

[13] Trotzdem wird der EEPROM byte-weise adressiert, die unteren beiden Bits der Adresse werden dabei ignoriert [5, Abs. 9.2.3.14].

2.2.4 Die Erfassung der Spektren

Eine der wichtigsten Funktionen der Datenerfassungskarte ist die Aufnahme von Spektren. Als Spektrum wird hier der zeitliche Verlauf eines analogen Signals bezeichnet, das in zeitlich äquidistanten Punkten abgetastet wird. In der Anwendung der IMS handelt es sich bei dem analogen Signal um die Spannung an einer Detektorelektrode, an der beim Auftreffen von Ionen Spannungsspitzen entstehen. Da die Ionen je nach Mobilität unterschiedlich lange für den Weg zwischen Quelle und Detektor brauchen, lässt sich aus dem so aufgenommenen Spektrum auf die Zusammensetzung des untersuchten Stoffgemisches schließen. Auf der Karte sind zur Erfassung dieser Spektren zwei ADCs des Typs ADS1675 verbaut, die jeweils über 24 bit Auflösung verfügen. Die Spektren sollen mit einer Abtastrate von 1 MSa/s erfasst werden, sodass sich bei einer Erfassungsdauer von 15 ms pro Spektrum $1\,\mathrm{MSa/s} \cdot 24\,\mathrm{bit} \cdot 15\,\mathrm{ms} = 360\,\mathrm{kbit} = 45\,\mathrm{kB}$ an Daten ergeben.

Als weitere Besonderheit soll die Karte über eine Möglichkeit verfügen, die erfassten Spektren aufzusummieren, bevor sie per USB an den Computer übertragen werden. Auf diese Weise kann die nötige Übertragungsrate auf dem USB reduziert werden[14]. Es werden von der Karte 32 bit pro Messpunkt gespeichert. Da ein Messpunkt über 24 bit verfügt, können also maximal 256 Spektren aufaddiert werden, bevor es zu einem Überlauf kommen kann. Zur Realisierung ist es nötig, zwei Spektren zu speichern: Eines, dass den aktuellen Zwischenstand der Summierung darstellt und eines, das gerade erfasst wird bzw. wurde. Da die Übertragung per USB auch einige Zeit in Anspruch nimmt und auch von der Auslastung des Host-Computers abhängig ist, wird zudem noch ein dritter Puffer verwendet, um weiter Daten erfassen zu können, während Daten per USB übertragen werden. Erfasste Spektren werden zunächst im Erfassungspuffer gespeichert, um dann mit dem Summierungspuffer verrechnet zu werden. Wenn ausreichend Spektren summiert wurden, werden Summierungs- und Übertragungspuffer getauscht und der neue Übertragungspuffer per USB gesendet. Da aus technischen Gründen auch für den Erfassungspuffer vier Byte pro Datenpunkt benötigt werden (statt der eigentlich nötigen $24\,\mathrm{bit} = 3\,\mathrm{B}$), ist insgesamt Speicher in der Größe von $3 \cdot 1\,\mathrm{MHz} \cdot 4\,\mathrm{B} \cdot 15\,\mathrm{ms} = 180\,\mathrm{kB}$ nötig. Der TM4C1294NCPDT verfügt über 256 kB internen SRAM[15] [1, Abs. 1.2], sodass nur die Erfassung eines Spektrums von *einem* der beiden ADCs möglich ist. Die Abtastung beider ADCs würde in dieser Konfiguration zu viel Speicher verbrauchen und konnte daher nicht implementiert werden.

Bei der Übertragung der Spektren per USB werden die Spektren(-pakete) mit einem Header versehen, dessen Layout in Tabelle 2.7 zu sehen ist. Einige der Daten sind für diese Implementation konstant, es sollte aber Kompatibilität zu eventuellen, zukünftigen Spektrenerfassungssystemen gewahrt werden. Der Zeitstempel im (summierten) Spektrum entspricht dem Zeitpunkt des ersten Spektrums in der Summierung aus Sicht des Controllers. Sollte es also bei der Übertragung zum Host-Computer zu Verzögerungen kommen, werden diese Zeitstempel davon nicht beeinträchtigt. Ebenso kann der Host anhand der Zeitstempel ermitteln, ob Spektrenpakete bei der Übertragung verloren gegangen sind,

[14] Da die Karte über eine High-Speed USB-Verbindung verfügt, ist die verfügbare Datenrate eigentlich nicht problematisch, auf diese Weise kann aber die Belastung des Computer reduziert werden.

[15] Von dem ein kleiner Teil auch noch von dem laufenden Programm benötigt wird

Offset	Länge (Byte)	Bedeutung
0	4	Länge des gesamten Spektren-Paketes, inkl. des kompletten Headers
4	8	Timestamp des ersten Spektrums in diesem Paket
12	1	Anzahl der Summierungen in diesem Paket
13	1	Bytes pro Datenpunkt
14	2	Abtastrate dieses Spektrums, in kHz
16	n	Spektrum-Daten

Tabelle 2.7: Übersicht über das Format eines Spektrums bei der Übertragung per USB. Der Timestamp entspricht dem Format, wie es von `SystemGetTimestamp` geliefert wird (s. Abschnitt 6.1)

da die Zeitstempel bei einer fehlerfreien Übertragung äquidistant sind. Die Übertragung der Spektren erfolgt dabei über einen, von der restlichen Kommunikation unabhängigen Kanal.

Die Erfassung eines Spektrums wird durch einen Trigger ausgelöst, der nach einer konfigurierbaren Verzögerung durch das Puls-Modul ausgelöst wird[16]. Auf diese Weise ist sichergestellt, dass die relative Position zwischen Spektrum und Pulsen immer gleich bleibt. Ohne diese Synchronisation wäre eine Auswertung der Spektren nicht möglich. Damit beim Auslösen des Triggers ein Spektrum erfasst wird, muss vorher über ein Kommando in Satz 1 ein Spektrum angefordert worden sein, alternativ ist es möglich über eine „AutoAquire“-Funktion bei jedem Trigger ein Spektrum aufzuzeichnen. Diese Funktion wird ebenfalls über Kommandos in Satz 1 aktiviert. Zusammen mit der *AutoFire*-Funktion des Puls-Moduls, ist es so möglich, kontinuierlich Spektren auszuzeichnen. ohne dass die Gefahr besteht, das die Spektren einen unterschiedlichen zeitlichen Abstand aufweisen, weil der Auslöser vom Host-Computer nicht rechtzeitig gesendet wurde[17].

2.2.5 Die Übertragung per USB

Die wichtigste Schnittstelle des Datenerfassungsmoduls zur Außenwelt stellt die USB-Schnittstelle dar. Der TM4C1294NCPDT verfügt über einen integrierten USB-Controller, der allerdings maximal Übertragungen nach dem Full-Speed-Standard mit maximal 12 Mbit/s unterstüzt [21, 1, Abs. 1.3.5.3]. Da diese Datenrate nicht für die kontinuierliche Übertragung von Spektren ausreicht, ist auf dem Datenerfassungsmodul ein externer USB-Baustein USB3300 von Microchip untergebracht, mit dessen Hilfe eine Übertragung nach dem High-Speed-Standard mit bis zu 480 Mbit/s möglich ist.

Das zur Kommunikation mit der Karte verwendete Transportprotokoll wurde von der USB-Messkarte übernommen, um die *LabView*-Blöcke zur Generierung der Daten weiter verwenden zu können. Im Protokoll beginnt jede Anfrage mit einem Byte, in dem der

[16] Zu Testzwecken kann aber auch manuell über ein Kommando in Satz 1 ein Triggersignal generiert werden

[17] Dies ist aktuell bei der USB-Messkarte ein Problem

Typ der Anfrage angeben ist. Wie Tabelle 2.9 zeigt, sind aktuell nur zwei Kommandos implementiert. Die USB-Messkarte verfügt noch über ein drittes Kommando um die Aufzeichnung von Spektren zu starten, dieses wird vom Datenerfassungsmodul nicht unterstützt, eine Aufzeichnung der Spektren wird stattdessen durch ein Kommando in Satz 1 aktiviert oder geschieht automatisch (siehe Abschnitt 2.2.4). Der genaue Aufbau des Protokolls ist in Tabelle 2.8 zu sehen. Es ist darauf zu achten, dass die Länge der Übertragung von der Art des Kommandos abhängt.

Nach Bearbeitung einer Anfrage wird eine Antwort auf Endpoint 1 an den Host gesendet. Das Format der Antwort ist abhängig von der Anfrage: Wenn die Anfrage nicht verstanden wurde, wird ein einzelnes Byte mit dem Wert *3* gesendet, dem Fehlercode für „not supported“, wie Tabelle 2.2 entnommen werden kann. Die Anfrage nach der I^2C-Adresse der Karte wird mit einem einzelnen Byte mit ebendieser Adresse beantwortet. Wenn eine *BusControl*-Transaktion durchgeführt werden sollte, wird in der Antwort zunächst der Fehlercode der Transaktion gemäß Tabelle 2.2 gesendet, und danach, wenn kein Fehler aufgetreten ist[18], das vollständige empfangende *BusControl*-Paket, wie in Abschnitt 2.2.1 beschrieben und in Tabelle 2.1 dargestellt. Die Antwort beginnt demnach immer mit einem Fehlercode gemäß Tabelle 2.2, es sei denn, es wurde nach der I^2C-Adresse gefragt. Im Normalbetrieb wird meistens zunächst die I^2C-Adresse der Karte abgefragt, um dann weiter über die Sätze mit der Karte zu kommunizieren.

Das USB-Transportprotokoll wird von der Treibersoftware noch ein Mal in ein weiteres Protokoll „eingepackt“, dies ist vor allem nötig, um das Ende einer Anfrage zu erkennen, da nicht mit der Verarbeitung begonnen werden darf, bevor die Anfrage vollständig übertragen wurde. Dieses Protokoll ist aus Sicht von *LabView* und den höheren Schichten der Mikocontroller-Software völlig transparent. In der Implementierung der USB-Messkarte wurde hier eine vereinfachte Form des USB-TMC-Protokolls ([22]) verwendet. Dieses Protokoll ist recht umfangreich und bietet viele Optionen, die in dieser Konstellation nicht benötigt werden. Es wurde daher ein anderes, einfacheres Protokoll implementiert[19].

Der einzige Zweck, dem dieses äußere Protokoll dient, ist, das Ende eine Anfrage zu kennzeichnen. Es würde sich daher anbieten, einige Bytes mit der Länge am Anfang zu senden, wodurch dann das Ende der Übertragung erkannt werden kann, ein Verfahren, wie es beispielsweise auch im *BusControl*-Protokoll verwendet wird. Der Nachteil dieser Vorgehensweise – auch verallgemeinernd als In-Band-Signalling bezeichnet [23] –, ist, nach einem Übertragungsfehler den Zustand von Sender und Empfänger wieder zu synchronisieren, da es nicht mehr eindeutig ist, ob es sich bei einer Übertragung um Längen- oder Datenbytes handelt.[20]. Eine Lösung für dieses Problem bietet allgemein das Out-of-Band-Signalling, bei dem Steuerinformationen getrennt von den eigentlichen

[18] Die USB-Messkarte sendet teilweise auch im Fehlerfall noch Daten, diesen fehlt aber – je nach Fehler – teilweise die Checksumme und es sind auch nicht immer die mit dieser Transaktion assoziierten Daten, daher sollten sie ignoriert werden.

[19] Die ursprüngliche Implementierung ist aber noch in der Treibersoftware enthalten, da sie zur Kommunikation mit der USB-Messkarte, die zusätzlich weiterhin unterstützt wird, nötig ist.

[20] Einige Berühmtheit hat im Institut in diesem Zusammenhang ein unüberlegt an eine Karte gesendetes „Q“ erlangt, dessen ASCII-Code als Länge interpretiert wurde und die betreffende Karte recht wirkungsvoll außer Gefecht gesetzt hat.

Offset	Länge (Bytes)	Bedeutung
0	1	Kommando
1	1	I^2C-Adresse des Ziels
2	1	(reserviert)
3	1	Größe des nachfolgenden Paketes
4	n	*BusControl*-Paket

Tabelle 2.8: Das USB-Transportprotokoll. Bei dem ersten Byte handelt es sich immer um ein Kommando aus Tabelle 2.9, der nachfolgende Teil ist nur bei dem Kommando 7 vorhanden.

Code	Konstante (USB_CMD_*)	Bedeutung
2	I2C_READ_WRITE	*BusControl*-Transaktion durchführen
7	GET_ADDRESS	Aktuelle I^2C-Adresse des Datenerfassungsmoduls abrufen

Tabelle 2.9: Unterstützte Kommandos des USB-Transportprotokolls. Alle Kommandos sind auch in der Datei *command_codes.h* zu finden.

Daten übertragen werden. Bei einer USB-Übertragung kann hier die Länge der Pakete benutzt werden. Im Normalbetrieb überträgt der Sender immer Pakete, deren Länge der maximalen Paketgröße des Empfängers entspricht. Wenn der Empfänger ein Paket erhält, das kleiner ist, kann er davon ausgehen, dass die Übertragung beendet ist. Ein Problem ergibt sich nur, wenn die Gesamtlänge der Übertragung ein Vielfaches der maximalen Paketgröße ist, da dann das letzte Paket auch vollständig gefüllt ist. In diesem Fall sendet der Sender nach der Übertragung noch ein Zero-Length-Package (ZLP), um das Ende der Übertragung zu signalisieren. Diese Art von Übertragungsprotokoll ist weit verbreitet, daher ist eine Unterstützung dafür in vielen Softwarebibliotheken enthalten [12, 6, Abs. 5.5.6.12].

Neben dem oben beschriebenen Endpoint 1 verfügt das Datenerfassungsmodul über einen Endpoint 2, der nur in IN-Richtung aktiv ist, also nur Daten zum Host hin übertragen kann. Über diesen Endpoint werden die (aufsummierten) Spektren übertragen. Auf diesem Endpoint wird das Ende eines Datenpaketes nicht durch ein nicht komplett gefülltes Paket angezeigt, da sich dies im Zusammenspiel mit der asynchronen Übertragung in der Treibersoftware nicht zuverlässig erkennen ließ. Da aber die Spektren schon eine Längenangabe in den ersten Bytes enthalten (siehe Tabelle 2.7) stellt dies kein Problem dar.

3 Grundlagen zur Programmierung des TM4C1294NCPDT

3.1 Entwicklungsumgebung

Der verwendete Mikrocontroller, ein TM4C1294NCPDT des Herstellers Texas Instruments, basiert auf einem Cortex-M4F-Kern von ARM, der mit zahlreichen Peripherie-Modulen erweitert wurde [1, Abs. 1.3]. Zur Programmierung kann ein beliebiger Compiler verwendet werden, der entsprechenden Maschinencode für den ARM-Kern erzeugen kann. Im Rahmen dieser Arbeit wurde das *Code Composer Studio* [3] des Herstellers Texas Instruments verwendet. Es handelt sich um eine Integrierte Entwicklungsumgebung (IDE) auf Basis von Eclipse, die auf die Mikrocontroller der Firma abgestimmt ist.

Zur Übertragung des fertigen Programms, sowie zum Debuggen des Programms während der Ausführung ist ein JTAG-Emulator erforderlich, ein Gerät, welches eine Verbindung zwischen Computer und JTAG-Schnittstelle des Mikrocontrollers herstellt. JTAG-Emulatoren sind von verschiedenen Herstellern erhältlich, Texas Instruments selbst bietet jedoch sogenannte *Evaluation Boards* an, Platinen, auf denen ein einfacher JTAG-Emulator und ein Mikrocontroller kombiniert sind. Ein großer Teil der Entwicklung wurde mit einem solchen Evaluation-Board durchgeführt, dem Modell *EK-TM4C1294XL*, das einen TM4C1294NCPDT enthält. Der große Vorteil der JTAG-Emulatoren auf den Evaluation-Boards ist, dass diese zusammen mit der kostenlosen Version des Code Composer Studios verwendet werden können; bei der Verwendung eines anderen JTAG-Emulators ist eine kostenpflichtige Version nötig.

Durch das Auftrennen einiger Verbindungen auf dem Evaluation-Board kann die Verbindung des JTAG-Emulators zu dem enthaltenen Mikrocontroller gelöst werden und der JTAG-Emulator mit einem anderen Controller verbunden werden. Das Board kann so als eigenständiger JTAG-Emulator verwendet werden. Diese Funktion wurde später, als die fertige Platine zur Verfügung stand, genutzt.

3.2 Programmierung

Eine wichtige Funktion der Software ist die Ansteuerung der zahlreichen Peripherie-Module des Controllers. Die Konfiguration aller Peripherie-Module wird über spezielle Register des Controllers durchgeführt, welche über spezielle Speicheradressen in den Speicherbereich des Controllers eingeblendet werden. Ein Zugriff auf ein Register eines Peripherie-Moduls unterscheidet sich also nicht von dem Zugriff auf eine andere Adresse

des Speichers. Funktion und Speicheradressen aller Register sind im Datenblatt des Controllers ausgiebig dokumentiert [1].

Viele Module sind mehrfach vorhanden, so gibt es etwa acht identische Timer-Module. Jedes dieser Module verfügt über einen identischen Satz an Registern. Im Datenblatt sind daher sowohl Start-Adressen der Module als auch die Offsets der einzelnen Register zu diesen Start-Adressen vermerkt. Die absolute Adresse eines Register ergibt sich dann aus der Addition beider Werte.

Soll beispielsweise die „Stall"-Funktion des Timers A des dritten Timerblocks aktiviert werden – unabhängig davon, was diese Funktion genau tut –, so ist dem Datenblatt zu entnehmen, dass dazu das Bit mit der Nummer 1 in dem Register *GPTMCTL* des entsprechenden Timerblocks gesetzt werden muss. Dieses Register hat das Offset 0x00C, die Start-Adresse des dritten Timerblocks ist 0x40032000. Es muss also das Bit 1 an der Speicheradresse 0x4003200C gesetzt werden.

3.2.1 Tiva Driverlib

Diese Art der Programmierung ist sehr aufwendig und fehleranfällig. Vom Hersteller Texas Instruments wurde daher eine Sammlung von Bibliotheken und Beispielcode für den Mikrocontroller unter dem Namen *TivaWare* veröffentlicht [4], welche die Programmierung sehr vereinfachen. Bestandteil dieser Code-Sammlung sind unter anderem die Bibliotheken *driverlib* und *usblib*. Die in dieser Arbeit erstellte Software macht von beiden Bibliotheken regen Gebrauch. Mit Hilfe der *driverlib* vereinfacht sich etwa der Code, um die oben genannte „Stall"-Funktion zu aktivieren auf den Aufruf von `TimerControlStall(TIMER2_BASE, TIMER_A, true);`. Der erste Parameter gibt den zu verwendenden Timer-Block an, danach wird angegeben, dass die Funktion für den Sub-Timer A aktiviert werden soll. Der letzte Parameter (`true`) legt schließlich fest, dass die Funktion aktiviert, und nicht deaktiviert werden soll.

Der Code der Bibliothek ist sehr schlank und besteht größtenteils aus Bitshifts und -maskierungen[1], erhöht aber die Lesbarkeit des Codes deutlich. Nachteil der Bibliothek ist ein kleiner Geschwindigkeitsnachteil, und die Tatsache, dass nicht alle möglichen Sonderfälle und -möglichkeiten der Hardware von der *driverlib* als Funktion abgebildet werden. Dies war laut der Dokumentation eine bewusste Design-Entscheidung beim Entwurf der Bibliothek [5, Abs. 1]. Zudem bildet die Bibliothek nur eine dünne Schicht über den Registern und bietet kaum Fehlerüberprüfung. Auch wenn alle Parameter auf plausible Werte hin geprüft werden, wird bei weitem nicht jede unsinnige Konfiguration der Hardware unterbunden. Trotz dieser Einschränkungen bietet die *driverlib* einen enormen Komfortgewinn bei der Programmierung und wird daher im vorliegenden Programm intensiv genutzt. Zum Verständnis der einzelnen Programmteile wird daher nicht nur die Lektüre des Datenblattes, sondern auch die Dokumentation der *driverlib* empfohlen[2].

[1] So ist etwa in dem vorhergehenden Beispiel die Konstante `TIMER2_BASE` als der Wert der Basisadresse des dritten Timerblocks gewählt, sodass eine Fallunterscheidung zur Laufzeit vermieden werden kann.

[2] Sowie die Lektüre der Errata-Liste [2], die bekannte Fehler des TM4C1294NCPDT enthält. Leider betreffen einige davon auch das vorliegende Programm.

Da die *driverlib* vermutlich in den meisten Programmen, die für die entsprechenden Mikrocontroller entwickelt werden, benutzt wird, bildet es eine gewisse Menge an Programmcode, der in identischer Form in jedem Programm enthalten ist. Texas Instruments liefert daher den TM4C1294NCPDT mit einer reduzierten Version der *driverlib* fest eingespeichert im ROM des Mikrocontrollers aus. In den Dateien des *TivaWare*-Pakets sind auch die Dateien *rom.h* und *rom_ map.h* enthalten. Die erste enthält Makros der Form `ROM_FunktionsName`, die die Version von `FunktionsName` aus dem ROM aufruft. Da nicht jede Funktion im ROM enthalten ist, definierte die zweite Datei Makros der Form `MAP_FunktionsName`, die die entsprechende Funktion im ROM aufruft, wenn sie dort vorhanden ist, und ansonsten auf die normale Version zurückfällt, die dann im Flash gespeichert wird. Im vorliegenden Programm wird diese Art der Aufrufe durchgängig benutzt[3] um Speicher im Flash zu sparen.

3.2.2 Tiva USBLib

Für die Programmierung des im TM4C1294NCPDT integrierten USB-Controllers stellt die Verwendung der *driverlib* die einzige Möglichkeit dar, da die Dokumentation der einzelnen Register nicht in der öffentlich zugänglichen Version des Datenblatts enthalten ist. Die vollständige Version des Datenblatts ist nur gegen die Unterzeichnung einer Verschwiegenheitserklärung (NDA) erhältlich [1, Abs. 21].

Wie im Kapitel 4 beschrieben, ist die USB-Spezifikation sehr umfangreich, sodass die Implementierung eines der Spezifikation entsprechenden Endgerätes mit erheblichen Aufwand verbunden ist. Von Texas Instruments ist neben der oben beschriebenen *driverlib* auch eine Bibliothek namens *usblib* verfügbar, die den größten Teil der nötigen Implementierung enthält. Die Bibliothek besteht aus drei Schichten, die über der *driverlib* angesiedelt sind, wie Abbildung 3.1 zeigt. In der Dokumentation sind drei typische Anwendungsfälle gekennzeichnet, für die sich die *usblib* eignet. Der erste Anwendungsfall, in der Abbildung als „Anwendung 1“ bezeichnet, stellt die Entwicklung eines eigenen USB-Stacks dar. Bei dieser Anwendung ist die Benutzung der *usblib* nicht sinnvoll, so dass direkt auf die *driverlib* zugegriffen wird. Anwendungsfall 2 stellt die Entwicklung eines USB-Gerätes dar, das in keine der vordefinierten Klassen der höheren Ebenen fällt. Hierbei kann auf einige Funktionen der *usblib* zugegriffen werden, es ist aber weiterhin nötig, auf Funktionen der *driverlib* zuzugreifen.

Die *usblib* enthält für einige der vom USB-Konsortium definierten Geräteklassen, wie etwa Human Interface Device (HID)-Geräte oder Massenspeicher, weitere Funktionen, die die Implementierung von solchen Geräten vereinfachen. Zusätzlich sind in der *usblib* Funktionen enthalten, die den Umgang mit der paketorientierten USB-Schnittstelle auf eine bytestromorientierte abbilden, um den Umgang damit zu einfacher zu gestalten. Diese werden als „USB Buffer“ bezeichnet. In der Dokumentation wird als Anwendungsfall 3 ein Gerät beschrieben, das in eine dieser Geräteklassen fällt und daher die entsprechenden Funktionen, zusammen mit den USB-Buffern, nutzen kann, wie in Abbildung 3.1 zu sehen.

[3] Abgesehen von einigen Fällen bei denen die Version im ROM laut dem Errata Fehler enthält.

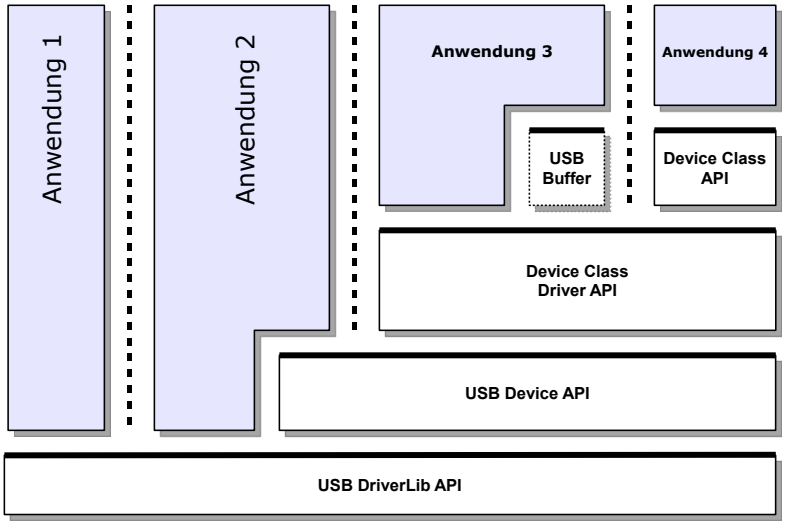


Abbildung 3.1: Die einzelnen Schichten der *usblib*. Die unterste Ebene stellt die *driverlib* dar. Quelle: [6]

Für einige Klassen, wie etwa die HID-Klasse, sind in der *usblib* noch weitere Funktionen enthalten, welche die Entwicklung von spezifischeren Gerätegruppen innerhalb dieser Klasse unterstützen, in diesem Fall etwa Mäuse oder Tastaturen. Eine Anwendung, die diese oberste Schicht der *usblib* nutzt, fällt in Anwendungsfall 4.

Schlussendlich enthält die *usblib* auch noch Funktionen um als USB-Host oder als USB-On-The-Go-Gerät zu fungieren, diese werden hier aber nicht weiter beachtet.

Unter den vom USB-Konsortium vordefinierten Klassen gibt es keine, in die das Datenerfassungsmodul fallen würde; es handelt sich um ein sogenanntes „vendor specific device“ [24, Abs. 7.1]. In der *usblib* ist mit dem „bulk device“ dennoch eine Geräteklasse vorhanden, die eine hohe Übereinstimmung mit dem Anforderungsprofil des Datenerfassungsmoduls zeigt. Diese Klasse ist laut der Dokumentation für Anwendungen geeignet, die mit einer Host-Anwendung kommunizieren müssen, welche zusammen mit dem Gerät entwickelt wird, und nur einen bidirektionalen Datenkanal benötigen. Dennoch ist die vordefinierte Klasse nicht flexibel genug, um alle Anforderungen zu erfüllen, so ist etwa neben dem bidirektionalen Kanal zu Kommunikation noch ein unidirektionaler Kanal zur Übertragung der Spektren nötig. Zudem ist für die automatische Treiberinstallation mittels WCID, in Abschnitt 5.2.2 beschrieben, das Beantworten bestimmter USB-Requests nötig, was mit der vorliegenden Implementierung nicht möglich ist. Es wurde daher eine Anwendung nach Anwendungsfall 3 entwickelt, es kommt dabei aber keine vordefinierte, sondern eine selbst entwickelte Geräteklasse zum Einsatz. Der Programmcode der Geräteklasse wurde stark an die Implementierung der „bulk device“-Geräteklasse angelehnt.

4 Grundlagen des USB

Im Folgenden soll ein Überblick über den Universal Serial Bus, kurz USB gegeben werden. Der Fokus dieses Kapitels liegt auf der Vermittlung des nötigen Wissens zum Verständnis des vorliegenden Codes. Für eine weitergehende Einführung in USB wird [24, 25] empfohlen, auf denen auch dieses Kapitel basiert.

4.1 Geschichtlicher Überblick

USB wurde von einem Industriekonsortium, als USB Implementers Forum (USB IF) bezeichnet, maßgeblich unter der Leitung von Intel entwickelt und 1996 veröffentlicht [26]. Ziel der Entwicklung war eine einfach zu benutzende, einheitliche, flexible und kostengünstige Schnittstelle, welche die bis dahin bestehenden, zahlreichen untereinander inkompatiblen Lösungen ablösen sollte [24]. Neben den Vorteilen aus Anwendersicht, wie etwa die Fähigkeit des Hot-Pluggings, bietet USB auch aus Entwicklersicht Vorteile, so ist es etwa nicht mehr nötig, die Interrupt-Kanäle (IRQs) und Speicheradressen der Peripheriegeräte eines PCs manuell zu verwalten, was eine deutliche Erleichterung darstellt. Nachteilig ist aber zu sehen, dass es deutlich mehr Arbeit erfordert, ein USB-kompatibles Gerät zu entwickeln, als eines, das etwa die serielle Schnittstelle (COM) nutzt[1]. Mit der weiten Verbreitung von USB über verschiedenste Anwendungsgebiete hinweg kann die Entwicklung von USB jedoch als Erfolg bezeichnet werden.

Im Jahr 2000 wurde die zweite Version der Spezifikation, USB 2.0, vorgestellt. Durch die Einführung einer neuen Geschwindigkeitsklasse (High-Speed), welche brutto Geschwindigkeiten von 480 Mbit/s bietet, wurde das Anwendungsfeld auch auf Festplatten, etc. erweitert, die vorher durch die maximale Datentransferrate von 12 Mbit/s stark limitiert wurden. Zusätzlich wurden neue Verbindungsmöglichkeiten, wie etwa USB-On-The-Go eingeführt, bei denen ein Gerät wechselweise als Host oder Device arbeiten kann. Diese sollen aber, genauso wie die später mit der Vorstellung von USB 3.0 und USB 3.1 eingeführten Erweiterungen, hier nicht weiter betrachtet werden.

4.2 Die Hierachie der Descriptoren

Ein wichtiges Design-Ziel bei der Entwicklung von USB war eine Erkennung von Geräten durch den Computer ohne manuelle Interaktion durch den Anwender, inklusive der automatischen Auswahl eines passenden Treibers, sofern möglich [24]. Um diese

[1] Aus diesem Grund konnte sich USB auch z.B. im Hobby-Bereich bei selbst entwickelten Geräten noch nicht sehr weit durchsetzen.

Funktion zu ermöglichen, ist es nötig, dass der Computer – im Weiteren auch als „Host“ bezeichnet – nach dem Verbinden des Gerätes Informationen über das neue Gerät in einem standardisierten Format abfragen kann. Diese Funktion wird von den *Deskriptoren* erfüllt. Jedes Gerät verfügt über genau einen *Device Descriptor*, in dem grundlegende Informationen zu dem gesamten Gerät angegeben sind, wie etwa der Hersteller und die Anzahl der möglichen Konfigurationen.

Jedes Gerät kann über mehrere solcher Konfigurationen verfügen, von denen aber immer nur eine gleichzeitig aktiv sein kann. Die aktive Konfiguration wird vom Host ausgewählt. Jede mögliche Konfiguration wird in einem *Configuration Descriptor* beschrieben. Dort sind auch Angaben zum Stromverbrauch zu finden; so ist es etwa vorgesehen, dass der Host die zur Verfügung stehende Strommenge unter den Geräten aufteilt, in dem er sie in Konfigurationen mit passendem Stromverbrauch schaltet[2]. Das hier vorgestellte Gerät verfügt nur über eine einzige Konfiguration.

Für jede Konfiguration kann das Gerät eine bestimmte Menge an *Interfaces* bereitstellen, welche die verschiedenen Funktionen eines Gerätes darstellen, etwa Druck- und Scanfunktion eines Multifunktionsgerätes. Jedes Interface wird durch einen *Interface Descriptor* beschrieben und besteht aus einer Menge von *Endpoints*. Diese Endpoints stellen – abgesehen von dem speziellen, zu keinem Interface gehörenden und nur zur Konfiguration benutzten Endpoint 0 – unidirektionale Kommunikationskanäle dar und werden durch einen *Endpoint Descriptor* beschrieben. Die Datenerfassungskarte verfügt nur über ein Interface, dieses besteht aus drei Endpoints. Endpoint 1 (EP1) wird mit beiden Datenflussrichtungen, IN und OUT definiert, was zwei unabhängige Endpoints darstellt. Hierdurch entsteht ein bidirektionalen Kommunikationskanal, über den die Kommunikation erfolgt. Zusätzlich ist EP2 in IN-Richtung definiert, über den die erfassten Spektren übertragen werden. Die Richtungsangaben erfolgen immer aus Sicht des Hosts.

Jeder Endpoint kann einer von vier Transferarten zugeordnet werden:

Control Spezielle Transferart zur Konfiguration, nur für EP0 zulässig.

Bulk Geeignet zur Übertragung großer Datenmengen. Es gibt keine Garantien zu Latenz oder Datenrate; die Daten werden mit Fehlerkorrektur übertragen.

Interrupt Geeignet zur Übertragung kleiner Datenmenge mit garantierten Abfragezyklus[3].

Isochronous Geeignet für Übertragungen mit hohen Anforderungen an die maximale Latenz (Streaming). Die Übertragung erfolgt mit reservierter Bandbreite und garantierter Latenz, aber ohne Fehlerkorrektur.

Alle drei Endpoints der Datenerfassungskarte sind Bulk-Endpoints. Für die Übertragung der Spektren wurde ein Isochronous-Endpoint in Erwägung gezogen, da es sich bei der Übertragung der Spektren in gewisser Weise um Streaming von Daten handelt. Diese

[2] Also etwa beim Anschluss eines weiteren Gerätes mit hohem Stromverbrauch ein anderes Gerät in eine stromsparendere Konfiguration schaltet.

[3] USB unterstützt ausschließlich Transaktionen, die durch den Host initiiert werden. Ein echter Interrupt durch ein Device ist daher nicht möglich und wird durch Polling simuliert.

Überlegung wurde aber Aufgrund der fehlenden Fehlerkorrektur bei der Übertragung wieder verworfen[4]. Da der Bus neben der Übertragung der Spektren kaum belastet wird, ist aber auch bei der Verwendung von einem Bulk-Endpoint mit einer zeitnahen Übertragung der Daten zu rechnen. In Versuchen konnten keine Probleme in dieser Hinsicht ermittelt werden.

Neben den schon beschriebenen Deskriptorarten sind noch die *String Descriptors* definiert. In diesen können beliebige Texte abgelegt werden, die durch den Host abgerufen werden können. Andere Deskriptoren, etwa der Device-Deskriptor, enthalten Felder, in denen auf einen String-Deskriptor verwiesen wird. In diesem Deskriptor kann dann beispielsweise der anzuzeigende Name des Gerätes als Text abgelegt werden. USB unterstützt eine Lokalisierung von Deskriptoren; der String-Deskriptor mit der Nummer 0 enthält dazu eine Liste aller unterstützten Sprachen, aus denen der Host – etwa unter Einbeziehung der aktuell eingestellten Sprache des Benutzers – bei der Abfrage anderer String-Deskriptoren eine auswählt, um die entsprechend lokalisierte Version abzurufen. Die String-Deskriptoren werden auch für die in Abschnitt 5.2.2 beschriebene, erweiterte Erkennung von USB-Geräten durch Windows (WCID) benutzt. Die Texte werden im Unicode-Format gespeichert und übertragen, was in den meisten Fällen aufgrund der Little-Endian Bytereihenfolge das Anhängen eines Null-Bytes an den ASCII-Code des Zeichens erfordert [24].

Eine sehr detaillierte Beschreibung sämtlicher Deskriptoren und die Kodierung aller enthaltener Felder ist in [24] zu finden. Im vorliegenden Programmcode sind die Deskriptoren in der Datei *device_ internal.c* zu finden. Es soll an dieser Stelle noch darauf hingewiesen werden, dass die Angabe der Version 2.0 der USB-Spezifikation im Device-Deskriptor *nicht* impliziert, dass das Gerät im High-Speed-Modus betrieben wird, sie gibt nur an, nach welchem Standard die Informationen in den Deskriptoren zu interpretieren sind. In dem Code der *usblib* von Texas Instruments wird in einem Kommentar das Gegenteil behauptet und dabei auf den Abschnitt 9.2.6.6 der USB 2.0 Spezifikation [21] verwiesen. Diesem Abschnitt ist aber nur zu entnehmen, dass ein Gerät, welches den Full-Speed-Modus unterstützt, mindestens Kompatibilität mit der Version 2.0 des Standards angeben muss. Die Implikation in die andere Richtung kann dem Text nicht direkt entnommen werden[5]. Tests haben gezeigt, dass ein Betrieb ausschließlich im Full-Speed-Modus problemlos möglich ist. Die Angabe der Version 2.0 im Device-Deskriptor ist zudem Bedingung für die erweiterte Erkennung von USB-Geräten durch Windows (WCID), sodass eine Angabe der Version 1.1 hier nicht möglich gewesen wäre.

Im Device-Deskriptor eines Gerätes ist unter anderem eine Vendor- und Product-ID angegeben, die zur eindeutigen Identifikation eines Gerätes, sowie zur Auswahl eines passenden Treibers dient. Die Zuteilung einer gültigen Vendor-ID durch das USB IF ist kostenpflichtig und für Prototypen und Kleinserien wie die vorliegende Datenerfassungskarte sehr aufwendig. Im vorliegenden Fall wurde die VID 0x1CBE verwendet. Diese ist

[4] Ein weiteres Problem hätte sich durch den verwendeten WinUSB-Treiber (siehe Abschnitt 5.2.1) ergeben, der Isochronous-Übertragungen erst seit Windows 8.1 unterstützt.

[5] Allerdings ist der Text nicht ganz eindeutig, da die Aussage „This indicates that such devices support the other_speed requests defined by USB 2.0.“ so verstanden werden kann, dass ein Betrieb im Full-Speed-Modus möglich sein muss.

laut der offiziellen Liste des USB IF nicht vergeben [27], in der inoffiziellen, erweiterten Liste des *Linux USB Projects* wird diese ID einer Firma namens *Luminary Micro Inc.*, die inzwischen von Texas Instruments aufgekauft wurde, zugeordnet. Diese Vendor-ID wird von Texas Instruments auch für die Entwicklungsboards der Launchpad-Serie, wie dem, auf dem diese Software entwickelt wurde, verwendet. Die Product-ID kann von einem Hersteller, der über eine gültige Vendor-ID verfügt, frei vergeben werden. In diesem Fall wurde die ID 0x0102 verwendet[6].

4.3 Externe USB-Bausteine

Die Kommunikation per USB 2.0 erfolgt mit einem Takt von 480 MHz. Der TM4C1294-NCPDT kann maximal mit einer Taktfrequenz von 120 MHz betrieben werden [1], daher ist eine Kommunikation per USB 2.0 technisch ohne weitere Bauteile nicht möglich. Der Controller unterstützt allerdings die Kommunikation per USB 1.1, die eine Datenrate von 12 MHz erfordert, nativ. Für den High-Speed-Modus ist ein externer Baustein nach dem USB Low Pin Interface (ULIP)-Standard nötig. Im vorliegenden Fall wird der IC USB3300 von Microchip verwendet. Dieser Baustein ist über acht parallele Datenleitungen mit dem Prozessor verbunden, sodass trotz der im Verhältnis niedrigen Taktrate des Prozessors die nötige Bandbreite erreicht werden kann. Die Umschaltung zwischen Full-Speed-Übertragung mit dem im Controller integrierten USB-Modul und High-Speed-Übertragung mit einem externen Bauteil nach dem ULIP-Standard ist aus Sicht der Anwendung völlig transparent und wird von der *usblib* und dem Controller intern gehandhabt.

[6] Zur Erklärung dieses Wertes sollte das Geburtsdatum des Autors einer genaueren Betrachtung unterzogen werden.

5 Der USB-Treiber

Der Begriff „Treiber“ kann unterschiedliche Bedeutungen haben. Meistens wird dabei von Software gesprochen, die nahe am Betriebssystemkern des Rechners arbeitet, und den Zugriff auf die Hardware gegenüber höheren Schichten und Anwendungen abstrahiert. Im weiteren Sinne ist damit aber auch jede Art von Software gemeint, die den Zugriff auf Hardware vereinfacht, auch wenn es sich dabei nicht um einen klassischen Hardware-Treiber handelt, der vom Betriebssystem geladen wird. Der hier entwickelte Treiber fällt in die zweite Kategorie, weswegen zur Abgrenzung in dieser Arbeit gelegentlich von „Treibersoftware“ gesprochen wird.

5.1 Grundlegende Überlegungen

Neben der Entwicklung der Software, die auf dem Mikrocontroller der Datenerfassungskarte läuft, wurde im Rahmen dieser Arbeit auch Software entwickelt, welche die Ansteuerung der Karte vom Computer aus ermöglicht. Hauptanwendungszweck war dabei der Zugriff aus *LabView*, die Software wurde aber so entwickelt, dass ein Zugriff aus beliebigen Anwendungen möglich ist. Primäres Ziel bei der Entwicklung dieses „Treibers“ war die vollständige Abbildung der Funktionalität der Karte, ohne dabei die Performance negativ zu beeinflussen. Zusätzlich sollte eine Einbindung in *LabView* möglichst wenig Änderungen an den bestehenden Blöcken nach sich ziehen. Das Design des Treibers wurde daher nahe an dem schon für die USB-Messkarte bestehenden gehalten. Zusätzlich zu den geforderten Eigenschaften wurde der neu entwickelte Treiber so entworfen, dass er sowohl auf die USB-Messkarte als auch auf das neue Datenerfassungsmodul zugreifen kann. Auf diese Weise kann aus *LabView* der Zugriff unabhängig von der verwendeten Karte erfolgen und eine Fallunterscheidung wird vermieden. Eine Erweiterung des alten Treibers für die neue Karte wurde erwogen, in der Neuerstellung eines Treibers wurden aber Chancen gesehen, die recht umständliche Anbindung an *LabView* zu vereinfachen.

5.2 Das Windows-Treibermodell

Primäres Ziel bei der Entwicklung der Treibersoftware war das Betriebssystem Windows in der Version 7 oder höher. Im Folgenden soll ein Überblick über das Windows-Treibermodell gegeben werden, um die Entscheidungen bei der Entwicklung des Treibers besser nachvollziehbar zu machen. Weiterführende Informationen zu dem Thema Windows-(USB-)Treiber sind in [24, 9] zu finden, die Erläuterungen in diesem Abschnitt basieren auf diesen Quellen.

In der Architektur des Windows-Betriebssystems werden sogenannte *Driver Stacks* benutzt. Die einzelnen Treiber sind in einem Stapel angeordnet, jeder Treiber kann

dabei auf die Schnittstellen des Treibers aus der Schicht unter ihm zugreifen. Der Windows-Geräte-Manager kann diese Schichtung (in begrenztem Umfang) darstellen, wie Abbildung 5.1 zeigt. Die dort selektierte Maus wird etwa von dem Windows-Standard-Maustreiber *mouhid* betrieben, wie in Eigenschaften auf der Registerkarte *Details* dem Eintrag *Dienst* entnommen werden kann. Diese Maus ist Teil eines allgemeinen HID-Gerätes, betrieben vom Treiber *HidUsb*. Dieses Gerät ist direkt an ein USB-Root-Hub des Mainboards angeschlossen, eingeschobene Hubs wären jedoch denkbar. Das USB-Root-Hub des Mainboards wird von dem Treiber *usbhub* betrieben, der seinerseits auf den Treiber *usbuhci* des Host-Controllers zurückgreift[1]. Da der Controller auf dem Mainboard über PCI mit der CPU verbunden ist, bildet die letzte Ebene des Treiberstapels der Treiber *pci*.

Der Aufbau dieses Stapels erfolgt beim Start des Betriebssystems durch das *Plug&Play*-System. Sofern es von der jeweiligen Hardware unterstützt wird, können Geräte auch im laufenden Betrieb hinzugefügt oder entfernt werden (hot-plugging). Beim Start des Betriebssystems wird zunächst der PCI-Treiber geladen, der dann auf den PCI-Bus zugreift und die dort vorhandenen Geräte enumeriert. Jedes gefundene Gerät wird dabei dem *Plug&Play"*=System gemeldet, das dann weitere Treiber lädt. Für den Treiber in der höheren Schicht ist es dabei (größtenteils) unerheblich, über welche untere Schicht die Verbindung zu „seiner“ Hardware zustande kommt. Ein USB-Controller kann so von dem gleichen Treiber angesteuert werden, unabhängig davon ob die Verbindung zur CPU über PCI oder beispielsweise Thunderbold[2] zustande kommt. Da sowohl für die wichtigsten unteren Schichten wie PCI(-Express) etc., als auch für wichtige Endgeräte wie Mäuse, Tastaturen etc., Treiber von Microsoft zusammen mit Windows ausgeliefert werden, ist bei einer Neuentwicklung immer nur ein kleiner Teil des Treiberstapels zu ersetzen.

Ein typischer Treiber für ein USB-Gerät, der sich an oberster Stelle des Treiberstapels befindet, kann daher durch den darunter liegenden USB-Treiber auf die Abstrahierung des Gerätes in Form von *Pipes*, die zu den Endpoints des Gerätes führen, zugreifen und bildet diese für die Anwendung auf einfachere Kommandos ab. Die Kommunikation erfolgt dabei über die allgemeinen Funktionen `ReadFile`, `WriteFile` und `IoControl` des Betriebssystems im Zusammenspiel mit virtuellen Gerätedateien.

Das Plug&Play-System von Windows erfordert es, das zu jedem Gerät, das von einer unteren Schicht erkannt wurde, ein passender Treiber geladen wird. Sollte kein Treiber gefunden werden, kann das Gerät nicht genutzt werden. Insbesondere ist auch kein direkter Zugriff von Anwendungen auf die weiter unten liegenden Schichten möglich, nur der oberste Treiber eines Stapels kann von Anwendungen angesprochen werden. Diese Einschränkung ist insbesondere bei einfachen USB-Geräten hinderlich, bei denen es vertretbar wäre, direkt auf die Endpoints eines Gerätes zuzugreifen, wie sie vom allgemeinen USB-Treiber des

[1] Der *Universal*-Host-Controller ist für Low- und -Full-Speed Geräte zuständig, ein High-Speed-Gerät würde in einem separaten Treiberstapel ab dem *Enhanced*-Host-Controller (mit dem Treiber *usb**e**hci*) eingebunden werden

[2] Eine von Intel und Apple entwickelte externe Schnittstelle, ähnlich zu USB. Es sind Dockingstationen erhältlich, die einen USB-Controller, neben anderen Schnittstellen, über Thunderbold anbinden.

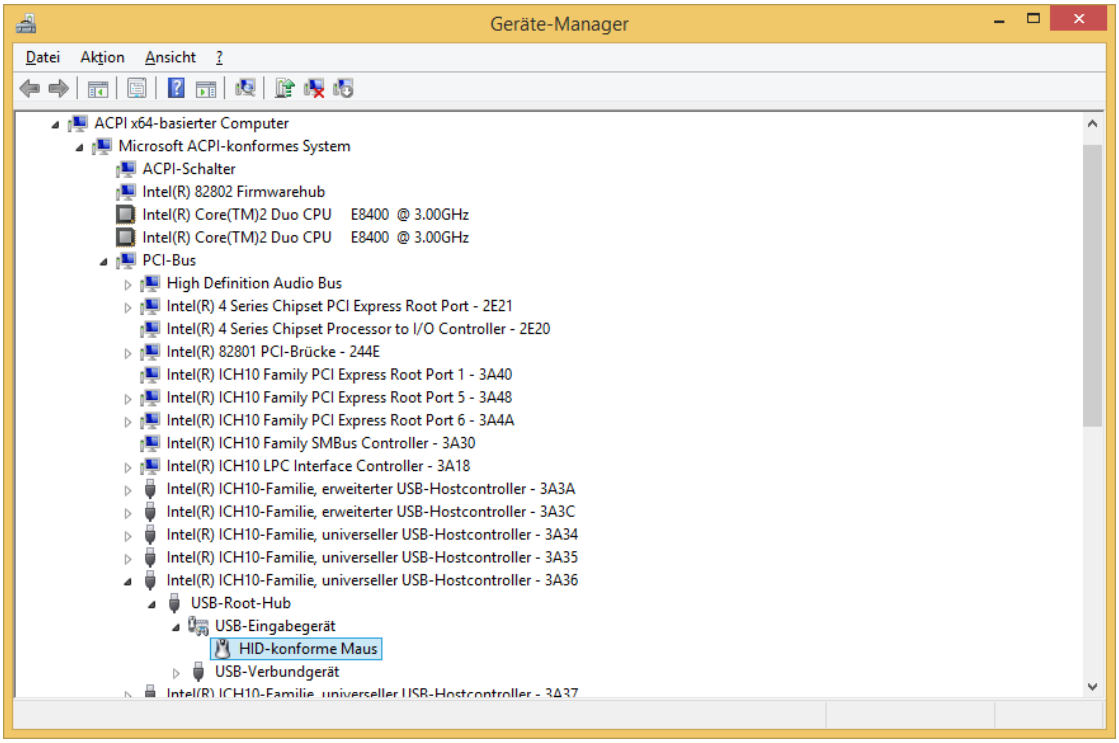


Abbildung 5.1: Der Windows Geräte-Manager zeigt die Schichtung der Treiber an, wenn Ansicht → Geräte nach Verbindung gewählt wird

Betriebssystems angeboten werden[3]. Die Logik des Treibers wird dabei in die Anwendung verschoben.

Die Erstellung eine USB-Treibers ist sehr aufwendig und erfordert äußerste Sorgfalt. Ein solcher Treiber läuft im privilegierten Kernel-Modus des Systems, was zum einen die Fehlersuche durch Debugging erschwert, zum anderen aber auch dazu führt, dass ein Fehler in einem solchen Treiber weitreichende Konsequenzen, bis zum Absturz des gesamten Systems, haben kann, da nahezu alle Schutzmechanismen des Betriebssystems im Kernel-Modus nicht aktiv sind.

5.2.1 WinUSB

Da die Erstellung eines Treiber, wie oben dargelegt, sehr aufwendig ist und für einfache USB-Geräte, speziell wenn sie nur von einer Anwendung genutzt werden sollen und eine Abstimmung von gleichzeitigen Zugriffen daher nicht nötig ist, auch nur wenig Vorteile bietet, gibt es von Microsoft seit Windows XP SP2 den sogenannten *WinUSB*-Treiber. Dabei handelt es sich um einen Kernel-Treiber, der für beliebige USB-Geräte geladen werden kann und im Wesentlichen alle Operationen des darunterliegenden Treibers an die

[3]Unter Linux und Mac OS ist dies beispielsweise möglich

Anwendung weiterleitet. Teil dessen ist auch die Programmbibliothek *winusb.dll*, die von einem Anwendungsprogramm (außerhalb des Kernel-Modus) genutzt werden kann, um auf ein Gerät mit WinUSB-Treiber zuzugreifen. Die Dokumentation der API ist in [11] zu finden.

Da die Benutzung von WinUSB die Entwicklung eines Treiber, wie oben dargestellt, einfacher und weniger fehleranfällig macht, wurde der vorliegende Treiber darauf aufgebaut und kein eigener Kernel-Treiber entwickelt.

5.2.2 Auswahl der Treiber durch Windows

Die Auswahl eines passenden Treibers erfolgt durch sogenannte *Device Identification Strings*, die bei der Enumeration durch einen tiefer liegenden Treiber dem *Plug&Play*-System gemeldet werden [13]. Dabei handelt es sich zum einen um *Hardware IDs*, die das spezielle Gerät, das angeschlossen wurde, beschreiben, zum anderen um *Compatible IDs*, die Klassen von Geräten beschreiben, zu denen das Gerät kompatibel ist.

Die Datenerfassungskarte meldet die Strings `USB\VID_1CBE&PID_0102&REV_0100` sowie `USB\VID_1CBE&PID_0102` als Hardware-IDs[4], die dort verwendeten Daten, also Vendor- und Product-ID, sowie die Revisionsangabe werden dem Device-Deskriptor, der bei der Enumeration abgefragt wird, entnommen. Diese IDs werden nun zur Auswahl eines passenden Treibers verwendet. Sofern der Rechner mit dem Internet verbunden ist, werden diese IDs an das Windows-Update-System geschickt, in dem Hersteller aktuelle Treiber für ihre Geräte hinterlegen können. Sofern keine Verbindung möglich ist oder kein passender Treiber gefunden wird, wird die Menge von auf dem Computer vorhandenen Treibern durchsucht. Aus diesem Grund gehört zu jedem Treiber neben der SYS-Datei, in der der eigentliche Programmcode enthalten ist, auch immer eine INF-Datei, in der Metadaten zu dem Treiber hinterlegt sind, unter anderem die von diesem Treiber unterstützten Geräte in Form von den oben genannten IDs. Windows verwendet die IDs in absteigender Reihenfolge zur Suche, sodass immer der Treiber mit der spezifischsten Angabe gewählt wird[5] [20]. Die Auswahl eines Treibers wird für ein Gerät gespeichert, sodass der Treiber beim nächsten Anschließen des Gerätes ohne erneute Suche zur Verfügung steht. Wenn das Gerät in seinem Device-Deskriptor die Kompatibilität zu einer der vom USB IF definierten Klassen angibt, werden zusätzlich die *Compatible IDs* der Form `USB\Class_XX&SubClass_YY` und `USB\Class_XX` generiert. Auf diese Weise kann etwa ein generischer Treiber für Drucker hinterlegt werden, der verwendet wird, wenn kein gerätespezifischer Treiber vorhanden ist. Ausführliche Informationen zu diesem Thema finden sich auch in [25].

Windows Common Device Identification

Da keine allgemeine ID für alle Arten von USB-Geräten generiert wird, ist es nicht möglich, eine INF-Datei für einen Treiber zu erzeugen, der auf jedes USB-Gerät passt, wie es etwa für den WinUSB-Treiber nützlich wäre. Für die USB-Messkarte wurde daher

[4] Im Geräte Manager unter den Details eines Gerätes zu finden

[5] Wenn also ein Treiber für spezielle Revision des Gerätes hinterlegt ist, wird dieser verwendet. Falls nicht, wird nach einem Treiber für das Gerät allgemein, ohne Revisionsangabe, gesucht.

Offset	Wert	Bemerkung
0	0x12	Länge des Deskriptors (18 Byte)
1	0x03	Deskriptor Typ (3 = String)
2	0x4D, 0x00, 0x53, 0x00, 0x46, 0x00, 0x54, 0x00, 0x31, 0x00, 0x30, 0x00, 0x30, 0x00	Signatur "MSFT100"(Unicode)
16	variabel	*Request Code*
17	0x00	reserviert

Tabelle 5.1: Das Format des String-Desktiptors 0xEE für WCID. Der *Request Code* ist beliebig und wird vom Betriebsystem bei der Abfrage der weiteren Deskriptoren verwendet. Quelle: [14]

eine eigene INF-Datei erzeugt, die den WinUSB-Treiber als kompatibel markiert. Dieses Vorgehen erfordert jedoch, dass die INF-Datei auf jedem Computer installiert sein muss, an dem die USB-Messkarte verwendet werden soll. Mit Windows Vista wurde daher ein neues Konzept namens „Windows Common Device Identification“, kurz WCID eingeführt. Mit diesem Konzept sollte eine flexiblere Variante zu den *Compatible IDs*, die auf den durch das USB IF definierten Geräte-Klassen basieren, geschaffen werden. Eine sehr gute Beschreibung des Systems ist in [14] zu finden.

Zur Nutzung von WCID muss das Gerät auf eine Abfrage des String-Deskriptors 0xEE mit einem Deskriptor gemäß Tabelle 5.1 reagieren. Dieser Deskriptor wird während der Enumeration automatisch abgerufen, sofern er existiert. Sollte der Deskriptor vorhanden sein und dem Schema entsprechen, wird danach ein *Vendor Request* mit Index 0x0004 und Request-Code gemäß dem Deskriptor an das Gerät gestellt. Durch diese Anfrage muss ein *Microsoft Compatible ID Feature Descriptor* gemäß Tabelle 5.2 zurückgegeben werden. Die dort angegebene *Compatible ID* wird schlussendlich in der Form `USB\MS_COMP_`*`X`* an das *Plug&Play*-System zusätzlich zu den anderen IDs gemeldet. Das *X* wird dabei durch die gemeldete, bis zu acht Zeichen lange ID ersetzt. Die INF-Datei des WinUSB-Treiber gibt die ID `USB\MS_COMP_WINUSB` an, daher wird dieser Treiber für ein Gerät automatisch installiert, wenn die ID `WINUSB` abgegeben wurde und kein spezifischerer Treiber zur Verfügung steht. Dies stellt eine große Vereinfachung dar, da eine Verteilung einer angepassten INF-Datei nicht nötig ist. Mit Windows 8 ist der WinUSB-Treiber in der Standard-Installation des Betriebssystems enthalten, sodass auch keine Online-Verbindung zu Windows Update mehr nötig ist. Dieses System ist nicht auf WinUSB beschränkt, andere Treiber können auf die gleiche Art und Weise geladen werden. Es stellt also im Wesentlichen eine flexiblere Version der Geräteklassen des USB-Standards dar.

Die Abfrage des String-Deskriptors 0xEE bei der Enumerierung erfolgt immer mit der ungültigen Sprach-ID 0x0000, auch wenn das Gerät diese Sprache nicht als unterstüzt angibt. Hierzu konnte keine Dokumentation gefunden werden, das Verhalten ist aber in der Aufzeichnung des USB-Verkehrs mit dem *Microsoft Message Analyzer* deutlich zu sehen, wie Abbildung 5.2 zeigt. Dieser Umstand musste im Laufe der Implementierung gesondert beachtet werden, siehe Abschnitt 6.9.3.

Offset	Wert	Bemerkung
0	0x28, 0x00, 0x00, 0x00	Länge des Deskriptors (40 Byte)
4	0x00, 0x01	Version (1.0)
6	0x04, 0x00	Deskriptor-Index (4)
8	0x01	Anzahl der Abschnitte (1)
9	0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00	reserviert
16	0x00	Interface-Nummer (0)
17	0x00	reserviert
18	0x57, 0x49, 0x4E, 0x55, 0x53, 0x42, 0x00, 0x00	Compatible ID ("WINUSB"), bis zu 8 ASCII Zeichen
26	0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00	Sub-Compatible ID, bis zu 8 ASCII Zeichen, ungenutzt
34	0x00, 0x00, 0x00, 0x00, 0x00, 0x00	reserviert

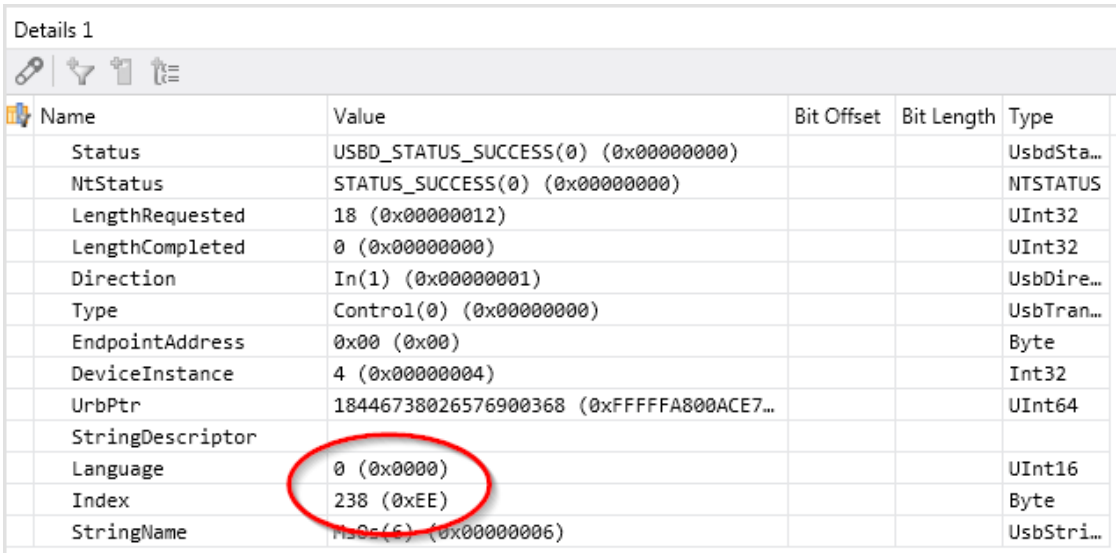
Tabelle 5.2: Das Format des Microsoft Compatible ID Feature Descriptor. Das Beispiel zeigt einen Deskriptor für ein Interface, welches als *Compatible ID* „WINUSB“ angibt, entsprechend sind auch andere Angaben mögich. Quelle: [14]

Ein weiteres Problem stellt die automatische Installtion des Treibers dar. Die Installation des Treibers funktioniert unter Windows 8 problemlos, unter Windows 7 ist der Treiber allerdings nicht in der Standard-Installation enthalten. Ein automatischer Download von Windows Update war reproduzierbar nicht möglich. Die Ursache des Problems konnte nicht ausgemacht werden, der Treiber kann aber manuell heruntergeladen werden[6]. Nach dem Download kann die CAB-Datei mit einem Pack-Programm wie etwa *7-Zip* ([28]) entpackt und mit dem Befehl `pnputil -a winusbcompat.inf` ins System eingefügt werden [31]. Die entpackte Datei sollte dabei nicht auf einem Netzwerklaufwerk liegen, der Befehl muss zudem mit Administratorrechten ausgeführt werden.

5.3 Wahl der Programmiersprache

Zur Entwicklung des Treibers wurde die Sprache C# gewählt. Diese Sprache nutzt das .NET-Framework von Microsoft und ist daher eine sogenannte *verwaltete* Sprache. Dies bedeutet unter anderem, das die Speicherverwaltung automatisch durchgeführt wird. Da nicht benutzter Speicher durch einen Garbage-Collector automatisch freigegeben wird, können keine Speicherlecks entstehen. Als weiterer Vorteil ist zu sehen, dass durch die verwaltete Architektur immer sichergestellt ist, dass alle Handles des Systems wieder korrekt geschlossen und freigegeben werden. Dies war ein großes Problem des alten, in nicht verwaltetem C++ geschriebenen Treibers, da bei einem Absturz des Programms oft die USB-Verbindung weiter als geöffnet galt, und keine neue Verbindung geöffnet werden

[6] `http://catalog.update.microsoft.com/v7/site/Search.aspx?q=usb%5Cms_comp_winusb` Download nur unter Windows und mit dem Internet Explorer möglich.


Details 1

Name	Value	Bit Offset	Bit Length	Type
Status	USBD_STATUS_SUCCESS(0) (0x00000000)			UsbdSta…
NtStatus	STATUS_SUCCESS(0) (0x00000000)			NTSTATUS
LengthRequested	18 (0x00000012)			UInt32
LengthCompleted	0 (0x00000000)			UInt32
Direction	In(1) (0x00000001)			UsbDire…
Type	Control(0) (0x00000000)			UsbTran…
EndpointAddress	0x00 (0x00)			Byte
DeviceInstance	4 (0x00000004)			Int32
UrbPtr	18446738026576900368 (0xFFFFFA800ACE7…			UInt64
StringDescriptor				
Language	0 (0x0000)			UInt16
Index	238 (0xEE)			Byte
StringName	MSOS(6) (0x00000006)			UsbStri…

Abbildung 5.2: Screenshot des *Microsoft Message Analyzes*. Zu sehen ist die Abfrage des String-Deskriptors 0xEE, es wird deutlich, dass das Betriebssystem die Sprach-ID 0x0000 verwendet.

konnte. Genereller Nachteil von verwalteten Sprachen ist die Abhängigkeit zum .NET-Framework. Da Windows seit der Version 7 mit einer vorinstallierten Version[7] des .NET-Frameworks geliefert wird, ist diese Abhängigkeit nicht kritisch.

5.3.1 LibUSB.NET

Da es sich bei der Programmbibliothek *winusb.dll* um eine nicht verwaltete DLL handelt, ist der Zugriff aus C# ohne weitere Hilfsmittel nicht möglich. Für die Entwicklung des Treibers wurde daher die Programmbibliothek *LibUSB.NET* benutzt [12], welche eine Anbindung an C# und andere Spachen des .NET-Frameworks ermöglicht. Die Bibliothek wurde hauptsächlich für den Einsatz mit der *libusb* entworfen, einer aus der Linux-Welt stammenden Bibliothek zur Ansteuerung von USB-Geräten. Unter Windows ist aber auch die Ansteuerung von WinUSB-Geräten problemlos möglich. Durch den plattformunabhängigen Ansatz von *LibUSB.NET* und des .NET-Frameworks im Allgemeinen sollte auch eine Ansteuerung des Datenerfassungsmoduls unter anderen Betriebssystemen als Windows möglich sein. Dies wurde jedoch nicht weiter untersucht.

[7] Es handelt sich um die inzwischen veraltete Version 2.0 des Frameworks. Da aber die Verwendung einer neueren Version das Abhängigkeitsproblem verschärft hätte, und auch die Einbindung in *LabView* dann nicht ohne weitere Konfiguration möglich gewesen wäre, wurde diese Einschränkung in Kauf genommen

5.4 Implementierung des Treibers

Die Implementation des Treibers ist in dem *VisualStudio*-Projekt *UsbCommunicationDll* zu finden. Das Projekt wird für das .NET-Framework 2.0 übersetzt, benutzt aber einige Sprachfeatures neuerer Versionen der Sprache C#. Es ist daher ein Compiler nötig, der die Sprachversion 4.0 von C# unterstüzt. Zusätzlich verwendet das Projekt die Bibliothek *LINQBridge* um einige der LINQ-Funktionen aus dem .NET-Framework 3.5 verwenden zu können. Diese Bibliothek, sowie *LibUSB.NET* sind als NuGet[8]-Paket eingebunden. Das gesamte Projekt ist mit allen Abhängigkeiten und Beispielcode zur Verwendung auf der CD im Anhang dieser Arbeit zu finden.

Der Haupteinstiegspunkt der Anwendung ist die statische Klasse `UsbConnectionFactory` und ihre Methode `GetConnection`. Diese Methode listet alle WinUSB-Geräte auf und durchsucht die Liste nach USB-Messkarten und Datenerfassungsmodulen. Zur Suche werden GUIDs verwendet, die bei der Installation des Treibers in der Registry abgelegt werden. Im Fall der USB-Messkarte ist die GUID in der INF-Datei, die zur Installation des Treibers benötigt wird, enthalten, im Fall des Datenerfassungsmoduls wird die GUID dem Betriebssystem über einen *Microsoft Extended Properties Feature Descriptor*, ähnlich dem schon erläuterten *Microsoft Compatible ID Feature Descriptor*, mitgeteilt, der auch im Rahmen der WCID-Erkennung abgerufen wird. Sofern die Suche erfolgreich war, wird eine Unterklasse von `UsbConnection`, je nach Typ des gefundenen Gerätes, zurückgegeben.

Die Oberklasse `UsbConnection` stellt dabei eine allgemeine Schnittstelle zu Datenerfassungsmodul und USB-Messkarte dar. Zusätzlich enthält sie die Implementierung einiger Funktionen, die zwischen beiden Karten gleich sind, insbesondere Funktionen um die empfangenen Spektren im Treiber aufzusummieren[9]. Die Anzahl der Aufsummierungen kann mit der Eigenschaft `AverageCount`[10] eingestellt werden. Es ist darauf zu achten, dass die Anzahl der Summierungen im Treiber zusammen mit den Summierungen direkt auf der Datenerfassungskarte, sofern verwendet, die Zahl 256 nicht übersteigt, da es ansonsten zu Messfehlern durch arithmetischen Überläufe kommen kann. Eine Übersicht über den Datenfluss innerhalb des Treibers ist in Abbildung 5.3 zu sehen.

Die weitere öffentliche Schnittstelle besteht aus den Methoden `Read` und `Write` mit denen Daten an EP1 gesendet/von EP1 gelesen werden können. Die Daten sollten sinnvollerweise dem USB-Transportprotokoll entsprechen. Die Methoden sind als abstrakt gekennzeichnet, da die eigentliche Implementierung von der verwendeten Karte abhängig ist. Die Verbindung zu dem entsprechenden Gerät wird durch die Methode `Open` bzw. `Close` verwaltet. Ein Schließen der Verbindung mit `Close` wird empfohlen, die Verbindung wird durch das .NET-Framework aber auch automatisch geschlossen, wenn die Instanz von `UsbConnection` aus dem Speicher geräumt wird. Die Schnittstelle zu den erfassten Spektren besteht aus den Methoden `HasSpectrum`, `GetSpectrum` und `FlushSpectrumBuffer`

[8] Ein Paket-Manager für .NET, weitere Informationen unter `http://www.nuget.org/`

[9] Dies sollte nicht mit der gleichartigen Funktion des Datenerfassungsmoduls verwechselt werden. Diese Funktion läuft auf dem Host-Computer und kann auch mit der USB-Messkarte benutzt werden.

[10] Trotz des Namens handelt es sich nicht um eine Durchschnittsbildung, da nicht durch die Anzahl der Summierungen geteilt wird. Dies muss von dem verarbeitenden Code, sofern gewünscht, erledigt werden.

sowie dem Event `NewSpectrumReady`. Eine Anwendung, die den Treiber benutzt, kann auf diese Weise entweder periodisch die Funktion `HasSpektrum` aufrufen, um abzufragen, ob neue Daten zur Verfügung stehen, oder sich über das Ereignis `NewSpectrumReady` benachrichtigen lassen. In jedem Fall kann das neue Spektrum dann mit der Funktion `GetSpectrum` abgerufen werden. Im Treiber werden bis zu acht (summierte) Spektren zwischengespeichert, wenn sie vom aufrufenden Code nicht ausreichend schnell gelesen werden. Sollte der Puffer überlaufen, wird das älteste Spektrum verworfen. Ein manuelles Leeren des Puffers ist durch die Funktion `FlushSpectrumBuffer` möglich.

Die Spektren werden als Instanz der Klasse `Spectrum` übergeben, die im Wesentlichen die Elemente des Spektrum-Headers (siehe Tabelle 2.7) auf Eigenschaften abbildet. Die eigentlichen Daten des Spektrums sind in dem Array `Data` zu finden, dessen Länge sich der Menge der erfassten Daten anpasst.

5.4.1 Die Klasse `UsbMesskarteUsbConnection`

Diese Klasse ist eine Unterklasse von `UsbConnection` und stellt eine Verbindung zu einer alten USB-Messkarte dar. Ein wesentlicher Unterschied ist das Protokoll, mit dem die Daten vor der Übertragung zur Karte eingefasst werden. Die USB-Messkarte verwendet hier eine einfache Form des USB-TMC-Protokolls, das an dieser Stelle generiert wird. Der Code basiert größtenteils auf dem bestehenden Treiber der USB-Messkarte und wurde von C++ in C# übersetzt.

Zusätzlich zu den Funktionen aus der Oberklasse `UsbConnection` verfügt die Klasse `UsbMesskarteUsbConnection` über die Methode `TriggerSpectrum`, mit der die Aufzeichnung eines neuen Spektrums auf der Karte ausgelöst wird, das danach an den Computer übertragen wird. Das Datenerfassungsmodul verwendet eine andere, umfangreichere Methode, das Aufzeichnen von Spektren auszulösen, daher ist die Methode nicht in der gemeinsamen Elternklasse enthalten. Dies bedeutet insbesondere, dass eine Instanz von `UsbConnection`, die von `UsbConnectionFactory.GetConnection` erhalten wurde, in die Unterklasse gecastet werden muss, wenn diese Methode benutzt werden soll.

Ein weiterer Unterschied zwischen den Karten ist das Format, in dem die Spektren übertragen werden. Die USB-Messkarte überträgt ausschließlich die Rohwerte und keine weiteren Metadaten. Die entsprechenden Felder in der Klasse `Spectrum` werden daher nachträglich gefüllt. Das größte Problem stellt dabei das Feld `Timestamp` dar. Um die Verarbeitung der Spektren zu vereinfachen, sollte der Timestamp das gleiche Format wie die Timestamps des Datenerfassungsmoduls haben. Da von der Karte kein Zeitstempel übertragen wird, kann nur die Zeit des Computers beim Empfang des Spektrums als Zeitbasis verwendet werden. Diese enthält allerdings auch mögliche Verzögerungen durch eine Überlastung des Computers[11]. Um die Zeitstempel vergleichbar zu halten, wird die aktuelle Zeit beim Öffnen der Verbindung zu einer USB-Messkarte gespeichert und die

[11] Dies war der hauptsächliche Grund, im Datenerfassungsmodul die Zeitstempel auf der Karte selbst zu generieren.

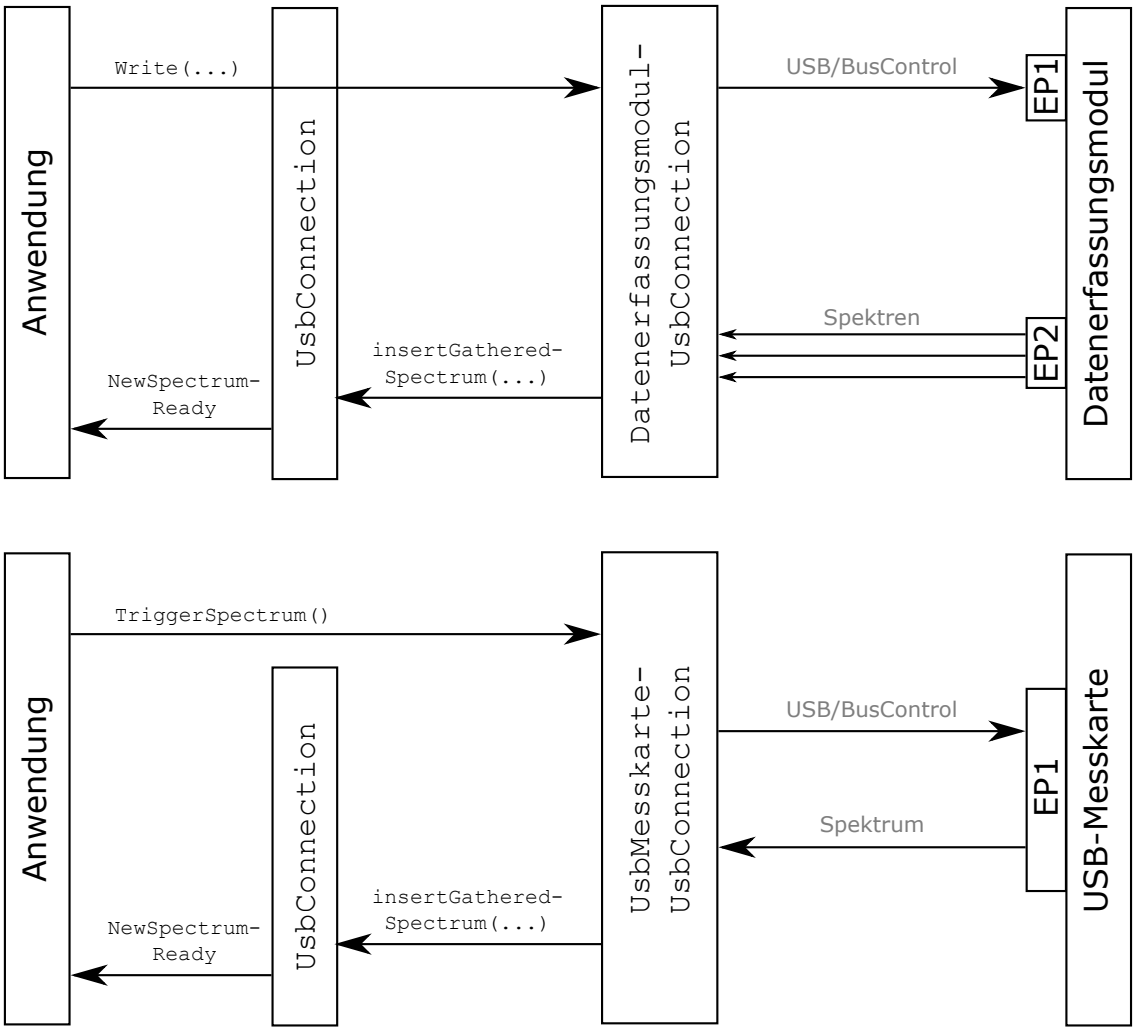


Abbildung 5.3: Der Datenfluss im Treiber. Die obere Hälfte der Abbildung zeigt die Verwendung eines neuen Datenerfassungsmoduls, die untere den Datenfluss in Verbindung mit einer alten USB-Messkarte.

Differenz zur aktuellen Zeit beim Empfang eines Spektrums als Zeitstempel verwendet[12]. Das Format wird dabei an das des Datenerfassungsmoduls angepasst.

5.4.2 Die Klasse `DatenerfassungsmodulUsbConnection`

Diese Klasse stellt eine Verbindung zu einem Datenerfassungsmodul dar. Zur Übertragung der Daten wird hier ein sehr viel schlankeres Protokoll verwendet, dass auf der Länge der Pakete basiert, wie in Abschnitt 2.2.5 beschrieben. Beim Lesen wird daher die Länge der empfangenen Pakete mit der Maximalgröße verglichen, und der Lesevorgang erst beendet, wenn ein nicht komplett gefülltes Paket erkannt wird. Beim Schreiben erfolgt die Aufteilung in Pakete, die jeweils der Maximalgröße entsprechen, automatisch, das Einfügen eines ZLP muss aber gegebenenfalls manuell geschehen.

Das Datenerfassungsmodul verfügt, neben der Möglichkeit Spektren manuell auszulösen, über einen Modus, in dem Spektren automatisch erfasst werden (siehe Abschnitt 2.2.4). Der Code, der die Spektren, die auf EP2 gesendet werden, empfängt, muss daher asynchron laufen, da jederzeit von der Karte Daten übertragen werden können. Hierbei ist zu beachten, dass die Menge der in einem Schritt empfangenen Daten variieren kann, es ist insbesondere nicht garantiert, dass der Ereignis-Handler für den Empfang von Daten für jedes USB-Paket genau ein Mal aufgerufen wird. Das Ende eines Spektrums kann daher nicht durch ein ZLP erkannt werden. Es muss beim Empfang der Daten ferner darauf geachtet werden, dass sowohl so wenig Daten übertragen werden könnten, dass noch nicht ein Mal die vier Byte des Headers, die die Länge angeben, vorliegen, genau so ist es aber auch möglich, dass schon Daten des nächsten Spektrums in einer Übertragung enthalten sein könnten. Diese Sonderfälle werden im Code behandelt.

Beim Herstellen oder Trennen der Verbindung sendet der Treiber einen Control-Transfer mit dem Code 0x21 an die Karte, die auf diese Weise erfährt, ob aktuell ein Treiber auf dem Computer verbunden ist[13]. Ohne Verbindung zu der Treibersoftware werden keine Spektren übertragen, da diese nie empfangen würden.

5.5 Anbindung an *LabView*

LabView bietet direkt Unterstützung für die Verwendung von .NET-Code durch spezielle Blöcke [16]. Es ist dabei zu beachten, dass ohne Konfigurationsänderungen nur die Version 2.0 des .NET-Frameworks unterstützt wird [17], der Code der Treibersoftware wurde daher für diese Version geschrieben. Die Sicherheitsfunktionen des .NET-Frameworks erlauben nicht, dass Code von nicht vertrauenswürdigen Speicherorten ausgeführt wird. In der Standardkonfiguration gehören dazu auch Netzwerklaufwerke. Soll die *UsbCommunicationDll* von einem Netzwerklaufwerk geladen werden, muss dieses zunächst als vertrauenswürdig gekennzeichnet werden. Dazu muss als Administrator dieser Befehl ausgeführt wer-

[12] Der Zeitstempelzähler des Datenerfassungsmoduls wird beim Einschalten der Karte zurückgesetzt, dieses Verhalten soll hiermit simuliert werden.

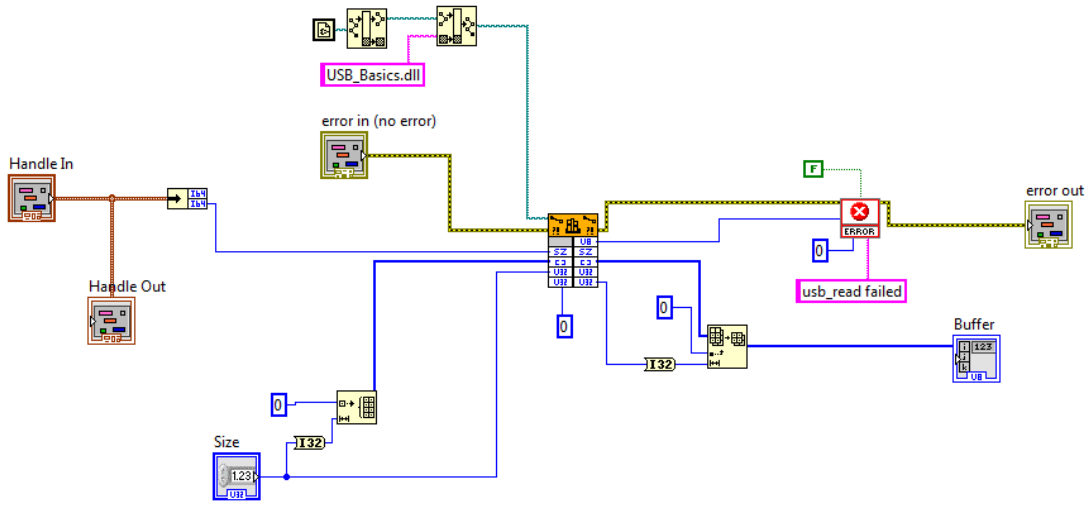
[13] Dies ist nicht zu verwechseln mit der allgemeinen USB-Verbindung. Die Karte kann per USB mit dem Computer verbunden sein, ohne das eine Instanz der Treibersoftware läuft und Daten empfangen kann.

den: `C:\Windows\Microsoft.NET\Framework64\v2.0.50727\CasPol.exe -q -machine -addgroup 1 -url file://x:/* FullTrust -name "Policy-Name"` [18]. Dabei muss `x:` durch den Buchstaben des Laufwerks (oder alternativ den UNC-Pfad) ersetzt werden, `Policy-Name` durch einen beliebigen Namen für diesen Eintrag, beispielsweise den Namen des Netzwerklaufwerks. Gegebenenfalls muss `Framework64` durch `Framework` ersetzt werden, wenn die 32-bit-Version des .NET-Frameworks verwendet wird.

Die nötigen Anpassungen an den *LabView*-Blöcken waren sehr gering, da sich die Funktionen, die auf den Treiber zugreifen, auf nur wenige Blöcke konzentrieren. Der Aufbau konnte in den meisten Fällen stark vereinfacht werden, da die Integration von .NET-Code in *LabView* deutlich einfacher ist, als die Integration von nativem Code[14]. Alle veränderten Blöcke sind auf der CD im Anhang dieser Arbeit zu finden. Abbildung 5.4 zeigt einen *LabView*-Block vor und nach der Anpassung auf den neuen Treiber.

Bei Tests der .NET-Einbindung konnte *LabView* reproduzierbar zum Absturz gebracht werden, wenn ein .NET-*Refnum* nicht korrekt über einen *Close Refnum*-Block geschlossen wurde, bevor das Programm beendet wurde. Dies stellt insbesondere ein Problem dar, wenn ein Programm durch den Knopf „Abort“ abgebrochen wird. Hierbei scheint es sich um einen Fehler in *LabView* zu handeln, eine Lösung hierfür konnte nicht gefunden werden. Es wird empfohlen, Programme nicht über „Abort“ zu beenden und darauf zu achten immer die vorgesehenen Blöcke zum Schließen der Verbindung zu nutzen, die intern *Close Refnum* nutzen.

[14] Insbesondere fällt das Speichermanagment und das Übergeben der Array-Längen weg.



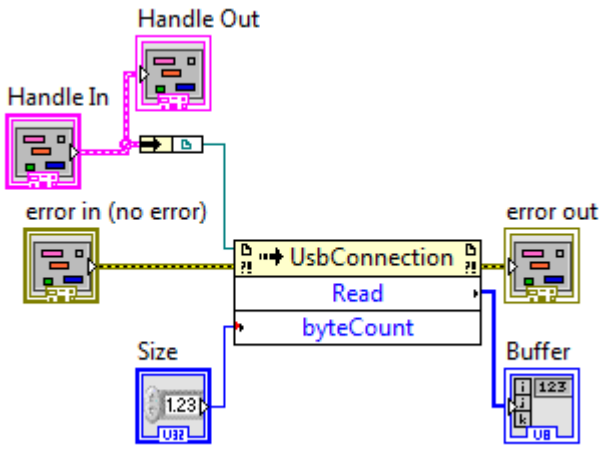


Abbildung 5.4: Der Block *usb_read* vor und nach der Umstellung auf den neuen Treiber. Die untere, neue Version ist sehr viel übersichtlicher und kommt ohne manuelle Verwaltung von Arrays aus.

6 Die Module der Software

In den folgenden Abschnitten werden die einzelnen Module der Software im Detail beschrieben. Es wird dabei immer zunächst Sinn und Zweck des Moduls erläutert, um dann die verwendete Hardware und ihre Besonderheiten zu beschreiben. Im letzten Teil wird dann auf die Implementierung eingegangen. Da der Programmcode ausgiebig kommentiert wurde, wird an dieser Stelle nicht jede Funktion eines Moduls detailliert beschrieben, sondern nur auf wesentliche Besonderheiten oder zugrunde liegende Konzepte eingegangen.

6.1 Das Modul *System*

6.1.1 Funktion und Aufbau

Das Modul *System* enthält einige Funktionen, die an vielen verschiedenen Stellen in der Software benötigt werden. Im Einzelnen sind dies:

- Ein Zeitgeber/Eine Uhr
- Ein Statusregister
- Die Steuerung der Board-LED

Der Zeitgeber

Da die vom Modul erfassten Spektren nicht immer sofort an den Computer übertragen werden können und die Übertragung generell mit einer unbekannten und variablen Verzögerung behaftet ist, ist es wichtig, die Spektren auf dem Modul mit eindeutigen Zeitstempeln zu versehen. Hierfür wurde die `SystemGetTimestamp`-Funktion entworfen. Zusätzlich ist es an verschiedenen Stellen nützlich, die ungefähre Zeit in Millisekunden zu kennen, hierzu existiert die Funktion `SystemGetTime`. Der Zeitgeber beginnt mit der Initialisierung des Moduls zu zählen, hat aber keinen Bezug zur absoluten Zeit, da keine Echtzeituhr vorhanden ist. Dies ist auch nicht notwendig, da es ausreichend ist, die Spektren einer Messreihe eindeutig zu identifizieren und das Modul dabei ununterbrochen läuft.

Das Statusregister

Über das Satz-Modul (siehe Abschnitt 6.4) kann ein spezieller Satz[1] mit Statusinformationen abgerufen werden. Es handelt sich um ein Bitfeld mit 16 Bits die verschiedene

[1] Es handelt sich um Satz 1, der auch zum Erteilen von Kommandos benutzt wird.

Bedeutung	#Bit	Wert
Genereller, nicht weiter bestimmter Fehler	1	1
Aktuelles Pulse-Setup ist gültig	2	2
„Auto-Aquire“ aktiv, Spektren werden automatisch aufgezeichnet und gesendet	3	4
Letztes Spektrum verworfen, weil das davor noch nicht verarbeitet wurde	4	8
Letzter Spektrum-Block verworfen, weil die letzte Übertragung noch nicht abgeschlossen war	5	16
Letzter Spektrum-Block verworfen, weil kein Treiber die Übertragung von Spektren aktiviert hat	6	32
Das USB-Modul ist mit einer Instanz der Treibersoftware verbunden	7	64
Fehler im EEPROM erkannt, Daten sind korrumpiert worden	8	128

Tabelle 6.1: Die Bedeutung der einzelnen Bits im Statusregister (Satz 1). Auch zu finden in der Datei *command_ codes.h*

Zustände des Moduls anzeigen. Der aktuelle Zustand des Statusregisters wird im *System*-Modul verwaltet, das über die Funktion `SystemWriteStatusRegBit` anderen Modulen ermöglich, ihren Status als eines der Bits anzuzeigen. Die Bedeutung der einzelnen Bits ist in Tabelle 6.1 zu finden.

Die LED-Steuerung

Das Board verfügt über eine RGB-LED, über die der Status des Boards ausgegeben werden kann. Im *System*-Modul ist die Steuerung für die LED enthalten. Die LED kann in einer Grundfarbe leuchten und zusätzlich in einer weiteren Farbe blinken. Über die Funktionen `SystemSetLedBaseColor` und `SystemSetLedBlinkColor` kann die Farbe der LED kontrolliert werden, wobei zwischen einmaligem und dauerhaftem Blinken gewählt werden kann. Theoretisch kann die LED durch die Benutzung von PWM nahezu jede beliebige Farbe darstellen, zur Vereinfachung der Ansteuerung sind aber aktuell nur die 3 Grundfarben, alle sich daraus direkt ergebenden Mischfarben, sowie Schwarz und Weiß implementiert. Zusätzlich kann die Konstante `SYSTEM_COLOR_NONE` als „Farbe“ zum Blinken benutzt werden, um anzugeben, dass die LED eben *nicht* blinken soll.

6.1.2 Verwendete Hardware-Module

Das *System*-Modul verwendet ausschließlich den *SysTick*-Hardware-Block des TM4C1294-NCPDT. Dabei handelt es sich um einen einfachen Timer, der von einem vorgegebenen Startwert aus mit dem Systemtakt herunter zählt und beim Erreichen des Wertes Null einen Interrupt auslöst. Für die Ansteuerung der LED werden drei GPIO-Pins ohne weitere Besonderheiten benutzt; das Statusregister ist vollständig in Software implementiert. Der *SysTick* Baustein ist ein Teil des Interrupt-Controllers des TM4C1294NCPDT, daher ist

es nicht nötig, das Modul mit `SysCtlPeriphEnable` zu aktivieren. [1, Abs. 3.1.1]. Der maximale Wert, mit dem der Zähler geladen werden kann, ist 16 777 216 [1, Abs. 3.3].

6.1.3 Besonderheiten der Implementierung

Als Startwert des Zählers wird der maximal mögliche Wert des Zählers (16 777 216) verwendet, da ein kleinerer Wert die Häufigkeit eines Interrupts erhöhen würde, ohne einen weiteren Vorteil mit sich zu bringen.Die Priorität des Interrupts wird auf den zweithöchsten Wert gesetzt, um eine schnelle Verarbeitung des Interrupts sicherzustellen. Dies ist nötig, da die aktuelle Zeit neben dem Zählerstand auch aus der Anzahl der erzeugten Interrupts bestimmt wird. Würde der Interrupt längere Zeit nicht bearbeitet, könnte es passieren, dass der Zähler schon wieder mit dem neuen Wert geladen wurde, die Anzahl der Interrupts aber noch nicht aktualisiert wurde und dadurch kurzfristig die falsche Zeit verwendet würde. Der Interrupt wird durch den Namen `FAULT_SYSTICK` identifiziert, weil der Interrupt durch den *SysTick* intern als einer der „Faults“ des Prozessors geführt wird, da er direkt im Prozessorkern generiert und nicht von außen induziert wird[2].

Ein Interrupt wird alle $^{16\,777\,216}/_{120\,\mathrm{MHz}} \approx 0.1398\,\mathrm{s}$ ausgelöst. Zusammen mit dem 32 bit Zähler für die Anzahl der Interrupts läuft der Zähler damit nach $2^{32} \cdot 0.139\,81\,\mathrm{s} \approx 600\,479\,950\,\mathrm{s} \approx 19.5\,\mathrm{a}$ über, dieser Effekt kann also vernachlässigt werden.

Das Ablaufen des *SysTick*s wird auch gleichzeitig als Zeitbasis für das Blinken der LED benutzt, daher wird im Interrupt-Handler des *SysTick*s die Funktion `ledBlinkTick` aufgerufen; auf diese Weise wird es vermieden, einen zusätzlichen Timer für diese Aufgabe zu benutzen.

6.2 Das Modul *Logging*

Das *Logging*-Modul ist sehr einfach aufgebaut. Dieses Modul ist im Wesentlichen ein Wrapper um das von Texas Instruments im Rahmen der *driverlib* bereitgestellte Codebeispiel `uartstdio.c`, welche eine Funktion bietet, die in großen Teilen zu der C-Funktion `printf` kompatibel ist. Sämtliche Ein- und Ausgaben erfolgen dabei über eines der im TM4C-1294NCPDT verbauten Univeral Asynchronus Receiver/Transmitter (UART)-Module, in diesem Fall das Modul mit der Nummer 2. Das Modul definiert das Makro `log`, mit dem im Code an beliebiger Stelle Meldungen ausgegeben werden können. Zusätzlich ist das Makro `logw` vorhanden, mit dem Warnungen ausgegeben werden können. Normale Meldungen werden nur ausgegeben, wenn die Präprozessordirektive `DEBUG` definiert ist, Warnungen hingegen werden immer ausgegeben. Zusammen mit dem im JTAG-Emulator verbauten UART-Adapter können die Meldungen auf dem Computer gelesen werden. Sämtliche Meldungen werden dabei mit einem Zeitstempel versehen.

Es ist zu beachten, dass es sich bei der Funktion `LoggingLog`, für die `log` ein Alias ist, um eine variadische Funktion handelt, also eine Funktion mit einer variablen Anzahl an

[2] Es handelt sich also keinesfalls um einen Interrupt, der ausgelöst wird, wenn ein Fehler im *SysTick*-System auftritt.

Parametern[3]. Die Argumente werden an die Funktion `UARTvprintf` aus dem Beispielcode übergeben.

6.3 Das Modul *ADC/GPIO*

6.3.1 Funktion und Aufbau

Neben den zwei Eingängen des USB-Datenerfassungsmoduls für die Spektren, die mit hoher Geschwindigkeit und Genauigkeit abgetastet werden können, sind auf dem Bus, mit dem die Karte verbunden ist, noch acht analoge und acht digitale Kanäle für die allgemeine Verwendung vorgesehen. Die analogen Kanäle werden von einem im Controller verbauten ADC ausgewertet, die digitalen Kanäle können sowohl als Ein- als auch als Ausgabe verwendet werden. Zusätzlich verfügt der Mikrocontroller noch über einen internen Temperatursensor, dessen Messwerte von diesem Modul ausgelesen und zusammen mit allen anderen Werten in Sätzen zur Verfügung gestellt werden.

6.3.2 Verwendete Hardware-Module

Die Ansteuerung der digitalen Kanäle (GPIOs) erfolgt über die normale GPIO-Steuerung des Controllers. Es ist zu beachten, dass die Ausgänge des Controllers nicht direkt mit dem Bus verbunden sind, sondern über Pegelwandler, welche die Spannungspegel zwischen den 3.3 V des Controllers und den 5 V des Busses anpassen.

Zur Erfassung der analogen Kanäle wird der ADC mit der Nummer 0 des TM4C1294-NCPDT verwendet. Dieser verfügt über 4 sogenannte Sequencer. Für jeden der Sequencer kann eine bestimmte Menge an Kanäle angegeben werden, die dann von dem Controller beim Auslösen eines Triggers automatisch erfasst und gespeichert werden [1, Abs. 15.3.1]. Der Sequencer 0 verfügt über acht Plätze, daher wird er zum Abfragen der acht analogen Kanäle des Busses benutzt. Zusätzlich wird Sequencer 3 benutzt, der nur über einen einzigen Platz verfügt, um den internen Temperatursensor auszulesen.

Neben dem manuellen Auslösen eines Triggers per Software ist es auch möglich, das Ablaufen eines Timer als Trigger für die ADCs zu verwenden. Von dieser Möglichkeit wird in diesem Modul Gebrauch gemacht. Es wird dazu Timer 0 verwendet, der so konfiguriert ist, dass er einmal pro Sekunde abläuft. Der so erzeugte Trigger löst beide Sequencer aus; gleichzeitig wird in einem Interrupt der aktuelle Zustand der GPIOs ausgelesen. Beiden Sequencern sind unterschiedliche Prioritäten zugewiesen, um sicherzustellen, dass beide nacheinander abgearbeitet werden.

Der Controller wird von einer externen Referenz mit einer stabilisierten Referenzspannung von 3 V versorgt. Diese Spannung ergibt gleichzeitig den Maximalwert, der gemessen werden kann. Da auf dem Bus Spannungen bis zu 10 V erlaubt sind, werden diese von einem Pegelwandler auf der Platine auf den Bereich von 0 V bis 3 V angepasst. Die im

[3] Gleiches gilt für die Funktion `LoggingLogWarn` bzw. `logw`

Controller integrierten Wandler haben eine Auflösung von 12 bit, daher ergibt sich zur Umrechnung des erhaltenen Wertes x und der wirklichen Spannung y

$$y = \frac{x}{2^{12}} \cdot 3\,\mathrm{V}$$

Es ist zu beachten, dass diese Formel nicht im Code implementiert ist. Nur die Rohwerte der Wandler werden in die Sätze geschrieben, dort ist aber diese Umrechnung als Kalibrierkurve (siehe Abschnitt 2.2.3) hinterlegt, sodass bei der Weiterverarbeitung eine Umrechnung vorgenommen werden kann. Durch diese Vorgehensweise kann jede Karte auch individuell kalibriert werden, was zur Erhöhung der Messgenauigkeit beiträgt.

Der Rohwert des internen Temperatursensors kann auf die gleiche Weise in eine Spannung umgewandelt werden, von größerem Interesse ist aber für gewöhnlich die Temperatur selbst. Mit der in [1, Abs. 15.3.6] angegeben Formel

$$V_{TSENS} = 2.7\,\mathrm{V} - \frac{TEMP + 55}{75}$$

(mit V_{TSENS} als gemessene Spannung und $TEMP$ als wirkliche Temperatur) lässt sich die Formel $TEMP = 75 * V_{TSENS} - 2.7\,\mathrm{V} + \frac{55}{75}$ bestimmen, auch diese Formel ist als Kalibrierkurve gespeichert[4].

6.4 Das Modul *Sets*

6.4.1 Funktion und Aufbau

Das Sätze-Modul übernimmt die Speicherung und Verwaltung der Sätze. Es ist nicht mit den *BusControl*-Modulen (siehe Abschnitt 6.5) zu verwechseln, welche für die *Übertragung* der Sätze zuständig sind. Für die persistente Speicherung wird, wie in Abschnitt 2.2.3 erläutert, der EEPROM des Mikrocontrollers verwendet, beim Start des Moduls werden alle Werte für schnellere Lesezugriffe in den RAM geladen.

Die EEPROM-Hardware des TM4C1294NCPDT verfügt über einige Fehler, wie in [2, S. 52] beschrieben. Ein wesentlicher Fehler ist, dass Sprünge aus dem Flash in den ROM unter Umständen[5] den Controller zum Absturz bringen, wenn währenddessen der EEPROM aktiv ist. Da sich der Programmcode im Flash befindet, Teile der *driverlib* aber im ROM, sind solche Sprünge nicht ungewöhnlich. In der Initialisierungsroutine `SetsInitModule` wird daher nur die Flash-Versionen der *driverlib* verwendet. Die ROM-Version von `EEPROMInit` ist fehlerhaft, daher wäre dies bei dieser Routine ohnehin nötig gewesen [2, S. 51].

[4] Das Satz-System unterstützt nur Kurven der Form $y = mx + b$, obige Formel kann aber trivialerweise in diese Form gebracht werden

[5] In der Tat scheint es von der Quelladresse abzuhängen, ob der Fehler auftritt oder nicht.

6.4.2 Besonderheiten der Implementierung

Nach dem Einlesen des EEPROM-Inhaltes werden in der Routine `SetsInitModule` einige Werte in der RAM-Kopie angepasst. Dabei werden die Werte jedes Satzes auf seinen Standardwert, wie in dem jeweiligen Satz gespeichert, gesetzt und einige Zähler mit Null initialisiert. Zusätzlich wird die aktuelle Software-Revision aus dem Flash kopiert. Es wäre entsprechend nicht nötig, diese Werte im EEPROM zu speichern, da sie automatisch überschrieben werden. Wie in Abschnitt 2.2.3 dargelegt, wurde aber zugunsten einer einfacheren Implementierung auf diese Optimierung verzichtet.

Bei der Initialisierung des Moduls wird eine Callback-Funktion angegeben, die bei der Änderung des Wertes eines Satzes (Parameter 11) aufgerufen wird. Diese bekommt alten und neuen Wert übergeben, der Rückgabewert der Funktion wird schlussendlich im Satz gespeichert. Auf diese Weise können Sätze als nicht änderbar markiert werden (indem immer der alte Wert zurückgegeben wird), ebenso ist es möglich den Wert zu übernehmen, aber etwa ein anderes Modul über diese Änderung zu benachrichtigen. Die Funktion `SetsWriteSetValue` übergeht dieses Callback, auf diese Weise kann etwa das ADC-Modul die aktuelle Chip-Temperatur in einen Satz eintragen, der von außen nicht änderbar ist.

Eine weitere Besonderheit stellt Parameter 2 des Satzes 0 dar. Wenn dieser Parameter mit einem Wert ungleich Null beschrieben wird, soll das Modul möglichst auffällig blinken, um seine physische Position ermitteln zu können. Diese Funktion ist besonders sinnvoll, wenn mehrere gleichartige Module in einem Aufbau verwendet werden. Das Blinken wird durch die Funktion `SystemControlRainbow` gesteuert. Der Aufruf dieser Funktion ist direkt in das Sätze-Modul integriert, da Änderungen an Satz 0 nicht an das Callback weitergeleitet werden, das ansonsten für eine solche Implementierung die erste Wahl gewesen wäre.

Wenn ein Parameter eines Satzes geändert wird, wird die Änderung sofort in das EEPROM zurückgeschrieben; dazu wird die synchrone Funktion `EEPROMProgram` verwendet, die den Programmablauf anhält, bis der Schreibvorgang abgeschlossen ist. Bei der Wartezeit handelt es sich laut [1, Abs. 8.2.4.1] um eine nicht näher definierte Zeitspanne: „Writes to words within a block are delayed by a variable amount of time. [...] The variability ranges from the write timing of the EEPROM to the erase timing of EEPROM, where the erase timing is less than the write timing of most external EEPROMs.“ Genauere Angaben konnten dem Datenblatt nicht entnommen werden. Es ist auch ein asynchrone Funktion verfügbar, bei der nicht auf das Ende des Schreibens gewartet wird, sondern dies der Anwendung durch einen Interrupt mitgeteilt wird. Diese Funktion kann jedoch immer nur ein Wort in den EEPROM schreiben, sodass ein System zur Speicherung der ausstehenden Schreibvorgänge hätte entwickelt werden müssen. Außerdem wäre es dann möglich, dass der EEPROM prinzipiell während der Ausführung von beliebigen anderen Codes aktiv ist, was zusammen mit dem Fehler des Controllers vollständig verhindert hätte, dass Code aus dem ROM ausgeführt werden könnte. Es wurde daher die synchrone Variante verwendet; in Tests konnte auch keine signifikante Blockierung des Programms durch das Warten auf den EEPROM festgestellt werden.

6.5 Die Module *BusControl-Master* und *-Slave*

Das BusControl-System besteht, wie in Abschnitt 2.2.1 erläutert, aus einem Master- und einem Slave-Modul, die größtenteils unabhängig voneinander sind.

Beide Module nutzen den gleichen Hardware-Block, das I^2C-Modul mit der Nummer 1. Dieses besteht aus einem unabhängigen Master- und Slave-Teil, der gleichzeitig genutzt werden kann. Da beide Teile des Moduls die gleichen physischen Pins nutzen, werden die Daten ohne weitere externe Verdrahtung zwischen Master- und Slave-Modul übertragen, es steht allerdings für beide Module nur ein gemeinsamer Interrupt zur Verfügung. Der Interrupt-Handler ist daher im Hauptprogramm untergebracht, das den Interrupt an beide Module weiterleitet, wo er ausgewertet wird, falls er von dem jeweiligen Modul verarbeitet wird.

6.5.1 Das Master-Modul

Der I^2C-Master des TM4C1294NCPDT verfügt über einen bekannten Fehler, bei dem durch wiederholtes Lesen des Statusregisters unter Umständen das Fehlerbit in diesem Register nicht korrekt gesetzt wird [2, S. 44]. Aus diesem Grund wird in der Funktion `waitForI2cFinished` durch wiederholtes Lesen des *Interrupt*statusregisters gewartet, bis dort ein Interrupt durch das Ende der Übertragung signalisiert wird. Der eigentliche Interrupt ist dabei maskiert und wird nicht vom Prozessor verarbeitet, in dem entsprechenden Register ist er jedoch dennoch sichtbar. Zusätzlich wird die Ausführung des Codes in der Funktion kurz verzögert, ohne diese Pause ergaben sich teilweise Probleme, insbesondere unvollständige Übertragungen, mit dem I^2C-Master, besonders, wenn ein niedriger Takt, wie die aktuell verwendeten 100 kHz, genutzt wurde[6].

Sollte bei der Kommunikation mit einer Karte ein Fehler auftreten, weil etwa die Karte nicht antwortet[7], wird ein entsprechendes Fehlerpaket als Antwort gesendet. Sollte ein Fehlerpaket von einer Karte erhalten werden, wird es unverändert weitergegeben.

6.5.2 Das Slave-Modul

Das Slave-Modul besteht im Wesentlichen aus dem Interrupt-Handler für das I^2C-Modul, in dem Leseanforderungen beantwortet und Schreibanforderungen entgegengenommen werden. Wenn ein vollständiges Paket empfangen wurde, wird dies dem Hauptprogramm durch eine Callback-Funktion mitgeteilt. Das Hauptprogramm kann nun mit der Funktion `BusControlSlaveGetSendBuffer` einen Zeiger auf den Sendepuffer anfordern und die Antwort dort hinterlegen. Der Puffer wird dabei gesperrt, und weitere Leseanfrage bis zum Entsperren nur mit Nullbytes beantwortet. Auf diese Weise wird sichergestellt, dass keine veralteten oder unvollständigen Antworten übertragen werden. Das Entsperren

[6] Vermutlich wird der gesamte Takt des I^2C-Blockes geteilt, sodass bei niedrigen Taktraten der Block nicht zuverlässig in einen Ruhezustand zurückkehrt, bevor der nächste Befehl durch das Programm erteilt wird.

[7] Das Datenerfassungsmodul reagiert hier in jedem Fall mit einem Fehler. Die USB-Messkarte erkennt diesen Fehler teilweise nicht und sendet die Antwort der Anfrage davor erneut.

des Sendepuffers erfolgt durch die Funktion `BusControlSlaveSendBufferCommit`. Wenn durch das Slave-Modul beim Empfang ein Fehler, wie etwas eine falsche Checksumme, detektiert, wird automatisch ein entsprechendes Fehlerpaket als Antwort gesendet. Wenn ein Fehler erst in einer höheren Schicht erkannt wird, wie etwas ein nicht existierender Satz[8], so kann über die Funktion `BusControlSlaveSendErrorPackage` ein Fehlerpaket gesendet werden.

Bei der Initialisierung des Moduls wird die Adresse, die der I^2C-Slave verwenden soll, von sieben Pins des Busses gelesen. Dabei werden – gemäß der Spezifikation des Bussystems – einige Bits invertiert.

6.6 Das Modul *DMA*

Dieses Modul verwaltet ausstehende Direct Memory Access (DMA)-Transaktionen. Es ist dabei speziell auf das Spektrum-Modul zugeschnitten, dass aktuell als einziges DMA verwendet[9].

6.6.1 Verwendete Hardware-Module

Diese Modul verwendet die DMA-Hardware des TM4C1294NCPDT. Diese Hardware ermöglicht es, Daten im Speicher unabhängig von den sonstigen Tätigkeiten des Prozessors zu kopieren. In den meisten Fällen kann der Kopiervorgang vollständig parallel zu den Berechnungen des Prozessors ablaufen, sodass sich eine sehr hohe Effektivität ergibt. Der DMA-Controller verfügt über 32 Kanäle, die sich – neben ihrer Priorität – durch die möglichen Trigger unterscheiden. Jeder Kanal hat dabei bis zu 10 mögliche Triggerquellen, von denen aber immer nur eine gleichzeitig aktiv sein kann [1, Abs. 9.2.1]. Die Konfiguration der Kanäle wird im RAM des Controller abgelegt[10], dazu muss die Anwendung einen 1 kB großen Block reservieren und die Adresse dem DMA-Controller mitteilen. In dieser Konfigurationstabelle sind für jeden Kanal Quell- und Zieladresse des Kopiervorgangs, die Anzahl der zu kopierenden Elemente, sowie die Größe eines Elementes (es sind 1, 2 oder 4 Byte möglich) und die Schrittweite, in der Quell- und Zieladresse erhöht werden, hinterlegt. Zusätzlich wird auch ein als *Arbitration Size* bezeichneter Wert gespeichert, welcher angibt, wie viele Elemente von dem DMA-Controller in einem Schritt, ohne ein erneutes Triggern des Kanal, kopiert werden. Schlussendlich ist jede Konfiguration in zwei Varianten vorhanden, die beiden werden als *primary* und *alternate* bezeichnet. Jeder Kanal kann in verschiedenen Modi betrieben werden; die wichtisten Modi sind *STOP*, *BASIC* und *PINGPONG*:

[8] Das Datenerfassungsmodul reagiert in jedem Fall mit einem Fehler, wenn ein Satz nicht gefunden wurde. Einige der anderen Karten geben hier undefinierte Daten zurück.

[9] Neben dem USB-Modul, das jedoch den USB-DMA-Controller verwendet und von der *usblib* verwaltet wird.

[10] Nicht etwa in Registern, wie bei den anderen Peripherie-Modulen üblich.

BASIC Bei jedem Triggersignal wird die Menge an Elementen kopiert, die in *Arbitration Size* angegeben ist. Wenn alle Elemente kopiert wurden, wird der Kanal automatisch in den STOP-Modus überführt und ein Interrupt ausgelöst.

PINGPONG Dieser Modus funktioniert zunächst ähnlich wie der BASIC-Modus. Wenn jedoch alle Elemente kopiert wurden, fährt der Controller automatisch mit der *alternate* Kontrollstruktur fort, zusätzlich wird ein Interrupt ausgelöst, durch den die Anwendung die *primary* Struktur erneut beschreiben kann. Wenn auch die *alternate* Struktur den PINGPONG-Modus enthält, können auf diese Weise beliebig lange Kopieraktionen durchgeführt werden. Um den Vorgang zu beenden wird normalerweise für die letzten Elemente ein BASIC-Transfer verwendet, wodurch der Controller den Kanal am Ende des Transfers automatisch in den STOP-Modus überführt.

STOP Der Kanal ist gestoppt, es werden keine Daten kopiert[11].

Die Priorität der DMA-Kanäle untereinander ergibt sich aus ihrer Nummer. Zusätzlich ist es möglich, jeden Kanal in einen *high priority* Modus zu schalten, von dieser Möglichkeit wurde aber kein Gebrauch gemacht. Die Priorität eines Kanals wird immer nur zum Beginn einer Übertragung ausgewertet, danach bleibt der Kanal aktiv, bis seine *Arbitration Size* abgearbeitet wurde [1, Abs. 9.2.3]. Da die CPU den gleichen Speicherbus wie das DMA-Modul verwendet, kann zwischen diesen beiden ein Konflikt auftreten. Die CPU hat in diesem Fall immer Vorrang, sie kann auch einen laufenden DMA-Transfer innerhalb einer Arbitrierung unterbrechen. Zusätzlich verfügt der verwendete Mikrocontroller noch über einen speziellen USB-DMA-Controller, der ausschließlich mit dem USB-Modul verwendet werden kann, seine Priorisierung konnte dem Datenblatt nicht entnommen werden. In der Praxis zeigen Versuche, dass DMA-Transfers bei weitem die schnellste Möglichkeit sind, Daten zu kopieren und einfachen Kopierschleifen auf der CPU deutlich überlegen sind.

Eine Besonderheit des DMA-Controllers stellen die Interrupts dar. Der TM4C1294-NCPDT verfügt zwar über einen dedizierten DMA-Interrupt, dieser wird allerdings nur für Fehlerbenachrichtigungen verwendet. Interrupts, die über das Ende einer Kopieraktion informieren, werden in dem jeweiligen Peripherie-Modul ausgelöst, das aktuell als Trigger für den jeweiligen Kanal verwendet wird [1, Abs. 9.2.10]. Dies muss bei der Implementierung beachtet werden.

Verwendete DMA-Kanäle

Auf dem Datenerfassungsmodul werden zwei Kanäle des DMA-Controllers benutzt, beide dienen der Spektrenerfassung. Sobald die Wandler melden, dass ein neuer Datenpunkt verfügbar ist, müssen 24 bit an Dummy-Daten an das SPI-Modul übertragen werden, wodurch diese gesendet und gleichzeitig die Daten des Wandlers empfangen werden. Die empfangenen Daten müssen schlussendlich von dem SPI-Modul in den Empfangspuffer kopiert werden.

[11] Dennoch wird bei jedem Trigger ein DMA-Interrupt ausgelöst, welches das Ende eines Transfers signalisiert.

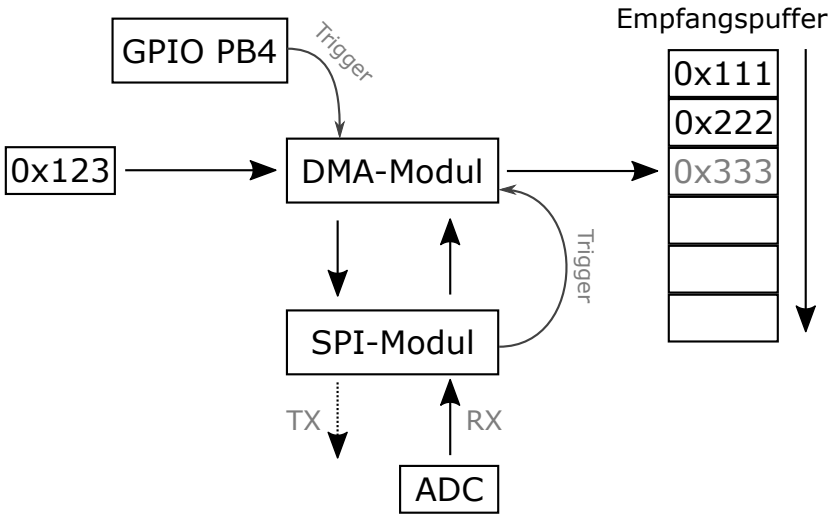


Abbildung 6.1: Datenfluss im DMA-Modul.

Es ist wichtig, zu beachten, dass das Kopieren der Dummy-Daten, die einen Transfer auslösen sollen, mit dem *Data Ready* Signal des Wandlers synchron sein muss, wie in Abschnitt 6.8 beschrieben. Da dieses Signal an Pin PB4 des Controllers anliegt, wird eine steigende Flanke an diesem Pin als Trigger für den ersten DMA-Kanal verwendet. Um dies zu erreichen, wird Port B des TM4C1294NCPDT so konfiguriert, dass eine entsprechende Flanke an Pin 4 einen DMA-Trigger auslöst, zusätzlich wird Kanal 5 des DMA-Controllers verwendet und auf GPIO-Port B als Trigger eingestellt. Als Übertragungsgröße werden jeweils zwei Elemente à 16 bit verwendet. Das SPI-Modul ist auf eine Wortbreite von 12 bit eingestellt, sodass jeweils die unteren 12 bit der beiden Elemente übertragen werden. Auf diese Weise können mit einer einzigen Flanke an PB4 genau 24 bit übertragen werden. Quell- und Zieladresse werden nicht inkrementiert, die Quelladresse zeigt dabei auf einen beliebigen Wert im Speicher, da sein Inhalt nicht relevant ist, die Zieladresse auf die Sendewarteschlange des SPI-Moduls.

Zum Übertragen der empfangenen Daten wird Kanal 10 verwendet, als Trigger wird das SPI-Modul konfiguriert. Dies sendet einen Trigger, wenn Daten im Empfangspuffer vorliegen [1, Abs. 17.3.8]. Als Übertragungsgröße werden vier Elemente verwendet, da ein Trigger gesendet wird, wenn diese Anzahl an Elemente im Empfangspuffer vorliegt[12]. Quelladresse ist der Empfangspuffer[13], Zieladresse ist der Empfangspuffer für Spektren im Speicher, diese Adresse wird entsprechend erhöht um die Elemente linear im Speicher abzulegen. Eine Übersicht zeigt Abbildung 6.1.

[12] Zusätzlich kann das Modul aber einen speziellen Trigger senden, um genau ein Element zu übertragen. Auf diese Weise wird sichergestellt, dass der Empfangspuffer immer vollständig geleert wird und nicht etwa drei Elemente zurückbleiben.

[13] Sende- und Empfangspuffer teilen sich eine Speicheradresse. Bei einem Schreibzugriff wird in die Sende-FIFO eingefügt, bei einem Lesezugriff aus der Empfangs-FIFO gelesen.

6.6.2 Besonderheiten der Implementierung

Der Speicherbereich, den der DMA-Controller benötigt, um die Konfiguration der Kanäle zu speichern, wird in diesem Modul unter dem Namen `dmaControlTable` reserviert. Dieser Speicherbereich muss immer auf eine 1024 B-Grenze ausgerichtet (*aligned*) sein, was durch die Präprozessordirektive `DATA_ALIGN` erreicht wird. Es ist zu beachten, dass diese Direktive für den TI-Compiler spezifisch ist, bei der Verwendung eines anderen Compilers muss hier das entsprechende Äquivalent verwendet werden.

Wenn ein Transfer gestartet werden soll, wird die Funktion `DmaStartTransfer` aufgerufen. Die übergebenen Daten werden in einer Struktur vom Typ `dmaChannel` gespeichert, um später zur Verfügung zu stehen. Wenn die angeforderte Datenmenge mit einem BASIC-Transfer übertragen werden kann, wird ein solcher gestartet, wenn nicht, wird ein PINGPONG-Transfer verwendet. Es ist dabei wichtig, schon vor dem Start *primary* und *alternate* Teil der Kontrollstrukur zu initialisieren, da am Ende des *primary*-Transfers sofort mit dem *alternate*-Transfer begonnen wird.

Da, wie oben beschrieben, die Interrupts, welche das Ende eines Transfers anzeigen, nicht in einem designierten Interrupt-Kanal erzeugt werden, sondern in dem jeweiligen Peripherie-Kanal, wird im DMA-Modul kein Interrupt-Handler verwendet. Die Interrupt-Handler der Peripherie-Blöcke, die typischerweise in anderen Modulen der Software zu finden sind, sind stattdessen so konfiguriert, dass sie bei einem anliegenden DMA-Interrupt die Funktion `DmaInterrupt` mit der entsprechenden Kanal-Nummer aufrufen. In der Funktion wird dann bei einem Transfer im PINGPONG-Modus der jeweils andere Teil der Konstrollstruktur neu initialisiert. Sollte ein Transfer vollständig abgeschlossen sein, wird eine Callback-Funktion aufgerufen, um die Anwendung über das Ende des gesamten Transfers zu informieren.

6.7 Das Modul *PulseGen*

6.7.1 Funktion und Aufbau

Aufgabe dieses Moduls ist die Erzeugung von sechs unabhängigen Pulsen auf den entsprechenden Leitungen des Bussystem. Zusätzlich muss für jeden Pulskanal ein *Enable*-Signal verwaltet werden, über das es möglich ist, einen Pulskanal vom Bus zu trennen, wenn die Erzeugung dieses Signals auf einer anderen Karte stattfinden soll. Pulsbreite und -verzögerung werden über Sätze der Karte eingestellt, ein Kanal gilt dabei als deaktiviert, wenn seine Pulsbreite auf 0 eingestellt wurde. Durch das in Abschnitt 6.7.2 dargestellte Verfahren bieten sowohl Verzögerung als auch Pulsbreite eine Auflösung von 24 bit. Da jedoch die Sätze für den *Wert*-Parameter nur 16 bit vorsehen, ist die Ausnutzung dieser Auflösung derzeit nicht möglich. Die verfügbaren 16 bit wurden daher auf die oberen – im Falle der Verzögerung – beziehungsweise unteren – im Falle der Pulsbreite – Bits abgebildet, um die typischen Anwendungsfälle abzudecken, eine Änderung dieser Aufteilung ist jedoch möglich[14].

[14] Die entsprechenden Konstanten finden sich in der Datei *config.h*.

6.7.2 Verwendete Hardware-Module

Die General Purpose Timer

Der verwendete Mikrocontroller verfügt über acht sogenannte General Purpose Timer (GPT), die für verschiedene Zwecke verwendet werden können. Jeder diese Blöcke kann als 32 bit Timer oder Zähler benutzt werden, alternativ kann er auch in zwei unabhängige 16 bit Timer aufgeteilt werden, die dann als A und B bezeichnet werden. Diese 16 bit Timer verfügen jeweils über einen 8 bit Vorteiler (*Prescaler*), der jedoch nicht direkt die Taktrate des Timers ändert[15], sondern die Rolle der höchstwertigen Bits (MSB) des Timers einnimmt, und komplett gelesen und geschrieben werden kann. Effektiv stehen damit zwei 24 bit Timer pro GPT-Block zu Verfügung.

Die Verwendung dieser Timer zur Erzeugung der Impulse bietet sich an, da die Timer unter anderem auch direkt den Zustand einiger Pins ändern können. Mit den vielfältigen Verwendungsmöglichkeiten der Timer geht jedoch eine hohe Komplexität einher. Da die ersten Versuche gezeigt hatten, das eine Software-Lösung zu langsam ist (siehe Abschnitt 2.2.2), soll ein großer Teil der Pulserzeugung in Hardware, unabhängig von dem restlichen Zustand des Prozessors, geschehen. Es wurden hier verschiedene Konzepte evaluiert, schlussendlich hat sich das Nachfolgende als realisierbar und den Anforderungen entsprechend bewährt, auch wenn es nicht vollständig in Hardware abläuft. Zu den verworfenen Konzepten gehörte, die Timer im *one-shot* Modus zu betreiben, um sie automatisch zu deaktivieren, dies ließ sich jedoch nicht realisieren, da immer zwei Triggerpunkte benötigt werden (Pulssignal An/Aus). Die Timer verfügen zwar über die Möglichkeit, neben dem Ablaufen des Timers noch das Erreichen eines beliebigen Zwischenwertes zu detektieren, hier ist aber nur das Auslösen eines Interrupts möglich, nicht das Umschalten eines Pins [1, Abs. 13.3.3.1]. Ein weiterer, verworfener Plan sah die Verwendung von zwei Timern je Pulskanal vor, deren Ausgabesignal mit Hilfe einer XOR-Gatters verschaltet werden sollte, um aus zwei steigenden Flanken die nötige steigende und fallende Flanke zu generieren. Diese Umsetzung hätte jedoch einen Umbau der Hardware erfordert, zudem hätte der Einsatz der XOR-Gatter eine zusätzliche Verzögerung von etwa 20 ns mit sich gebracht.

Pulserzeugung mit PWM

Neben anderen Betriebsmodi kann jeder einzelne Timer (A und B) in einem PWM-Modus betrieben werden. In diesem Modus wird der zugeordnete CCP-Pin auf den High-Pegel gezogen, sobald der Timer startet. Der Timer beginnt nun abwärts zu zählen, bis der sogenannte „Match-Wert“ erreicht wird, dann wird der CCP-Pin auf den Low-Pegel gezogen, der Timer zählt aber weiter. Sobald der Timer den Wert 0 erreicht, wird er mit dem „Load-Wert“ neu geladen [1, Abs. 13.3.3.5]. Auf diese Weise lässt sich ein PWM-Signal mit nahezu beliebiger Frequenz und Tastverhältnis erreichen. Abbildung 6.4 verdeutlicht dieses Prinzip.

[15] ... anders als in vielen anderen Mikrocontrollern, wie beispielsweise Atmels ATmegas

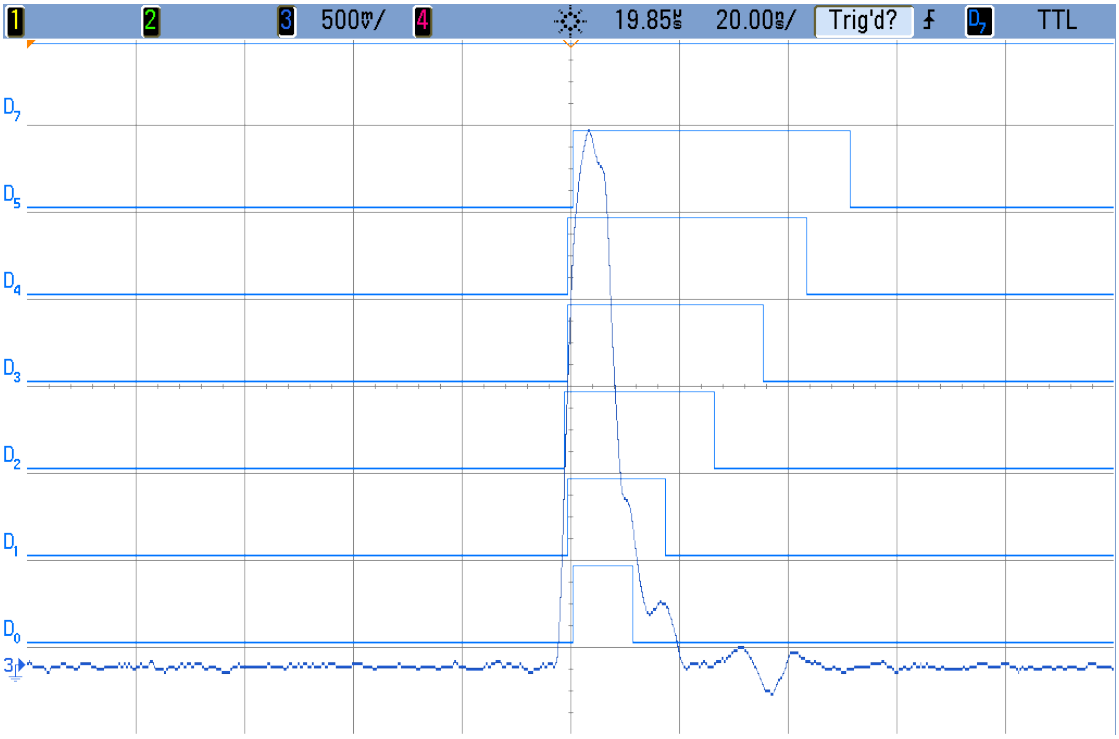


Abbildung 6.2: Pulse mit 1 bis 6 Takten Länge. Der Kanal 3 zeigt eine analoge Aufnahme des 1 Takt langen Pulses auf Kanal D0. Das eigentlich rechteckige Signal ist stark verschliffen.

Dieser Betriebsmodus deckt die Anforderungen an den Pulsgenerator recht gut ab, allerdings wäre eine Erzeugung des Puls-Signals am *Ende* des Timers wünschenswert. Zu diesem Zweck ist ein invertierter Modus vorhanden, in dem der CCP-Pin am Anfang auf den Low-Pegel gezogen wird und nach Erreichen des Match-Wertes den High-Pegel annimmt [1, Abs. 13.3.3.5]. Wie aus Abbildung 6.4 hervorgeht ist die minimale Pulsbreite 1 Takt, also 8.33 ns, sie wird erreicht, wenn der Match-Wert auf 0 gesetzt wird; der CCP-Pin ist dann genau einen Takt auf High-Potenzial[16]. Es ist daher zu beachten, dass immer genau ein Takt weniger angegeben werden muss, als die gewünschte Pulsbreite beträgt. Abbildung 6.2 zeigt Pulse verschiedener Längen. In der analogen Darstellung wird deutlich, dass sich die Hardware hier an ihrem Limit befindet. Die maximale Pulslänge und die maximale Verzögerung vor einem Puls wird durch die Auflösung des Timers bestimmt, mit 24 bit Auflösung (bei Ausnutzung des Prescalers) beträgt die maximale Verzögerung bei einem Takt von 120 MHz $^{2^{24}}/_{120\,\mathrm{MHz}} \approx 140\,\mathrm{ms}$.

Es ist zusätzlich zu beachten, dass die GPT-Blöcke zwar grundsätzlich in einem sogenannte „one-shot"-Modus betrieben werden können, bei dem sie automatisch nach einem Durchlauf angehalten werden, dies lässt sich aber nicht mit dem PWM-Modus kombinieren[17]. Um die Timer wieder zu deaktivieren wurde daher auf einen Interrupt zurückgegriffen, der auf der fallenden Flanke des PWM-Signals, am Ende des Pulses, ausgelöst wird. In der Interrupt-Routine wird der entsprechende Timer dann deaktiviert. Das Betreten und Verlassen einer Interrupt-Routine benötigt maximal $32 + 27 = 59$ Takte [19], die eigentliche Routine benötigt, wie Messungen und Zählungen der Assemblerbefehle zeigen, maximal 54 Takte[18]. Ein vollständiger Interrupt benötigt also maximal 113 Takte, wenn alle sechs Pulskanäle benutzt werden, sind alleine für die Interrupts also $6 \cdot 113 = 678$ Takte nötig. Um sicher zu stellen, dass alle Timer korrekt deaktiviert werden, bevor der Match-Wert erneut erreicht wird, ist daher eine minimale Verzögerung von 2000 Takten vor einem Puls vorgesehen, zusätzlich wurde die Priorität der entsprechenden Interrupts auf das zweithöchste Level angehoben, um sicherzustellen, dass die Timer auch dann deaktiviert werden, wenn durch den USB oder andere Quellen sehr viele Interrupts erzeugt werden. Abbildung 6.3 zeigt, wo sich diese Zeitspanne befindet. Schlussendlich muss noch sichergestellt werden, dass alle Timer synchron gestartet werden. Hierzu bietet der TM4C1294NCPDT eine Funktion, mit dem sich alle angegebenen GPT-Blöcke in einem Takt synchron neu starten lassen. Die entsprechenden Interrupts werden dabei nicht ausgelöst. [1, Abs. 13.3.5].

6.7.3 Besonderheiten der Implementierung

Im Verlauf der Implementierung zeigte sich, dass es zu einer Pegeländerung an den CCP-Pins der Timer kommen kann, wenn der invertierte Modus aktiviert wird. Die betreffenden Timer kurz zu aktivieren und sofort danach zu deaktivieren scheint das Problem zuverlässig zu unterdrücken, sodass dieses Verfahren benutzt wird.

[16] Abbildung 6.4 zeigt die nicht invertierte Situation, dort also auf Low-Potenzial
[17] Wie viele weitere, potenziell sehr interessante Kombinationen, etwa ein 32-bit PWM-Modus.
[18] Es wird hier die Flash-Version der *driverlib* verwendet, da die Zugriff drei Takte schneller ist.

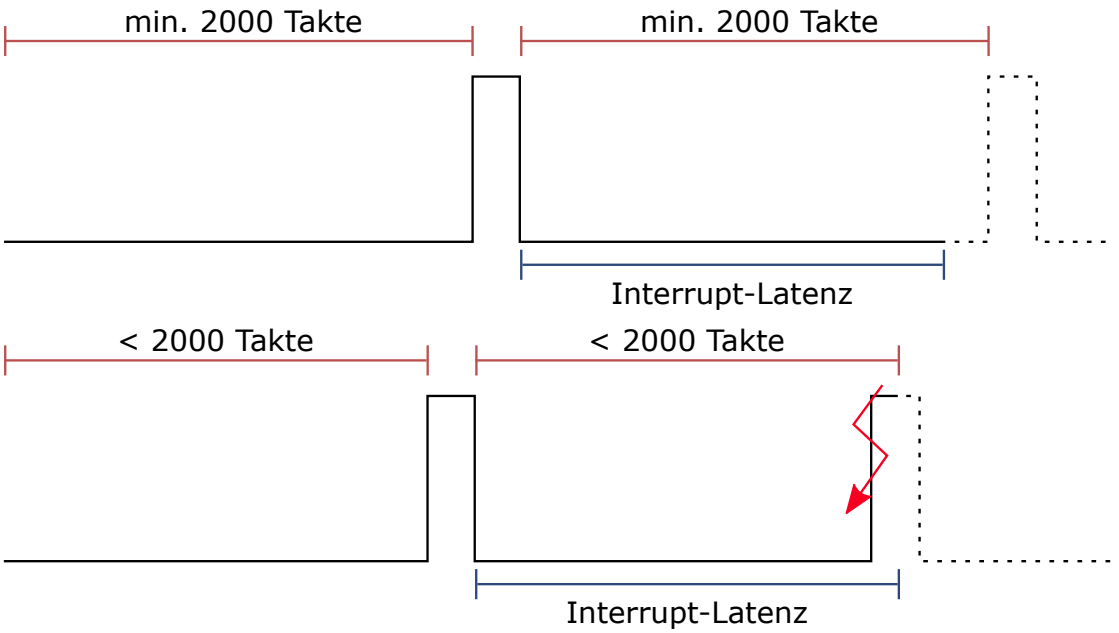


Abbildung 6.3: Auswirkungen der Interrupt-Latenz bei der Pulsgenerierung. Ohne ausreichende Verzögerung zwischen Start des Timers und Aktivierung des Pulses kann ein ungewollter zweiter Puls produziert werden.

In der Funktion `PulseGenSetup` werden die Timer für ein spezifisches Puls-Setup eingestellt. Nachdem die Timer auf diese Weise für ein spezielles Puls-Setup vorbereitetet wurden, kann dieses Setup – auch mehrfach – über die Funktion `PulseGenFire` ausgelöst werden. Hierzu werden die entsprechenden Timer aktiviert und dann über `TimerSynchronize` gleichzeitig zurückgesetzt, sodass alle Timer gleichzeitig zu zählen anfangen[19]. Das Starten der Timer wird dabei nicht direkt in der Funktion `PulsGenFire` durchgeführt, sondern in einem Interrupt-Handler für Pin PB4, der mit dem *Data Ready* Signal der Wandler zur Spektrenerfassung verbunden ist. Auf diese Weise wird sichergestellt, dass der relative Versatz zwischen den Messungen der Wandler und der Erzeugung der Pulse immer identisch ist. Eine Besonderheit stellt hierbei die Implementierung dar: Es wird keine Variable benutzt, um dem Interrupt-Handler zu signalisieren, dass ein Auslösen der Pulse jetzt gewünscht ist, es wird stattdessen die Interrupt-Quelle komplett deaktiviert[20]. Dies ist nötigt, da das *Data Ready* Signal mit einer Frequenz von 1 MHz ausgelöst wird. Würde der Interrupt dauerhaft aktiv bleiben, würde der Controller einen nicht unerheblichen Teil seiner Zeit mit der Verarbeitung des Interrupts verbringen.

[19] Durch die minimale Verzögerung von 2000 Takten vor dem Puls ist auch hier sichergestellt, dass nicht schon ein Puls ausgelöst wird, bevor die Synchronisierung durchgeführt wird.

[20] Es wird hierbei nur die Signalisierung seitens der GPIO-Ports deaktiviert. Der Kanal des Interrupt-Controllers bleibt aktiv, da dieser Interrupt auch vom Spektren-Modul für die DMA-Benachrichtigungen verwendet wird (siehe Abschnitt 6.6.1).

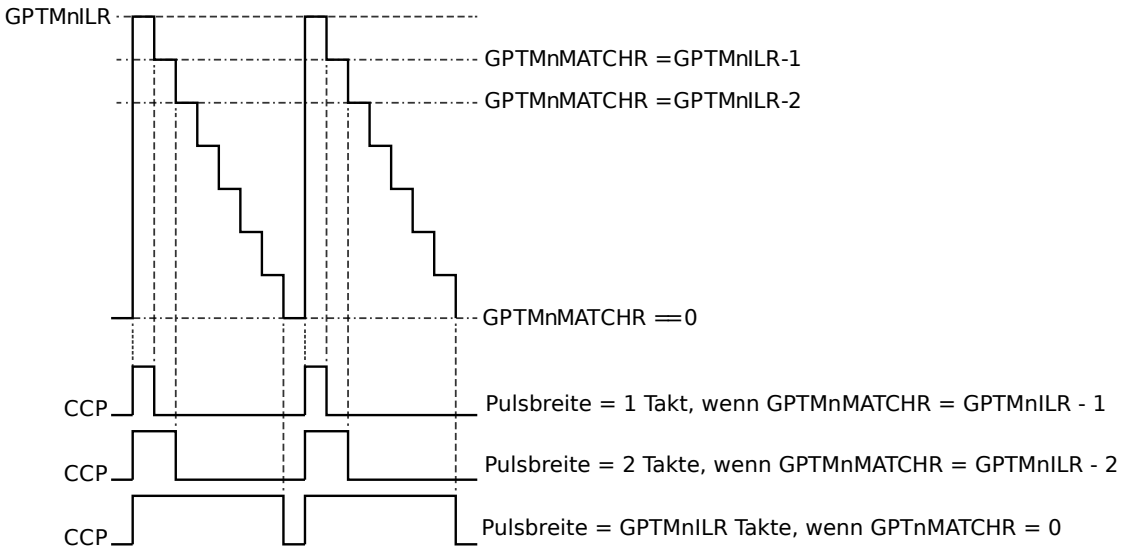


Abbildung 6.4: Beispiel zur PWM-Signal-Erzeugung. Der „Match-Wert“ wird hier als GPTMnMATCHR bezeichnet, der „Load-Wert“ als GPTMnILR. Quelle: [1, Abb. 31-7]

Die Implementierung der „AutoFire“-Funktion, bei dem die Pulse nach einer konfigurierbaren Zeitspanne automatisch ausgelöst werden, nutzt den Timer 1 im periodischen Modus. Wenn die Aussendung eines Triggers an das Spektrum-Modul aktiviert wurde, wird dazu Timer 5 genutzt. Er wird im *one shot* Modus betrieben[21], wodurch es nicht nötig ist, ihn, wie die anderen Timer, im entsprechenden Interrupt-Handler zu deaktivieren.

6.8 Das Modul *Spectrum*

6.8.1 Funktion und Aufbau

Das Spektrum-Modul steuert die Erfassung der Spektren von den verbauten Wandlern. Die Erfassung wird dabei durch einen Trigger ausgelöst. Nach der Erfassung wird das Spektrum mit einem Akkumulierungspuffer verrechnet, in dem bis zu 256 Spektren addiert werden können. Wenn die gewünschte Zahl an Summierungen erreicht wurde, wird der Inhalt des Akkumulierungspuffers per USB an den Computer übertragen. Da die Übertragung per USB möglicherweise mit einer nicht unerheblichen Verzögerung behaftet ist[22], wird ein doppelt gepuffertes System verwendet, bei dem Akkumulierungspuffer und Sendepuffer regelmäßig getauscht werden.

[21] Was hier möglich ist, da der PWM-Modus nicht benötigt wird.
[22] Hier wurde eine mögliche kurzfristige Überlastung des Host-Computers in Betracht gezogen.

6.8.2 Verwendete Hardware-Module

Zur Erfassung der Spektren werden ADCs des Typs ADS1675 von Texas Instruments verwendet. Diese übertragen die erfassten Daten per Serial Peripherial Interface (SPI) an den Mikrocontroller. Der Mikrocontroller agiert dabei als Master. Die ADCs werden mit einer Wandlungsrate von 1 MHz betrieben. Das Ende einer Wandlung wird durch eine Flanke des *Data Ready* Pins signalisiert, der mit dem Pin PB4 des TM4C1294NCPDT verbunden ist [29]. Nach diesem Signal können die Daten per SPI ausgelesen werden. Die maximale Taktrate des SPI-Moduls des TM4C1294NCPDT beträgt im Master-Modus 30 MHz [1, Abs. 17.3.1]. Da ein Datenpunkt 24 bit umfasst, ist diese Taktrate für die Übertragung der Daten ausreichend. Tests zeigen aber, dass eine nicht unerhebliche Verzögerung zwischen dem Anliegen des *Data Ready* Signals und der tatsächlichen Übertragung der Daten auftritt, die augenscheinlich durch das DMA-System verursacht wird. Abbildung 6.5 zeigt das Ergebnis des Tests. Es ist zu erkennen, dass die Übertragung des letzten Bits leicht mit dem nächsten *Data Ready* Signal überlappt. Es ist daher denkbar, dass die Übertragung des letzen Bits nicht sicher erfolgt, da möglicherweise schon das erste Bit des nächsten Datensatzes anliegt. Es wird aber vermutet, dass die Überlappung zu klein ist, um Probleme zu verursachen, zumal das Anlegen der Daten durch den Slave bei der steigenden Flanke erfolgt. Tests mit der realen Hardware konnte nicht durchgeführt werden, da diese nicht rechtzeitig zur Verfügung stand.

Zur Übertragung von genau 24 bit wird das SPI-Modul in einen Modus mit 12 bit Wortbreite geschaltet, es werden dann immer genau zwei Worte übertragen. Ein Transfer ist im verwendeten SPI-Protokoll immer bidirektional, es ist daher nicht möglich Daten zu empfangen ohne gleichzeitig zu senden. Da die gesendeten Daten nicht beachtet werden, wird als Dummy-Datum der Wert 0x123 übertragen, jeder andere Wert wäre ebenso denkbar.

Der TM4C1294NCPDT verfügt über vier identische SPI-Module (im Datenblatt allgemeiner als SSI-Module bezeichnet), von denen der verwendete Wandler des ersten Spektrum-Kanals mit dem Modul mit der Nummer 1 verbunden ist. Im Laufe der Arbeit konnte ein nicht dokumentierter Hardware-Fehler in diesem Modul ausgemacht werden, bei dem der DMA-Interrupt dauerhaft auftritt, sobald er einmal ausgelöst wurde. Auf diese Weise wird der Controller vollständig blockiert, sodass ausschließlich die entsprechende Interrupt-Routine ausgeführt wird[23]. Das Modul *SSI1* kann daher nicht verwendet werden, sodass in einer neueren Platinenrevision stattdessen das Modul *SSI0* verwendet werden soll. Der vorliegende Programmcode ist für diese neuere Version geschrieben. Das Vorliegen des Hardware-Fehlers kann mit dem beigelegten Programm *spi_dma_test* nachvollzogen werden, dort ist zu erkennen, dass der Interrupt-Handler ununterbrochen ausgeführt wird, was sich durch das ständige Umschalten des Pins PA3 zeigt. Im Internet konnten Beschreibungen desselben Problems anderer Anwender – jedoch keine Lösungen – gefunden werden [24].

[23] Im Errata ist ein ähnlicher Fehler beschrieben („SSI Transmit Interrupt Status Bit is not Latched“) [2, S. 67]. Dieser Fehler entspricht jedoch nicht genau dem hier beobachteten Verhalten und betrifft laut der Dokumentation alle SSI-Module, was bei diesem Fehler nicht der Fall ist.

[24] siehe [32, 33]

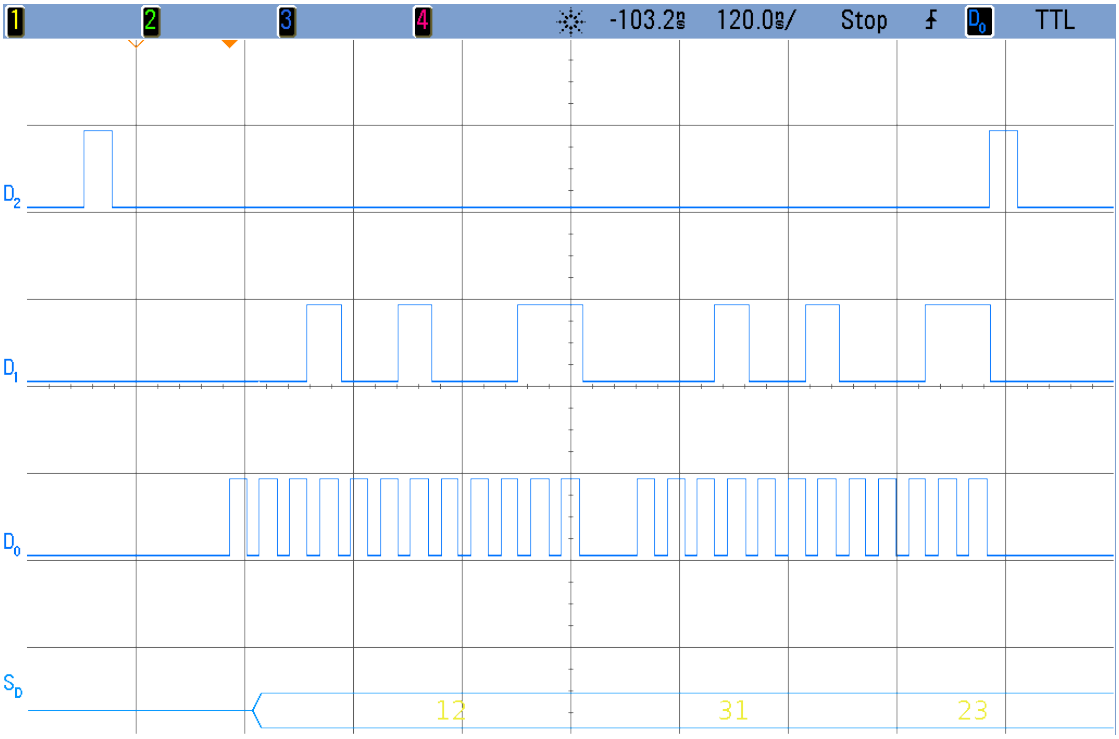


Abbildung 6.5: Timingprobleme bei der SPI-Übertragung. D2 zeigt den Trigger, D1 die Daten und D0 den Takt. Die Übertragung der Daten endet gleichzeitig mit dem nächsten Trigger. Zudem ist eine Verzögerung zwischen Trigger und Start der Übertragung sichtbar.

Als weitere Besonderheit wurde festgestellt, dass das SPI-Modul nur dann korrekt funktioniert, wenn das TX-Signal des Moduls (als *SSI0XDAT0* bezeichnet) auf einen physischen Pin geroutet wird. Wenn das Signal über das Pin-Muxing-System mit keinem Pin verbunden ist, werden auf dem *Empfangs*pin nur Nullen eingelesen. Auch hier wird ein Hardware-Fehler vermutet. Im Code wird daher das entsprechende Signal auf PA4 geschaltet, der bei der Verschaltung offen gelassen werden sollte.

Die Konfiguration der ADS1675 erfolgt ausschließlich über externe Pins, sie verfügen über keine internen Register oder ähnliches [29]. Die Konfigurationspins sind mit den Pins PN0-PN4 verbunden.

6.8.3 Besonderheiten der Implementierung

Die Erfassung eines Spektrum zu starten, erfordert im Wesentlichen, die entsprechenden, in Abschnitt 6.6.1 beschriebenen, DMA-Transfers zu starten. Zusätzlich wird noch die DMA-Trigger-Funktion des Pins PB4 aktivert. Diese Funktion kann nicht ständig aktiv bleiben, da ansonsten durch die kontinuierlich generierten Signale des ADC eine hohe Systemlast durch den Interrupt auftritt, der bei einem Kanal im STOP-Modus bei jedem Trigger generiert wird. Die Erfassung wird in der Funktion `SpektrumTrigger` gestartet, sofern vorher `SpectrumAquire` aufgerufen wurde oder die automatische Erfassung aktiv ist, und die letzte Erfassung bereits beendet wurde.

Nach Abschluss der Erfassung wird das Hauptprogramm durch eine Callback-Funktion benachrichtigt. Dies führt später zu einem Aufruf von `SpectrumProcess`, wodurch die im Puffer vorliegenden Daten mit dem Akkumulierungspuffer verrechnet werden. Diese Trennung ist nötig, da die Benachrichtigung über den Abschluss der Erfassung aus einem Interrupt-Handler erfolgt. Würde die vollständige Verarbeitung des Spektrums, die etwa eine Millisekunde erfordert, aus diesem Kontext erfolgen, könnte in dieser Zeit kein Interrupt gleicher Priorität ausgelöst werden. Nahezu alle Interrupts des Systems verwenden die niedrigstmögliche Priorität 255, sodass der Chip für diese Zeitspanne blockiert würde. Der Umweg über das Hauptprogramm, dass die Verarbeitung des Spektrums später außerhalb eines Interrupt-Kontextes anstößt, umgeht dieses Problem. Es handelt sich um das gleiche Konzept, das auch bei der Verarbeitung der empfangenden USB-Pakete und der Kommandos aus Satz 1 angewendet wird.

Das Senden der Spektren wird schlussendlich durch das USB-Modul durchgeführt. Sollte das Modul noch durch eine Übertragung beschäftigt sein, werden die aktuellen Daten verworfen und mit den nächsten Daten ein neuer Übertragungsversuch gestartet. Auf diese Weise wird sichergestellt, dass immer die aktuellsten Daten – sofern möglich – übertragen werden.

Optimierung durch Abrollen von Schleifen

Die Verarbeitung der Spektren wird in einer Schleife durchgeführt, die für jeden Datenpunkt ein Mal durchlaufen wird. Der Inhalt der Schleife ist sehr klein, wodurch die Laufzeit durch die Schleife selbst, also das Inkrementieren des Schleifenzählers und das Überprüfen der Abbruchbedingung, nicht zu vernachlässigen ist. Es wurde daher untersucht, ob eine

	Zeit (Takte)		Rel. Anteil	
Variante	*Copy*	*Add*	*Copy*	*Add*
ohne Abrollung	139333	154225	100%	100%
mit `UNROLL(10)`	99974	127248	70%	82%

Tabelle 6.2: Messungen zur Optimierung durch Abrollen von Schleifen. *Copy* bezeichnet einen Durchlauf, bei dem ausschließlich Werte kopiert wurden, *Add* einen, bei dem die Werte mit den bestehenden verrechnet wurden. Alle Messungen wurden im Release-Build mit aktivierten Compiler-Optimierungen durchgeführt. Größere Abrollfaktoren erzielten keine weitere Verbesserung.

Optimierung des Codes durch ein sogenanntes *Abrollen* der Schleife möglich ist. Dabei wird der Inhalt der Schleife beispielsweise in jedem Durchlauf zehnfach ausgeführt, bevor die Schleifenbedingung erneut geprüft wird [30]. Dies erhöht den Platzbedarf des Codes, kann aber insbesondere bei Schleifen, deren Inhalt sehr kurz ist, zu einer deutlichen Verbesserung des Laufzeitverhaltens führen. Es wurde daher untersucht, ob dies hier auch der Fall ist[25].

Das Abrollen der Schleife ist per Hand möglich, der verwendete Compiler ermöglicht jedoch auch ein automatisches Abrollen einer Schleife durch die Präprozessordirektive `UNROLL(x)`, wobei x angibt, wie häufig der Rumpf der Schleife wiederholt werden soll. Bei der automatischen Abrollung wird auch entsprechender Code erzeugt, der die letzten Elemente der Schleife bearbeitet, die auftreten können, wenn die Anzahl der Schleifendurchläufe nicht durch x teilbar ist[26]. Tabelle 6.2 zeigt das Ergebnis der Messungen. Es ist zu erkennen, dass die Optimierung besonders bei der *Add*-Schleife wirksam ist (-30%). Die *Copy*-Schleife, deren Rumpf etwas mehr Operationen enthält, profitiert weniger (-18%). Da dennoch eine deutliche Verbesserung zu erkennen ist, wurde die Abrollung im Code beibehalten.

6.9 Das Modul *USB*

6.9.1 Funktion und Aufbau

Das USB-Modul ist sehr umfangreich, aus diesem Grund wurde es in zwei Teile aufgeteilt. Die Dateien *device.c/h* enthalten den Code, welcher der Modulschnittstelle entspricht. Dieser Code verwendet seinerseits Code aus den Dateien *device_ internal.c/h*. Der dortige Code ist näher an der *usblib* angesiedelt und entspricht dort eher einer der in Abschnitt 3.2.2 erläuterten vordefinierten Klassen; er basiert stark auf dem *bulk_ device*-Beispielcode von Texas Instruments.

[25] Die Messungen basieren auf einer älteren Version des Codes, die einen leicht anderen Code im Rumpf der Schleife verwendete. Die Ergebnisse sind aber vergleichbar.

[26] Dies funktioniert auch, wenn, wie im hier vorliegenden Code, die Anzahl der Durchläufe erst zur Laufzeit bestimmt wird.

Die Schnittstelle zwischen Modul und Hauptprogramm besteht neben der, bei der Initialisierung anzugebenden, Callback-Funktion, über welche die Anwendung über neue Daten informiert wird, größtenteils aus zwei gemeinsam genutzten Datenspeichern in Form von zwei USB-Buffern.

Die *usblib* verwendet in den vordefinierten Geräteklassen ein ereignisorientiertes Modell zur Kommunikation mit der Anwendung, sodass auch die hier selbst implementierte Geräteklasse diese Kommunikationsmethode verwendet. Die Ereignisse werden dabei an die Anwendung über zwei Callback-Funktionen weitergeben, `rxCallback` und `txCallback`. Neben der Möglichkeit, diese Ereignisse direkt in der Anwendung zu verarbeiten, ist es möglich, sowohl in Sende- als auch in Empfangsrichtung einen *USB Buffer* einzufügen. Diese Strukturen stellen einen Ringpuffer dar, der zwischen dem paketorientierten USB und der bytestromorientierten Anwendung übersetzt. Die USB-Buffer sind Teil der *usblib* [6, 5.5]. Die USB-Buffer enthalten die Funktion `USBBufferEventCallback`, die als Callback-Funktion an eine USB-Geräteklasse übergeben werden kann. Diese Funktion zweigt Ereignisse, die den Puffer betreffen, ab, alle anderen werden an die Anwendung weitergeleitet[27]. Auf diese Weise wird der Puffer transparent eingefügt. Neben der Einbindung der Callback-Funktionen ist es nötig, vier Funktionen bereit zu stellen, mit denen ein USB-Paket gelesen und geschrieben, sowie der Füllstand der Sende- und Empfangswarteschlange des USB-Moduls abgefragt werden können.

6.9.2 Verwendete Hardware-Module

Es wird der USB-Baustein des TM4C1294NCPDT genutzt. Es ist dabei sowohl möglich, den im Controller eingebauten USB 1.1 Transreceiver, als auch einen externen Baustein nach dem ULIP-Standard zu verwenden, da diese Umschaltung transparent erfolgt. Alle wesentliche Interaktion mit der Hardware erfolgt durch die *usblib*, sowie teilweise durch die *driverlib*, sodass hier keine Besonderheiten zu beachten sind.

Bei Verwendung des USB-DMA-Moduls durch die *usblib* sind Transferraten von etwa 7.7 Mbit/s realisierbar, was recht nahe an dem theoretischen Maximum von 9.7 Mbit/s einer Full-Speed-Übertragung über Bulk-Pipes liegt [24, Tab. 2.7], vermutlich ist hier also noch eine Verbesserung durch die Verwendung des High-Speed-Modus von USB 2.0 möglich. Ohne Verwendung von DMA sind nur Transferraten von etwa 5 Mbit/s realisierbar, hier wird die Übertragung also durch den Prozessor limitiert.

6.9.3 Besonderheiten der Implementierung

Die Datei *device.c*

Der Code in der Datei *device.c* ist nicht sehr umfangreich. Es werden die beiden verwendeten USB-Buffer initialisiert, wobei die Funktionen `usbRxHanlder` und `usbTxHandler` als

[27] Die Verwendung einer solchen allgemeinen Funktion wird dadurch erleichtert, dass alle Geräteklassen einen allgemeinen Pointer vorsehen, der immer als erster Parameter an die jeweilige Callback-Funktion übergeben wird. In diesem Pointer kann ein Verweis auf den jeweiligen USB-Buffer gespeichert werden, sodass die Funktion `USBBufferEventCallback` für alle Instanzen gleichzeitig verwendet werden kann.

Callback-Funktion angegeben werden. Beim Empfang neuer Daten wird überprüft, ob ein nicht komplett gefülltes Paket erhalten wurde, was gemäß dem Protokoll das Ende einer Übertragung anzeigt. Wenn dies der Fall ist, wird die bei der Initialisierung des Moduls angegebene Callback-Funktion aufgerufen, damit die Anwendung die Daten verarbeiten kann.

Durch die zwischengeschalteten USB-Buffer ist nur die gesamte Anzahl der Bytes im Puffer bekannt und es kann nicht direkt bestimmt werden, wie viele Bytes in diesem Schritt empfangen wurden, was nötig ist, um das Ende der Übertragung zu erkennen. Dieses Problem wird gelöst, indem die Anzahl der im letzten Schritt vorhandenen Bytes zwischengespeichert wird. Aus der Differenz ergibt sich dann die gesuchte Menge der in diesem Schritt empfangenen Daten.

Die Übertragung der Daten zwischen Modul und Anwendung erfolgt durch die gemeinsam genutzten[28] USB-Buffer. Die Anwendung wird durch das Callback benachrichtigt, dass neue Daten vorliegen, und liest diese aus dem Empfangspuffer. Eine etwaige Antwort wird von der Anwendung in den Sendepuffer geschrieben, was automatisch, ohne weitere Interaktion mit dem USB-Modul der Software, das Senden der Daten per USB auslöst.

Die Datei *device_internal.c*

Die Datei *device_internal.c* stellt die Schnittstelle zwischen Anwendung und *usblib* dar. In der Datei enthalten ist eine Geräteklassendefinition, die stark an das *usb_bulk*-Beispiel von Texas Instruments angelehnt ist (siehe Abschnitt 3.2.2). Ein großer Teil der Datei besteht aus der Definition von Deskriptoren, wie sie in Abschnitt 4.2 erläutert wurden. Neben den dort beschriebenen Deskriptoren ist auch noch ein „Microsoft Extended Feature Descriptor“ vorhanden, der über den gleichen Mechanismus wie die WCID-Daten abgefragt wird (siehe Abschnitt 5.2.2). Dieser Deskriptor enthält einige weitere Metadaten über das Gerät, die von Windows bei der Erkennung in der Registry ablegt werden, darunter die *Device Interface GUID*, mit der das Gerät später im Treiber erkannt wird. Zusätzlich werden noch ein Name und ein Icon definiert, das von der Version von WinUSB ab Windows 8 im Geräte-Manager angezeigt wird[29]. Die Deskriptoren sind dabei als `const` deklariert, da sich ihr Inhalt nicht ändern kann. Ein Ausnahme stellt der String-Deskriptor mit der Nummer 3 da, der die Seriennummer des Gerätes enthält. Da die Seriennummer in Satz 0 gespeichert ist, und vom Anwender geändert werden kann, ist dieser nicht als `const` markiert. Beim Start der Anwendung wird der Inhalt des Deskriptors mit dem Wert aus Satz 0 überschrieben, der Standardwert „00000“ wird nur verwendet, wenn der EEPROM keine Daten enthält, also bei einer fabrikneuen Karte[30].

Eine weitere Besonderheit der Implementierung stellt die Auflistung der String-Deskriptoren im Array `usbStringDescriptors` dar, die sich aus Platzgründen in der separaten Datei *usb_string_table.h* befindet, deren Inhalt über die Präprozessordirektive `#include` eingebunden wird. Die *usblib* verfügt über eine Funktion, welche Anfragen des Hosts nach

[28] erkennbar an der `extern` Deklaration im Header

[29] In früheren Versionen wird das Gerät nur allgemein als „WinUSB-Gerät“ bezeichnet

[30] Da Windows die Treiberinstallation mit der Seriennummer eines USB-Gerätes assoziiert, ist daher nach dem Setzen der Seriennummer eine neue (automatische) Installation des Treibers nötig.

den String-Deskriptoren automatisch ohne Interaktion der höheren Schichten beantwortet werden [6, Abs. 2.23.1.3]. Zu diesem Zweck benötigt die *usblib* dieses Array mit allen String-Deskriptoren. Das erste Element des Arrays enthält einen Verweis auf String-Deskriptor 0, der laut USB-Spezifikation die Liste der unterstützten Sprachen enthält, sodass die *usblib* auf diese Information zugreifen kann. Wenn auf diese Weise mehrere Sprachen unterstützt werden, müssen die Deskriptoren aller Sprachen danach linear in einem Array abgelegt werden, also etwa zunächst drei Deskriptoren in der Sprache 1 auf den Indices 1 bis 3, dann drei Deskriptoren in der Sprache 2 auf den Indices 4 bis 6 usw.

Für die Unterstützung der WCID-Funktion ist es, wie in Abschnitt 5.2.2 beschrieben, nötig, auf eine Anfrage nach dem Deskriptor 0xEE (dezimal 238) mit der Sprach-ID 0x0000 zu reagieren. Eine solche Anforderung ist in der *usblib* zunächst nicht vorgesehen. Es wäre möglich, den Code der *usblib* entsprechend zu ändern und etwa ein Callback vorzusehen, über das die Anwendung zusätzliche Deskriptoren bereitstellen kann. Da aber eine Modifizierung der Bibliothek eine spätere Wartung der Software erschwert hätte, weil immer auf die Verwendung der korrekt modifizierten Version hätte geachtet werden müssen, wurde eine alternative Lösung, die ohne eine Modifikation auskommt, gesucht. Die implementierte Lösung besteht aus einer sehr umfangreichen String-Deskriptortabelle. Dabei sind in den ersten Einträgen zunächst 5 „normale“ Deskriptoren für die erste Sprache, in diesem Fall Deutsch, Code 0x0409, vorhanden. Danach folgen $238 - 5 = 233$ Verweise auf das Feld `usbStringDeskriptorEmpty`, um die Tabelle bis zum Index 238 aufzufüllen. Danach folgenden die Einträge für die Sprach-ID 0x0000, die aus 237 leeren Verweisen bestehen, auf die dann an Indexposition 238 der Verweis auf den WCID-Deskriptor folgt. Der leere Deskriptor `usbStringDeskriptorEmpty` entspricht dabei nicht vollständig dem USB-Standard, da auf einen nicht vorhandenen Deskriptor mit einem *Stall* an Endpoint 0 reagiert werden müsste [24, Abs. 2.9.1]. Die hier verwendete Variante mit einem Deskriptor der Länge 0 scheint aber in der Praxis ebenso zu funktionieren[31].

Die unteren Ebenen der *usblib* geben Informationen und Anfragen des Host in Form von Callbacks an die oberen Schichten weiter. In den meisten Fällen wird das entsprechende Ereignis direkt an die entsprechenden Callbacks der Anwendung weitergegeben, wo es entweder direkt, oder aber von einem zwischengeschalteten USB-Buffer verarbeitet wird. Besondere Beachtung verdient nur noch die Funktion `usbHandleRequestCb`, die von der *usblib* aufgerufen wird, wenn ein unbekannter Request empfangen wird. In dieser Funktion werden daher die beiden Vendor-Requests implementiert, die von dem Gerät unterstützt werden. Zum einen handelt es sich dabei um den im WCID-Deskriptor angegebenen Request, der vom Betriebssystem zur Abfrage von weiteren Deskriptoren verwendet wird, zum anderen ist ein Request implementiert, mit dem der Treiber anzeigen kann, wenn eine Verbindung auf- oder abgebaut wird. Beide Deskriptoren werden in dieser Funktion behandelt. Wenn ein Request nicht verstanden wird, wird ein gemäß der Spezifikation *Stall* ausgelöst, um dem Host dies zu signalisieren [24, Abs. 2.9.1].

[31] Dabei ist der Deskriptor selbst eigentlich ungültig, da die Längenangabe auch immer das Byte, welche die Längenangabe enthält, mit einschließt. So kann ein Deskriptor eigentlich nicht 0 Byte groß sein, da diese Information selber schon 1 Byte umfasst.

6.10 Das Hauptprogramm

Ein wesentlicher Teil des Codes der Anwendung ist in den Modulen der Software enthalten, das Hauptprogramm in der Gestalt der Datei *main.c* enthält hauptsächlich Code, der die einzelnen Module verbindet. Neben dieser Datei gehört noch die Datei *tm4c1294ncpdt_startup_ccs.c* zu dem Hauptprogramm. Diese Datei enthält insbesondere die Tabelle aller Interrupt-Vektoren des TM4C1294NCPDT und verknüpft diese mit den entsprechenden Interrupt-Handlern, die in den meisten Fällen in den jeweiligen Modulen zu finden sind. Zwei Ausnahmen stellen in dieser Hinsicht die Interrupt-Handler für GPIO-Port B und das I^2C Modul mit der Nummer 1 dar, da sich die Interrupt-Handler für diese Peripherie-Bausteine im Hauptprogramm finden. Dies ist nötig, da die Interrupts von jeweils zwei Modulen verarbeitet werden müssen, das Hauptprogramm leitet die Interrupts durch Aufrufe der entsprechenden Funktionen an beide Module weiter.

Der Haupteinstiegspunkt der Anwendung, die Funktion `main`, initialisiert zunächst alle Module der Software durch Aufruf der entsprechenden Initialisierungsfunktionen, wobei die Abhängigkeiten der Module in der Reihenfolge der Initialisierung widergespiegelt werden. Der Rest der Funktion `main` besteht aus einer Endlosschleife, in der auf das Vorliegen eines neuen USB-Paketes, eines Kommandos aus Satz 1 oder eines Spektrums gewartet wird. Wenn ein solches Ereignis eintritt, wird eine entsprechende Verarbeitung vorgenommen. Durch diese Trennung von Empfang und Signalisierung sowie Verarbeitung eines Ereignisses wird sichergestellt, dass die rechenintensive Verarbeitung eines Ereignisses nicht in einem Interrupt-Handler stattfindet und auf diese Weise der Chip für eine längere Zeit blockiert wird. Bei der Erkennung eines neuen USB-Paketes werden zwei Zähler mit vorhandenen und schon verarbeiteten Paketen verwendet, auf diese Weise wird verhindert, dass ein Paket, das während der Verarbeitung des vorangegangenen Paketes eintrifft, übersprungen wird. Bei den Spektren kann dieses Problem nicht auftreten, da kein Spektrum erfasst werden kann, bevor das letzte verarbeitet wurde; bei den Kommandos an Satz 1 wurde dieser Fall nicht beachtet, da diese Kommandos nur sehr selten benutzt werden[32].

Ein weiterer Teil des Hauptprogramms ist die Implementation der zahlreichen Callback-Funktionen, über die die Module Informationen an die Anwendung zurückgeben. In den meisten Fällen werden dabei nur Signale gesetzt, um das Ereignis dann, wie oben beschrieben, in der Hauptschleife zu bearbeiten. Ausnahmen stellen hier die Funktionen `bcCallback` und `setsValueChangedCallback` dar. Beim ersten, dem Callback beim Empfang eines *BusControl*-Paketes, ist eine sofortige Verarbeitung nötig, da der Master unverzüglich nach dem Senden der Anfrage mit dem Lesen der Antwort beginnt, sodass diese möglichst schnell bereit stehen muss. Die zweite Funktion, die beim Ändern eines Wertes eines Satzes aufgerufen wird, muss dem Satz-Modul sofort eine Antwort geben, ob der Wert geändert werden darf oder nicht, sodass auch hier eine sofortige Verarbeitung unumgänglich ist. Da der Code dieser Funktion aber sehr schnell ausgeführt wird, stellt dies kein Problem dar.

[32] Sollte dennoch ein Kommando eintreffen, während noch ein anderes verarbeitet wird, wird allerdings eine Warnung ausgegeben.

7 Zusammenfassung

Alle in Kapitel 1 definierten Ziele konnten erreicht werden. Tabelle 7.1 stellt die technischen Daten von USB-Messkarte und Datenerfassungsmodul im Vergleich dar. Es wird deutlich, dass die Datenerfassungskarte die alte USB-Messkarte in allen Punkten deutlich übertrifft.

Die gemessene Übertragungsrate per USB liegt recht nahe an der theoretischen maximalen Nettodatenrate von 9.7 Mbit/s der USB 1.1 Schnittstelle [24, Tab. 2.7], es ist daher anzunehmen, dass sich hier eine weitere Verbesserung mit der USB 2.0 Schnittstelle einstellt. Da die Hardware der Datenerfassungskarte in diesem Punkt nicht funktionsfähig war, konnten hier jedoch keine Messungen durchgeführt werden.

Die erreichte minimale Pulsbreite von 8.33 ns übertrifft die aktuellen Anforderungen deutlich. Durch Optimierung der Ansteuerung konnten zudem die Pulsfilter, die auf dem ersten Entwurf des Datenerfassungsmoduls vorgesehen waren, entfernt werden.

Die Erfassung der Spektren ist nun mit deutlich größerer vertikaler wie horizontaler Auflösung möglich. Gleichzeitig wurde die maximale Länge eines Spektrums erhöht. Zur Entlastung des Messcomputers können die Spektren nun schon auf der Datenerfassungskarte durch eine Summierung vorverarbeitet werden. Durch ein dreifach gepuffertes System wird dabei sichergestellt, dass Spektren auch bei kurzzeitiger Überlastung des Host-Computers lückenlos aufgezeichnet werden. Dies wird auch durch ein neues Triggersystem vereinfacht, welches das Aussenden von Pulsen und Aufzeichnen von Spektren ohne Interaktion des Computers ermöglicht. Durch die Phasensynchronisation zwischen A/D-Wandlern und Pulsen sowie Spektren wird die Genauigkeit des Systems weiter erhöht.

Nicht umgesetzt werden konnte die Funktion, zwei Spektren parallel auf einem Datenerfassungsmodul aufzuzeichnen, da der zur Verfügung stehende Speicher des Microcontrollers hierzu nicht ausreichte. Da eine vollständig funktionsfähige Version des Datenerfassungsmoduls nicht rechtzeitig zu Verfügung stand, war es nicht möglich, alle Funktionen auf der

	USB-Messkarte	Datenerfassungsmodul
USB-Übertragungsrate	666 kbit/s	7.7 Mbit/s
Auflösung Pulse	1 µs	8.33 ns
minimale Pulsbreite	3 µs	8.33 ns
maximale Länge eines Spektrums	2000 Punkte	15 000 Punkte
Auflösung der Spektren	16 bit	24 bit
Abtastrate der ADCs	90.9 kSa/s	1 MSa/s
Erfassungsrate der Spektren	≈ 8 Spektren/s	≈ 50 Spektren/s

Tabelle 7.1: Vergleich von USB-Messkarte und Datenerfassungskarte

realen Hardware zu testen. Die externen Bausteine wurden daher gemäß ihrer Spezifikation mit Funktionsgeneratoren und Oszilloskopen simuliert.

Durch die Modularisierung der Software können Teile der Software, wie etwa der *BusControl*-Slave problemlos auf zukünftigen Karten weiterverwendet werden. Da Master- und Slave-Teil des *BusControl*-Systems getrennt wurden, ist es auch möglich, ein Datenerfassungsmodul rein als Slave im Bussystem einzusetzen. Auf diese Weise können auch zwei Datenerfassungsmodule gleichzeitig verwendet werden, was die Aufzeichnung von zwei parallelen Spektren ermöglicht und das oben genannte Problem entschärft.

Die Installation der Treiber wird durch den Wegfall von INF-Dateien vereinfacht, da der Treiber von Windows Update heruntergeladen werden kann bzw. im Fall von Windows 8 und neuer schon in der Installation des Betriebssystems enthalten ist. Die neugeschriebene Treibersoftware ist durch die Verwendung der Sprache C# robuster gegenüber Abstürzen des Steuerungsprogramms und ermöglicht eine einfachere Einbindung in *LabView*. Da die neue Treibersoftware auch die alte USB-Messkarte vollständig unterstützt, konnte eine Fallunterscheidung vermieden und die Kompatibilität der Systeme weiter erhöht werden.

Schlussendlich wurde die Software der Datenerfassungskarte ausgiebig dokumentiert, wodurch Wartbarkeit und Wiederverwendbarkeit sichergestellt wurden.

Zusammenfassend stellt die Datenerfassungskarte damit eine deutlich leistungsfähigere Alternative zu der USB-Messkarte dar, die in allen wesentlichen Punkten zu ihr kompatibel ist.

A Verwendete Hardware

Software-Modul	Peripherie-Module	Pins	Interrupts
ADC/GPIO	ADC0 (Sequencer 0, 3)	PE0-3, PK0-3 (ADC)	ADC0Seq0,
	Timer 0 A/B	PC4-6, PH0-3, PQ3 (GPIO)	ADC0Seq3
		PC7 (GPIO-Enable)	Timer0A
BusControl Master	I2C1 (Master)	PG0 (SCL), PG1 (SDA)	I2C1
BusControl Slave	IC21 (Slave)	PG0 (SCL), PG1 (SDA)	I2C1
		PA6-7, PF0-4 (Adresse)	
Logging	UART2	PD4 (RX/in), PD5 (TX/out)	UART2
Sets	EEPROM	-	-
PulseGen	Timers 2-4 A/B (Pulse)	PM0-5 (Pulse)	Timer2-4 A/B,
	Timer 5 A (Trigger)	PA2-5, PM6-7 (Pulse Enable)	Timer 5 A,
	Timer 1 (AutoFire)	PB4 (Date Ready)	GPIO Port B
USB	USB0	PL6-7 (Full-Speed)	USB0
		PP1-5, PL0-5, PB0-3	
		(High-Speed)	
System	SysTick	PK5-7 (RGB-LED)	SysTick
Spectrum	SSI0	PA2 (Clock), PA5 (RX),	SSI0, GPIO Port B
		PA4 (TX, nicht verbinden)	
DMA	DMA controller	-	-

Literaturverzeichnis

[1] Texas Instruments, Hrsg., *Tiva TM4C1294NCPDT Microcontroller*, 2014.

[2] Texas Instruments, Hrsg., *Tiva C Series TM4C129x Microcontrollers Silicon Revisions 1, 2, and 3 Silicon Errata*, 2015.

[3] http://www.ti.com/tool/ccstudio

[4] http://www.ti.com/tool/sw-tm4c

[5] Texas Instruments, Hrsg., *TivaWare Peripheral Driver Library*, 2015

[6] Texas Instruments, Hrsg., *TivaWare USB Library*, 2015

[7] http://www.linux-usb.org/usb.ids

[8] https://msdn.microsoft.com/en-us/library/windows/hardware/hh450799%28v=vs.85%29.aspx

[9] https://msdn.microsoft.com/en-us/library/windows/hardware/ff554721%28v=vs.85%29.aspx

[10] https://msdn.microsoft.com/en-us/library/windows/hardware/ff540196%28v=vs.85%29.aspx

[11] https://msdn.microsoft.com/en-us/library/windows/hardware/ff540046%28v=vs.85%29.aspx#winusb

[12] http://sourceforge.net/projects/libusbdotnet/

[13] https://msdn.microsoft.com/en-us/library/windows/hardware/ff541224%28v=vs.85%29.aspx

[14] https://github.com/pbatard/libwdi/wiki/WCID-Devices

[15] http://www.albahari.com/nutshell/linqbridge.aspx

[16] http://zone.ni.com/reference/de-XX/help/371361J-0113/lvconcepts/using__net_with_labview/

[17] http://digital.ni.com/public.nsf/allkb/32B0BA28A72AA87D8625782600737DE9

[18] http://digital.ni.com/public.nsf/allkb/F326A891E72073B486256EEC0006E019

[19] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka16366.html

[20] https://msdn.microsoft.com/en-us/library/windows/hardware/ff549553%28v=vs.85%29.aspx

[21] Intel et al., *Universal Serial Bus Specification Revision 2.0*, 2000

[22] USB Implementers Forum *Universal Serial Bus Test and Measurement Class Specification (USBTMC) Revision 1.0* 2003

[23] https://en.wikipedia.org/wiki/In-band_signaling

[24] H. J. Kelm (Hrsg.), *USB 2.0 Studienausgabe, 4. Auflage*, Franzis Verlag, Poing, 2006

[25] J. Axelson *USB 2.0 Handbuch für Entwickler*, REDLINE, Heidelberg, 2007

[26] https://de.wikipedia.org/wiki/Universal_Serial_Bus

[27] http://www.usb.org/developers/tools/comp_dump

[28] http://www.7-zip.de/

[29] Texas Instruments, Hrsg., *4MSPS, 24-Bit Analog-to-Digital Converter ADS1675*, 2010

[30] https://de.wikipedia.org/wiki/Loop_unrolling

[31] https://msdn.microsoft.com/en-us/library/windows/hardware/ff550419%28v=vs.85%29.aspx

[32] https://e2e.ti.com/support/microcontrollers/tiva_arm/f/908/t/361916

[33] https://e2e.ti.com/support/microcontrollers/tiva_arm/f/908/t/437340

Alle Online-Quellen wurden zuletzt am 12. August 2015 abgerufen.