HELMUT SCHMIDT
UNIVERSITÄT
Universität der Bundeswehr Hamburg

**Henning Wolf**

# Ermöglichen einer Positionsbestimmung mit dem PRHS Framework

# Enabling Position Detection with the PRHS Framework

Bachelor-Arbeit

Fakultät für Elektrotechnik

| | |
|---|---|
| Studiengang: | Elektrotechnik und Informationstechnik |
| Matr.-Nr. | 874320 / ET2014 |
| Übernahme: | 17. Oktober 2016 |
| Betreuer: | Univ.-Prof. Dr. phil. nat. habil. Bernd Klauer |
| Weiterer Prüfer: | Univ.-Prof. Dr.-Ing. habil. Udo Zölzer |

# Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst, keine anderen als die im Quellen- und Literaturverzeichnis genannten Quellen und Hilfsmittel, insbesondere keine dort nicht genannten Internet-Quellen benutzt, alle aus Quellen und Literatur wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe und dass die auf einem elektronischen Speichermedium abgegebene Fassung der Arbeit der gedruckten entspricht.

Hamburg,

| ..... | ..... |
|---|---|
| (Datum) | (Unterschrift) |

# Inhaltsverzeichnis

**Abkürzungsverzeichnis**

**FPGA** Field Programmable Gate Array

**CLB** Configurable Logic Block

**FF** FlipFlop

**LUT** Look-Up-Table

**IOB** I/O-Block

**PRHS** Partial Reconfigurable Heterogeneous System

**CPU** Central Processing Unit

**KDE** K Desktop Environment

**GNOME** GNU Object Model Environment

**TCP/IP** Transmission Control Protocol/Internet Protocol

**UUCP** Unix to Unix Copy

**GPS** Global Positioning System

**GPSd** Global Positioning System-Daemon

**NAVSTAR** Navigation System with Time and Ranging

**SPI** Serial Peripheral Interface

**UART** Universal Asynchronous Receiver Transmitter

**VHDL** Very High Speed Integrated Circuit Hardware Description Language

**NMEA** National Marine Electronics Association

**SMA** Sub-Miniature-A

**AIS** Automatic Identification System

**Daemon** Disk and Execution Monitor

**DGPS** Differential Global Positioning System

**PID** Process-Identification-Data

**ARM** Advanced RISC Machines

**IRQ** Interrupt Request

**POSIX** Portable Operating System Interface

**3DF** 3D-Fix

**PPS** Pulse-per-Second Signal

**MEMS** Microelectromechanical System

**CPOL** Clock Plarity

**CPHA** Clock Phase

**FIFO** First In-First Out

**MSB** Most Significant Bit

**GGA** Global Positioning System Fix Data

**MISO** Master In-Slave Out

**MOSI** Master Out-Slave In

**SS** Slave Select

**SCK** Serial Clock

**GND** Chassis Ground

**TUI** Text User Interface

**L4PRHS** Linux for Partial Reconfigurable Heterogeneous System

**BIOS** Basic Input/Output System

# 1 Einleitung

Die Technologieentwicklung, im speziellen die der Computer, kann auf eine rasante Entwicklung in den letzten Jahrzehnten zurückblicken. Jedoch ist diese Technik mittlerweile so hochentwickelt, dass es mit zunehmender Zeit schwieriger wird, die Bauteile noch schneller, kleiner und effizienter herzustellen, denn auch hier gibt es Grenzen. So liegt es an den Entwicklern neue Wege zu finden, um diese Beschränkungen zu umgehen.
So kam es zur Entwicklung der FPGAs, welche es nun ermöglichen sollten, komplexe Logikschaltungen rekonfigurierbar herzustellen, sodass diese jederzeit erneuert, verändert und damit den Gegebenheiten und Anforderungen angepasst werden konnten. Durch die Optimierung während der Laufzeit können Kosten gesenkt und Ressourcen geschont werden. Mittlerweile sind moderne FPGAs mit genügend Recourcen ausgestattet um eine große Anzahl an Logikgattern zu berechnen und sind in der Lage komplexe Rechenoperationen durchzuführen.
Genau diese Entwicklung nutzte die Professur für Technische Informatik an der Helmut-Schmidt-Universität / Universität Hamburg der Bundeswehr, um im Rahmen dieser ein eigenes System zu entwickeln und auf dem Gebiet zu forschen. Das dort genutzte Partial Reconfigurable Heterogeneous System (PRHS)-Framework ermöglicht es, eine Nachbildung von Rechnern auf vielen verschiedenen FPGA-Boards zu nutzen. Neben den verschiedenen Hardwaremodulen wird als Betriebssystem das sogenannte Linux for Partial Reconfigurable Heterogeneous System (L4PRHS) genutzt.

## 1.1 Ziel der Arbeit

Das Ziel dieser Arbeit ist es, dieses vorhandene System um zwei weitere Module zu erweitern und damit eine Positionsbestimmung mit dem PRHS-Framework zu ermöglichen. Diese Module sind zum einen ein *Pmod GPS-Receiver* und zum anderen der On-Board Beschleunigungssensor des Nexys 4 DDR. Für die Umsetzung wird die Hardware mithilfe der Hardwarebeschreibungssprache Very High Speed Integrated Circuit Hardware Description Language (VHDL) beschrieben und der Treiber für Linux in der

Programmiersprache *C*. Für die Nutzung und Steuerung der Sensoren werden, die im PRHS-Framework bereits vorhandenen, Schnittstellen genutzt und an die Anforderungen angepasst.

## 1.2 Struktur der Arbeit

Zur Umsetzung des Zieles, wird die Arbeit wie folgt gegliedert.
Zu Beginn der Arbeit werden im Kapitel 2 die theoretischen Grundlagen vermittelt. Als Ausgangspunkt wird im Kapitel 2.1 auf den Aufbau und die Funktionsweise von FPGAs eingegangen, gefolgt von einer näheren Erklärung des PRHS-Framework in Kapitel 2.2. Die Kommunikation zwischen Peripheriegeräten und dem System findet über Schnittstellen statt (Kapitel 2.3). Dies beinhaltet die im Rahmen dieser Arbeit genutzten Schnittstellen Universal Asynchronous Receiver Transmitter (UART), für den GPS-Receiver, in Kapitel 2.3.1 und die Serial Peripheral Interface (SPI)-Schnittstelle in Kapitel 2.3.2 für den Beschleunigungssensor ADXL362. Die technischen Daten und wichtige Informationen des genutzten GPS-Sensors werden in Kapitel 2.5 erläutert, wobei im Unterkapitel 2.6 explizit auf den sogenannten Global Positioning System-Daemon (GPSd) eingegangen wird. Im Kapitel 2.7 werden die Funktionsweisen und Eigenschaften des Beschleunigungssensor ADXL362 dargestellt. Das Kapitel 2.8 erklärt grob die Funktionsweise eines Betriebssystems, wie hier anhand des Beispiels Linux. Hierbei gibt es zwei Unterkapitel. Das Kapitel 2.8.1, welches die wesentlichen Eigenschaften und Funktionsweisen eines Treibers erläutert und das Kapitel 2.8.2. Dieses beschreibt ein Tool, welches zu Erzeugung eines Betriebssystems genutzt werden kann.
Die praktische Implementierung in Kapitel 3 gliedert sich in zwei Teile. Ein Teil davon besteht aus dem GPS-Sensor, dessen Hardwareanbindung in Kapitel 3.1.1 näher erklärt wird und einer Beschreibung des Treibers in Kapitel 3.1.2. Der zweite Teil von Kapitel 3 beschäftigt sich mit der Anbindung des Beschleunigungssensors in Kapitel 3.2, welche wiederrum aus der Beschreibung der Hardwareanbindung in Kapitel 3.2.1 und der des Treibers in Kapitel 3.3 besteht.
Um nun die erarbeiteten Ergebnisse zu testen, wird in Kapitel 4 beschrieben, wie erst der GPS-Sensor in Kapitel 4.1 und der Beschleunigungssensor in Kapitel 4.2 getestet wird. Abschließend werden alle Module in einer Applikation zusammengefasst, welche in Kapitel 4.3 beschrieben wird.
Im Kapitel 5 wird die Arbeit noch einmal zusammengefasst und ein Ausblick gegeben.

# 2 Grundlagen

## 2.1 FPGA

Mit der Markteinführung der wiederprogrammierbaren Logikbausteine im Jahr 1984 begann ihre rasante Entwicklung. In den folgenden 30 Jahren konnten die FPGA unter anderem ihre Kapazität um den Faktor 10.000 steigern, die Geschwindigkeit um das 100-fache verbessern und gleichzeitig den Kosten- und Energieverbrauch um das 100-fache senken. [26]
Diese Vorteile führten zum Aufstieg des FPGA, indem man fortan wiederprogrammierbare Siliziumchips nutzte, welche nicht durch die Anzahl der verfügbaren Prozesskerne eingeschränkt waren und im Gegensatz zu Prozessoren echte Parallelität boten. [14]
Die wichtigsten Komponenten eines FPGAs sind die konfigurierbaren Logikblöcke, auch Configurable Logic Block (CLB) genannt (Aufbau siehe Abbildung 2.1). Diese bestehen wiederrum aus zwei elementaren Teilen: Denm FlipFlop (FF) und der Look-Up-Table (LUT).

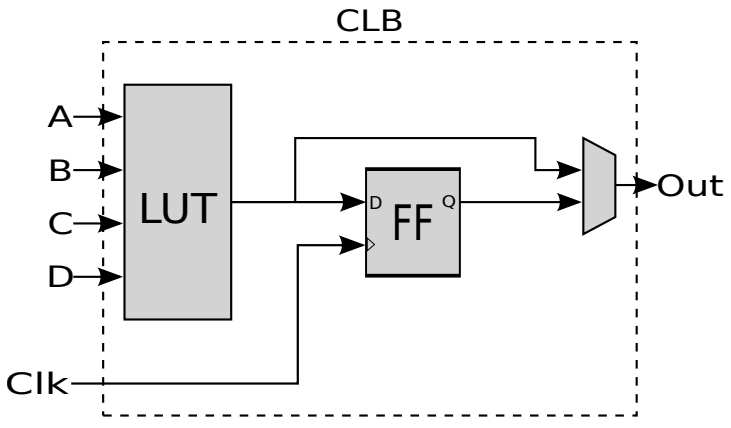


Abbildung 2.1: Aufbau eines CLBs nach [22]

Die oben genannten FlipFlops dienen zur Synchronisierung von Logikzuständen, sowie zur Speicherung der Zustände, zwischen zwei Taktraten. Am Eingang des FlipFlop liegt entweder der Wert *'1'*(True) oder der Wert *'0'*(False) an, welcher bis zur nächsten steigenden Taktflanke gehalten wird (Abbildung 2.2). [14]

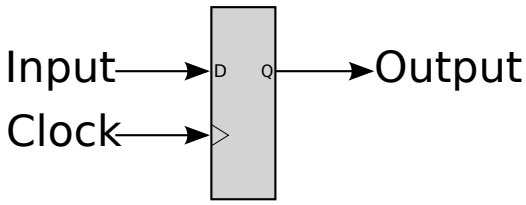


Abbildung 2.2: FlipFlop-Symbol nach [14]

In der LUT wird jeder möglichen Kombination von Eingangswerten ein festgelegter Ausgangswert zugewiesen (siehe Abbildung 2.3). Durch diese Eigenschaft ist es dem CLB möglich, jede, vom Eingang abhängige, beliebige boolsche Funktion anzunehmen. Standardmäßig werden heutzutage 6-Input-LUTs genutzt, welche, im Gegensatz zu den vorherigen 4-Input-LUTs, den Aufwand zwischen zwei LUTs verringern.[18]

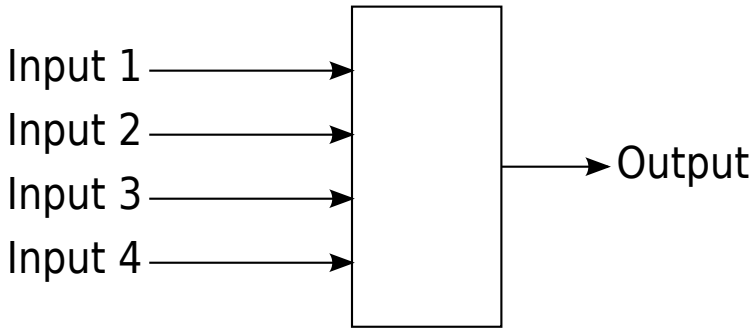


Abbildung 2.3: 4-Input-LUT nach [14]

Das FPGA verfügt als weitere Bausteine über sogenannte I/O-Blocks (IOBs). Diese werden benötigt, um Signale auf (Input), beziehungweise von dem FPGA zu senden (Output). Diese Blöcke sind ebenfalls konfigurierbar, sodass sie an die benötigten Standards angepasst werden können (CMOS/TTL).

Dadurch, dass die Architektur mittlerweile weitestgehend standardisiert wurde, wird die Hardware heutzutage mithilfe einer softwareähnlichen Hardwarebeschreibungssprache

beschrieben. Somit besitzt das System alle Vorteile der wiederverwendbaren und austauschbaren Bausteine bzw. Hardwarekomponenten. Jedoch sind FPGAs in der heutigen Herstellungsart hochkomplexe Strukturen, die einen Mittelweg zwischen hoher Taktfrequenz, um genügend Ressourcen zu besitzen und preisgünstiger Herstellung meistern finden müssen. [18]

## 2.2 PRHS-Framework

Das Kapitel 2.2 bezieht sich fast ausschließlich auf eine Arbeit, welche an der Helmut-Schmidt-Universität, Universität der Bundeswehr in Hamburg, genauer an der Professur für Technische Informatik in Form einer Dissertation entstanden ist (siehe [7]).
Das sogenannte PRHS ist zum Teil portierbar. Dabei ist es möglich, während der Laufzeit den FPGA zu konfigurieren, da dieser vom System simuliert wird.[7]

### PRHS-Bus

Die Verbindung zwischen der Central Processing Unit (CPU), in diesem Fall der Master und dem Speicher beziehungsweise den Peripheriegeräten, auch Slave genannt, wird durch den PRHS-Bus dargestellt. Das Hauptsystem, auf welches diese Arbeit aufbaut, besitzt spezifische Eigenschaften, welche im Nachfolgenden näher beschrieben werden.

- Unter anderem werden dem System Datenleitungen zum Lesen(ReadData) und Schreiben(WriteData) zur Verfügung gestellt. Dies bedeutet, dass die angeschlossenen Peripheriegeräte jederzeit kommunizieren können.
- Eine weitere Operation ist die *swap*-Operation, bei der zu erst Daten von einer bestimmten Adresse gelesen werden, bevor Daten auf diese geschrieben werden. Diese Operation kann *nicht* unterbrochen werden.
- Es handelt sich bei dem System, mit dem gearbeitet wird, um ein 32-Bit System. Daraus folgt, dass Adress- und Datenleitungen genutzt werden, die eine Bitbreite von 32 vorweisen.
- Über diese Datenleitungen sollen verschieden breite Datenworte übertragbar sein. Dazu zählen *word* mit 32-Bit, *half-word* mit 16-Bit und *byte* mit 8-Bit.

- Essenziell wichtig war zudem, dass alle Geräte und Teilnehmer innerhalb des Bussystems über eine gleiche Taktrate arbeiten.

Der verwendete PRHS-Bus lässt sich folgendermaßen darstellen [7]:

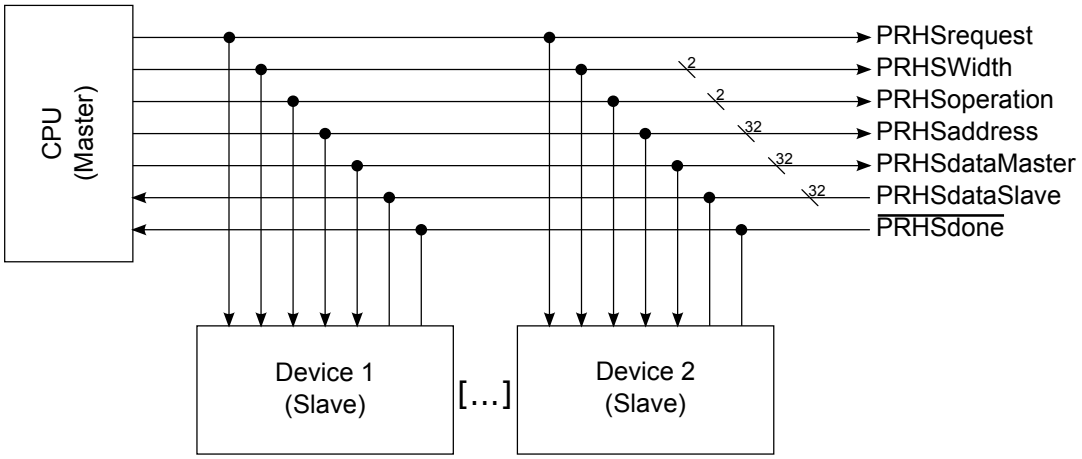


Abbildung 2.4: Aufbau des verwendeten PRHS-Busses aus [10]

### Schreib- und Leseoperationen

Damit die gewünschte Schreiboperation ausgeführt wird, muss der Master einige Parameter des PRHS-Busses setzen. Das *PRHSoperation*-Signal muss auf '*OPWrite*' gesetzt werden, um mitzuteilen, dass es sich um eine Schreiboperation handelt. Des Weiteren muss vor dem Ausführen die *PRHSaddress*-Leitung mit der gewünschten Adresse beschrieben werden. Der *PRHSdataMaster* wird mit dem zu schreibenden Datenwort belegt. Die Breite des Datenwortes wird über das *PRHSWidth*-Signal gesetzt. Der Master signalisiert über das Setzen des *PRHSrequest* auf '*1*' , dass die Daten auf dem Bus eine Anfrage darstellen.
Nun überprüfen die angeschlossenen Slaves bei der nächsten steigenden Taktflanke, ob die in *PRHSaddress* hinterlegte Adresse in ihrem Bereich liegt. Das angesprochene Gerät quittiert den Request, in dem es das *PRHSdone*-Signal von '*1*' auf '*0*' setzt. Nun ist der Master wieder in der Lage neue Daten auf den gewünschten Bus zu legen und den *PRHSrequest* zu resetten.
Jegliche Daten auf dem *PRHSdataSlave*-Signal werden vom Master ignoriert. Dies erlaubt, unnötigen Schaltaufwand innerhalb des Slaves zu verhindern.[7]

Grafisch dargestellt sieht eine Schreiboperation wie folgt aus :

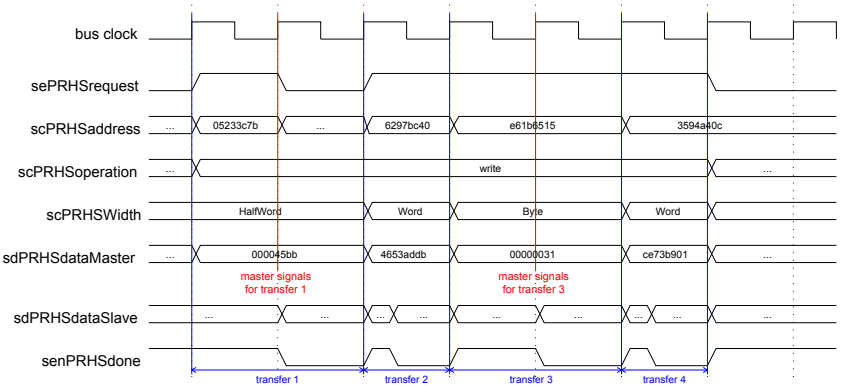


Abbildung 2.5: Darstellung der Signale einer Schreiboperation nach [7]

Für die Leseoperation gibt es zwei Varianten. So kann der Slave im ersten Fall direkt antworten oder im zweiten Fall den Bus solange blockieren, bis er die Daten abrufen möchte. Die zweite Variante hat dadurch den klaren Nachteil, da in der Zeit der Bus für weitere Slaves blockiert ist.
Um nun die Daten vom Slave über die *PRHSdataSlave*-Leitung zu empfangen, muss der Master die Signale *PRHSaddress*, *PRHSWidth* und *PRHSrequest* setzen. Entscheidend ist nun, dass der Master das *PRHSoperation*-Signal auf '*OPread*' setzt.
Fallunterscheidung:

1. Der Slave legt seine Antwort unverzüglich auf die *PRHSdataSlave*-Leitung und quittiert per *PRHSdone*-Signal, dass die vom Master gestellte Anfrage bearbeitet wurde.(siehe Abbildung 2.6)

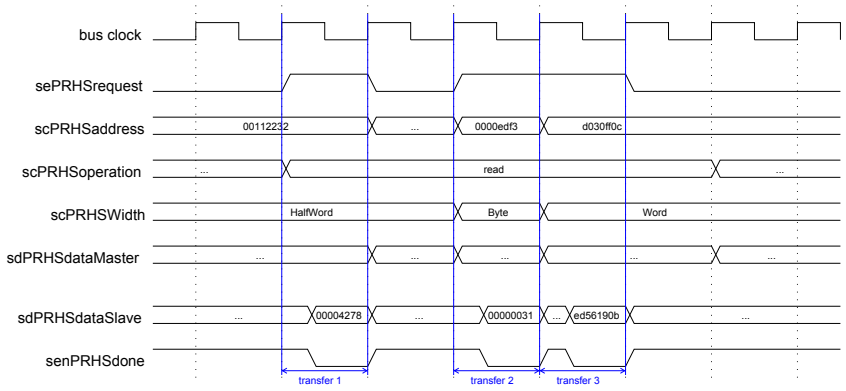


Abbildung 2.6: Darstellung einer Leseoperation mit sofortiger Antwort nach [7]

2. Erwartet der Slave nun weitere Daten, oder muss Berechnungen durchführen, hält er das *PRHSdone*-Signal solange zurück und blockiert damit den Bus. Sollte nun ein Gerät nicht antworten, kann es zum Absturz des Gesamtsystems kommen.(siehe Abbildung 2.7)

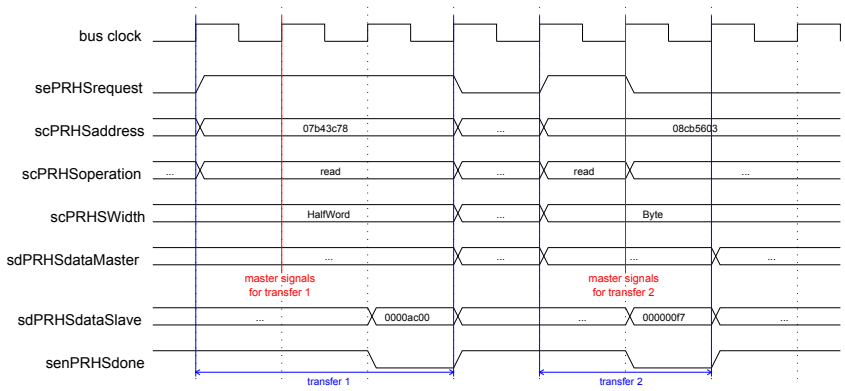


Abbildung 2.7: Darstellung einer Leseoperation mit verzögerter Antwort nach [7]

Dabei ist es wichtig zu erwähnen, dass diese Operationen für Geräte immer *non-immediate* geschehen. Das heißt, dass nach dem Senden des *PRHSdone*-Signals eine Zeitverzögerung auftritt, um eine Befehlüberschneidung zu verhindern. Der Nachteil besteht darin, dass somit die Operationen mehr Zeit in Anspruch nehmen.

## 2.3 Schnittstellen

Damit das Nexys 4 DD-Board mit externen Geräte kommunizieren kann, müssen zum Beispiel serielle Schnittstellen wie RS-232 oder Bus-Systeme wie SPI genutzt werden, welche sowohl vom Board, als auch vom Gerät unterstützt werden. [21]
In den nachfolgenden Unterkapiteln werden beide Arten der Kommunikation beschreiben.

### 2.3.1 UART

Um mit dem verwendeten GPS-Modul zu kommunizieren, wurde UART als serielle Schnittstelle genutzt. Bei dieser Art der Kommunikation werden serielle Daten zwischen dem Board (*Master*) und dem GPS-Empfänger (*Slave*) ausgetauscht. Dabei spielen die TxD und die RxD-Datenleitungen eine wichtige Rolle, da der Datenaustausch über diese Leitungen realisiert wird.
Im Gegensatz zu der SPI-Schnittstelle ( 2.3.2), wird kein Clock-Signal übertragen um Daten zu validieren. Des Weiteren wird die Verbindung mit einer definierten Geschwindigkeit realisiert, der sogenannten *Baudrate*. [3]
Die Baudrate bezeichnet dabei die Anzahl der Bits, welche pro Sekunde übertragen werden. Die bei dieser Arbeit genutzte Baudrate von 9600 wurde durch das genutzte GPS-Modul ebenso vorgegeben, wie die Übertragung von 8 Daten-Bits, keine Parität und einem einzigen Stop-Bit.[6]
UART konvertiert die Bytes in serielle Bits, überträgt diese über eine einzelne Leitung und liest die zugehörigen Start- und Stop-Bits aus.
Das sogenannte *Character*(Zeichen) besteht aus einer konfigurierbaren Anzahl an Datenbits (in den meisten Fällen 7 oder 8), aus einem *low-level* Start-Bit , einem optionalen Parity-Bit und einem oder mehreren logischen *high-level* Stop-Bits.
Das Start-Bit teilt dem Receiver mit, dass ein neues *Character* empfangen wird. Die nächsten Bits, je nachdem wie viele Daten-Bits vorher konfiguriert wurden , stellen dann den Inhalt des *Character* dar. Darauf folgt das optionale Parity-Bit, welches anzeigt, ob die Anzahl der mit '*1*' belegten Daten-Bits gerade oder ungerade ist. Am Ende der Folge stehen dann entweder ein oder zwei *high-level* Stop-Bits, welche dem Receiver eindeutig signalisieren, dass die Übertragung vollständig ist. Dadurch, dass das Start-Bit *high-level* (1) und das Stop-Bit *low-level* (0) ist, ist immer eine klare Abgrenzung zwischen dem derzeitigen und dem folgenden *Character* möglich.

Die Datenübertragung lässt sich nach 2.8 darstellen.

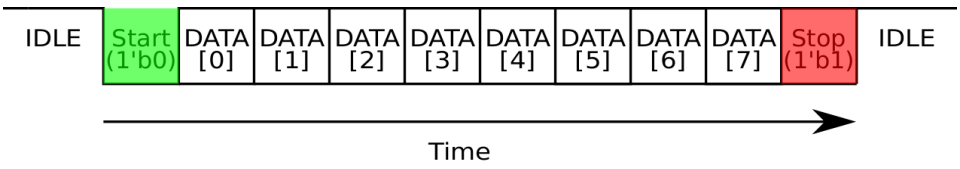


Abbildung 2.8: Datenübertragung per UART

### 2.3.2 SPI

SPI ist eines der am meisten genutzten seriellen Protokolle für *inter-chip* oder *intra-chip* Datenübertragungen. In dieser Arbeit wurde SPI dafür genutzt, um mit dem On-Board Beschleunigungssensor zu kommunizieren.
Bei der Übertragung via SPI findet der Datenaustausch zwischen dem Master (Kernel) und dem Slave (Beschleunigungssensor) statt. Um einen Austausch zu gewährleisten benötigt der Master ein *Clock*-Signal, über welches gesteuert wird, wann Daten geschrieben beziehungsweise gelesen werden können. So können erst Daten gesendet werden, nachdem die empfangenen Daten gelesen wurden. Daraus ergibt sich, dass ein Gerät im SPI-Protokoll immer ein Receiver und Transmitter ist und niemals nur eines von beidem. Für die Übertragung nutzt SPI folgende Signale:

- SCK (Serial Clock) Dieses Signal synchronisiert die Übertragung zwischen Master und Slave und wird vom Master gesetzt.
- MOSI (Master Out – Slave in) Eine einfache Daten-Bit-Verbindung, welche der Master aus seinem Register heraus befüllt.
- MISO (Master In Slave Out) Über diese einfache Daten-Bit-Verbindung kommuniziert der Slave mit dem Master, indem er die Bits aus dem Slave-Register sendet.
- SS (Slave Select) Wenn dies Verbindung eine *'1'* überträgt und somit ein *high-level*-Signal wird, wird das entsprechende Gerät ausgewählt. Diese Auswahl wird durch den Master getätigt.

Die Datenübertragung basiert dabei auf einem 8-Bit Datenregister, welches sowohl der Master, als auch der Slave besitzt. Durch das vom Master generierte *Clock*-Signal findet nun eine Übertragung statt und der Master schiebt sein Datenbyte auf das *MOSI*-Signal. Analog dazu kann der Slave seine Daten wiederrum auf das *MISO-Signal* legen. Wenn nun acht *Clock*-Impulse generiert wurden, werden die Daten in das Register des Anderen geschoben. [19]

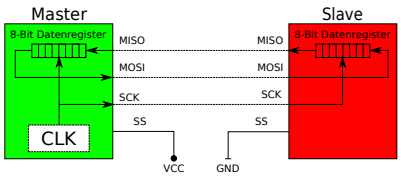


Abbildung 2.9: Datenübertragung per SPI nach [27]

### SPI-Modus

Über die SPI-Schnittstelle kann der Slave in verschiedenen Modi betrieben werden. Diese Modi setzen sich dabei aus den Werten der Parameter Clock Plarity (CPOL) und Clock Phase (CPHA) nach Tabelle 2.1 zusammen.

| Mode | CPOL | CPHA |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 0 |
| 3 | 1 | 1 |

Tabelle 2.1: SPI-Modi aus [2]

In dieser Arbeit wird der SPI-Modus '*0*' genutzt. Das bedeutet, dass sowohl CPOL, als auch CPHA den Wert '*0*' annehmen. Dabei ist der Clock Idle 'Low'(CPOL='0') und die Daten werden bei der ersten Flanke übernommen, nach dem SS auf 'Low' gezogen wurde(CPHA='0'). Daraus ergibt sich, dass die ankommenden Daten bei der ersten Flanke übernommen werden, welche nur eine steigende Flanke sein kann. [21]

## 2.4 GPS

Was in den 60er Jahren als militärisches Projekt begann, lässt sich heute in fast jedem Auto, Schiff, Flugzeug und Smartphone wiederfinden. Das GPS, auch NAVSTAR genannt, wurde als erstes Navigationssystem auf Basis von Satellitenortung entwickelt. Nach einigen finanziellen und technischen Problemen läuft das System seit 1995 ohne größere Probleme.[1]
GPS setzt sich aus 24 Satelliten zusammen, welche auf elliptischen Bahnen um die Erde kreisen und sich dabei in einer Höhe von ca. 20200 km befinden. Die Satelliten sind dabei *nicht* geostationär, sondern bewegen sich zu je vier Satelliten auf sechs unterschiedlichen Bahnen. Gegeneinander sind sie zu 60 Grad und gegen die Äquatorebene 55 Grad geneigt. So wird sichergestellt, dass zu jeder Zeit und an jedem Ort der Erde mindestens vier Satelliten nutzbar sind. [1]
Die Entfernungsbestimmung durch Laufzeit ist ein durchaus kompliziertes Verfahren. Die Position des Empfängers wird durch drei Satelliten bestimmt, indem diese Datenpakete senden, welche die aktuelle Position und die Sendezeit beinhalten. Da nun diese Signale mit Lichtgeschwindigkeit übertragen werden, würde es bei einem Fehler von ei-

ner tausendstel Sekunde zu einer Abweichung von 300 km kommen. Um die notwendige Genauigkeit zu gewährleisten, werden mehrere Atomuhren an Bord der Satelliten eingesetzt, die rund um die Uhr von der Erde aus kontrolliert werden. Durch den Einfluss der speziellen Relativitätstheorie, dass heißt aufgrund der reduzierten Schwerkraft, gehen die Uhren an Bord langsamer. Dem wirkt man entgegen, indem diese Atomuhren mit einer niedrigeren Grundfrequenz eingestellt werden. [1]

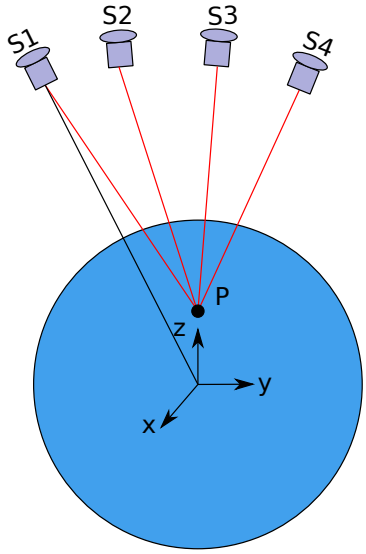


Abbildung 2.10: Positionsbestimmung per GPS nach [1]

Die Positionsbestimmung wird anhand der kartesischen Koordinaten berechnet und angeben. Um das Gleichungssystem mit den Unbekannten $X_p$ , $Y_p$ , $Z_p$ und dem Distanzfehler D zu lösen, werden mindestens vier Satelliten benötigt. Befindet sich nun ein Satellit bezüglich des Koordinatensystems auf der Position ($X_0$ , $Y_0$ , $Z_0$), wie in der Abbildung 2.10 dargestellt, wird dieser Punkt als Referenzpunkt für die Berechnung der anderen Satelliten genutzt. So gilt für $R_0$:

$$R_0 = \sqrt{(X_p - X_0)^2 + (Y_p - Y_0)^2 + (Z_p - Z_0)^2} - D$$

## 2.5 Pmod-GPS-Receiver

Der in dieser Arbeit genutzte GPS-Sensor ist ein *PmodGPS*-Sensor der Firma Digilent Inc..
Dieser kann verschiedene Systeme um die Funktion der Positionsbestimmung via Satelliten erweitern und besitzt des Weiteren folgende Eigenschaften:[12]

- Zwei- und dreidimensionale Positionsbestimmung
- Hohe Empfindlichkeit (-165 dBm)
- Eine Genauigkeit von ±3 Meter
- Aktualisierungsrate von bis zu 10 Hz
- Kleine Bauart (0,8 cm x 5,0 cm x 2,0 cm)
- National Marine Electronics Association (NMEA)-Standard für die Kommunikation
- Datenübertragung zum System via UART
- Automatisches Wechseln zwischen interner und externer Antenne(falls vorhanden)

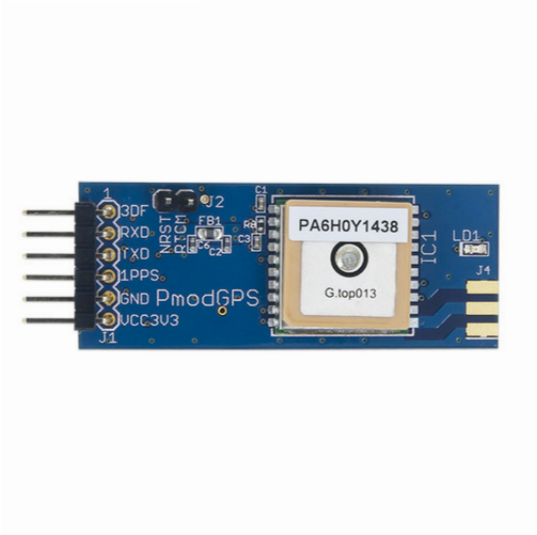


Abbildung 2.11: Der Pmod GPS-Receiver nach [12]

### 2.5.1 System Block Diagram

Die Abbildung 2.12 zeigt das System Block Diagram des GPS-Sensors.

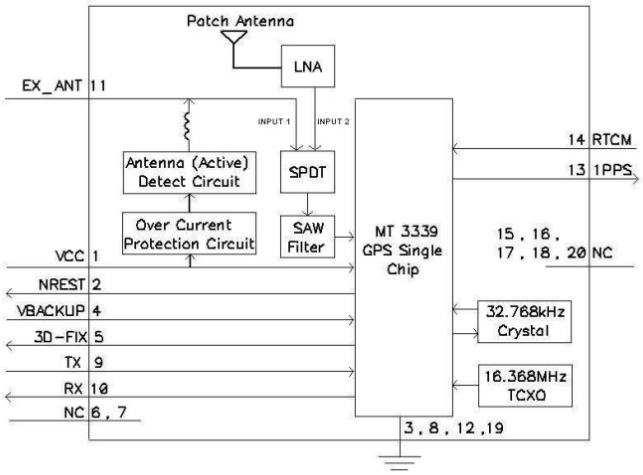


Abbildung 2.12: System Block Diagram aus [12]

Für die Implementierung sind jedoch nur die I/O-Signale wichtig(siehe Abbildung 2.2). In diesem Kapitel wird näher auf die sechs Pins des Sensors eingegangen, sowie auf den Pin, welcher zum externen Zurücksetzen (*Reset*)genutzt werden kann. Das Erweitern des Sensors mithilfe einer externen Antenne zur Verbesserung des Empfangs wird im nächsten Abschnitt näher erläutert.

| **Connector J1** | | |
|---|---|---|
| **Pin** | **Signal** | **Description** |
| 1 | 3DF | 3D-Fix Indicator |
| 2 | RX | Receive |
| 3 | TX | Transmit |
| 4 | 1PPS | 1 Pulse Per Second |
| 5 | GND | Power Supply Ground |
| 6 | VCC | Power Supply (3.3V) |
| **Connector J2** | | |
| **Pin** | **Signal** | **Description** |
| 1 | ~RST | Reset (active low) |
| 2 | RTCM | DGPS data pin |

Tabelle 2.2: Pinübersicht des GPS-Sensors nach [5]

Wie in der Abbildung 2.12 und in der Tabelle 2.2 gezeigt, besitzen die einzelnen Pins verschiedene Aufgaben:[12]

1. **3DF**: Der sogenannte *3D-Fix Indicator* gibt den Status der Positionsbestimmung des Nutzers an. Bei konstanter Positionierung ist der Wert '*0*', sprich ein *low-level*-Signal. Ist es nicht möglich eine Position zu bestimmen, wird das Signal jeweils abwechselnd für eine Sekunde auf '*1*' beziehungsweise '*0*' gesetzt.
2. **RX/RXD**: Ermöglicht die Übertragung von seriellen Daten für Softwarekommandos oder ein mögliches Firmware Update über UART.
3. **TX/TXD**: Dieses Signal wird für die Übertragung von seriellen GPS-Daten via UART genutzt, welche standardmäßig das NMEA-Format haben. Diese Informationen können anschließend in Applikationen genutzt werden.
4. **1PPS**: Dieser Pin gibt dem Eingang des angeschlossenen System ein *Puls-pro-Sekunde*-Signal, welches zur Synchronisation der GPS-Zeit dient.
5. **GND**: Der Masse (engl. *chassis ground*) ist das Potential null zugeordnet und stellt somit das Bezugspotential für alle Signal- und Betriebsspannungen dar.
6. **VCC**: Dieser Pin dient als Gleichspannungsversorgung des Moduls, wobei der Wert zwischen 3,0 und 4,3 Volt liegen muss. Die typische Spannung beträgt 3,3 Volt.

Das ~*RST*-Signal, welches auf dem zweiten Pin des *J2-Connector* liegt, ist ein sogenanntes *active-low*-Signal und damit vom Sinn her invertiert. Um das Modul zu zurückzusetzen, muss demnach eine '*0*' auf das Signal gegeben werden.

### Externe Antenne

Das Erweitern des Moduls um eine externe Antenne ist leicht umzusetzen. Hierfür wird ein Sub-Miniature-A (SMA)-Adapter benötigt, welcher dann am *J4-Connector* angelötet wird. In dieser Arbeit wird dafür der *CONSMA003.062* der Firma *Linx Technologies Inc.* verwendet(siehe Abbildung 2.13).

An diesem SMA-Adapter wird nun die externe Antenne angeschraubt. Hier wird die *ANT-GPS-SH-SMA-ND*-Antenne der Firma *Linx Technologies Inc.* genutzt(siehe Abbildung 2.14), welche einen Frequenzbereich von 1570,24MHz bis 1580,42MHz abdeckt.

Abbildung 2.13: Abbildung des verwendeten SMA-Adapters aus [9]

Ddamit wird eine der GPS-Trägerfrequenzen abgedeckt, welche bei 1575,42MHz liegt.[15]

Abbildung 2.14: Abbildung der verwendeten Antenne aus [8]

### Output Sentences

Die Daten, welche von dem GPS-Sensor auf das Datensignal gelegt werden, entsprechen dem NMEA-Standard. Dabei enthält ein gesamtes Datenpaket fünf verschiedene *Options*. Diese *Options* beinhalten verschiedene Informationen und sind im Anhang näher erklärt.

Als Beispiel für eine der *Options*, wird die Global Positioning System Fix Data (GGA) in Listing 2.1 und Tabelle 2.3 erläutert.[12]

```
1 \$GPGGA,064951.000,2307.1256,N,12016.4438,E,1,8,0.95,39.9,M,17-8,M,,*65
```

Listing 2.1: *NMEA-Message*-Beispiel für GGA aus [12]

| Example | Description |
|---|---|
| $GPGGA | Message ID |
| 064951.000 | UTC Time (hhmmss.sss) |
| 2307.1256 | Latitude (ddmm.mmmm) |
| N | N/S Indicator |
| 12016.4438 | Longitude (ddmm.mmmm) |
| E | E/W Indicator |
| 1 | Position Fix Indicator |
| 8 | Satellites used |
| 0.95 | HDOP |
| 39.9 | MSL Altitude |
| M | Units |
| 17.8 | Geoidal Separation |
| M | Units |
| | Age of Diff.Corr. |
| *65 | Checksum |
| <CR><LF> | End of message indicator |

Tabelle 2.3: GGA-Daten Format aus [6]

Beim späteren Verwenden der Daten in einer Applikation(siehe Kapitel 4.3), hilft der GPSd, welcher die Daten aufbereitet und in *structs* gespeichert zur Verfügung stellt. Dies ist der entscheidene Vorteil gegenüber eines direkten Datenabgriffs, ebenso wie die Möglichkeit, dass der GPSd diese Daten an mehrere Applikationen verteilen kann.

## 2.6 GPS-Daemon

Das Kapitel 2.6 beschäftigt sich mit der näheren Erläuterung und Implementierung des sogenannten *GPSd*. Es handelt sich hierbei um einen *Disk and Execution Monitor (Daemon)* , welcher unter dem Betriebssystem Linux/Unix(siehe Kapitel 2.8) ein Programm darstellt, das im Hintergrund abläuft und bestimmte Dienste, in diesem Fall

GPS-Daten, zur Verfügung stellt. Es gibt keine direkte Interaktion mit dem Benutzer, sondern lediglich indirekte Wege, welche beispielsweise über Signale oder Netzwerksockets realisiert werden. [31]

Der GPSd im Speziellen stellt dem System Informationen von GPS- und Differential Global Positioning System (DGPS)-Geräten, sowie von Automatic Identification System (AIS)-Empfängern zur Verfügung. Dabei ist er in der Lage, neben dem NMEA-Standard-Protokoll weitere NMEA-Dialekte, wie zum Beispiel *MKT-3301*, *iTrax*, *Motorola OnCore*, aber auch Binärprotokolle (*Garmin*,*Navcom*,*SiRF* etc.) abzufragen und zu interpretieren. [24]
Der *Daemon* besitzt die Fähigkeit, Unterschiede zwischen den unterstützen GPS-Typen zu verstecken und kann ebenfalls Kommandos an den GPS-Sensor schicken, um gegebenenfalls die Latenz zu verringern. [24]

Der spätere Aufruf des Daemon erfolgt über die Eingabe in der Kommandozeile, bei der neben dem Aufruf selbst noch diverse Optionen mit angehängt werden können. [24]
Die, für diese Arbeit wichtigen Optionen, werden nun näher erklärt.

```
1 gpsd [-b] [-n] [-N] [-D n] [-F sockfile] [-P pidfile] [-S port] [-h]
      device...
```

Listing 2.2: Aufruf des GPSd in der Kommandozeile

- **-b** : Stellt eine 'Nur-Lesen'-Verbindung zum GPS-Gerät her, da einige GPS-Geräte sich nicht von dem *Daemon* rekonfigurieren, beziehungsweise aktualisieren lassen.
- **-n** : Auf etwaige Anwendungen wird nicht gewartet, sondern die Positionsdaten ohne Nutzung oder Bestätigung versendet, sodass diese von Beginn an auf dem gewählten Port liegen.
- **-N** : Wie bereits am Anfang des Kapitels angemerkt, lässt sich ein *Daemon* durch Benutzereingaben im Vordergrund ausführen, um so, wenn vorhanden, Logausgaben zu bekommen.
- **-F** : Es wird eine Socket-Datei erstellt, bei der die bidirektionale Softwareschnittstelle angegeben wird. Hierbei ist eine Angabe eines Speicherortes nötig. In diese Datei lassen sich dann Befehle schreiben, welche die Geräteliste des *Daemon* editieren.

- **-P** : Über diese Option erreicht der Nutzer, dass eine Process-Identification-Data (PID)-Datei erstellt wird. Hierbei wird die Angabe eines Speicherortes, sowie des Dateinamens vorausgesetzt.
- **-D** : Diese Option setzt das sogenannte *Debug*-Level. Es ist standardmäßig auf *'0'* gesetzt und kann so zwischen *'0'* und *'9'* liegen. Ab *Debug*-Level *'2'* schreibt der *Daemon* sämtliche Aktionen und eingehende Befehle in den Standard-Error, wenn er im Vordergrund läuft (siehe Option *'-N'*) oder in den *syslog*, wenn er im Hintergrund läuft.
- **-S** : Der *Daemon* bestizt die Fähigkeit, die empfangenen Daten per Transmission Control Protocol/Internet Protocol (TCP/IP) im Netzwerk zur Verfügung zu stellen. Mit dieser Option lässt sich der Port explizit angeben. Wird kein Port definiert, nutzt der *Daemon* standardmäßig den Port 2947.
- **-h** : Wird diese Option im Aufruf genutzt, werden sämtliche Befehle und Erklärungen als Hilfe angezeigt
- **-V** : Anzeige der genutzten Version. Danach wird der *Daemon* beendet.

## 2.7 Beschleunigungssensor

Bei dem ADXL362-Beschleunigungssensor handelt es sich um einen sogenannten *On-Board,3-axis Microelectromechanical System (MEMS) Accelerometer* der Firma Analog Devices Incorporated. Er ist, wie in Abbildung 3.2 (Nummer 2) gezeigt, auf der Rückseite des Nexys 4 DDR-Boards verbaut. Er bietet bei der Bewegungserkennung eine hohe Genauigkeit und misst dabei Beschleunigung, Neigung, Erschütterung und Vibration und stellt diese Daten etwaigen Applikationen zur Verfügung. Darüber hinaus besteht die Möglichkeit die Chiptemperatur auszulesen.
Dabei findet er neben dem Einsatz auf dem Board weitere Anwendung in Bereichen wie *Medizinischen Messgeräten*, *Spielekonsolen* oder als *Alarm-* und *Bewegungsmelder*.[13]
Als typische Eigenschaften weist der Sensor folgende auf :[11]

- Niedriger Energieverbrauch
- Hohe Auflösung
- Aktivitäts-/Inaktivitätsüberwachung

- Versorgungsspannungsbereich: 1,8V-3,6V
- Digitale SPI- und $I^2C$-Schnittstelle
- Breiter Temperaturbereich: -40°C bis +125°C

Wie oben aufgeführt nutzt dieser Sensor die SPI-Schnittstelle, welche in Kapitel 2.3.2 näher erklärt wird.

Der ADXL362 verfügt über zwei Betriebsarten: *Measurement-Mode* für eine kontinuierliche, große Bandbreitenerfassung und *Wake-Up-Mode* für begrenzte Bandbreitenaktivität. Ebenfalls ist es möglich, die Messung auszusetzen, indem der Sensor in den *Stand-By*-Modus geschaltet wird. In dieser Arbeit findet der *Measurement-Mode* Anwendung, nachdem er einmal initialisiert wurde.(siehe Kapitel 3.2.1)

Wie bereits erwähnt, wird für die Datenübertragung eine SPI-Schnittstelle genutzt. Dabei agiert der Beschleunigungssensor als *Slave* und somit werden die Daten des Sensors ignoriert, solange der *Master* Daten an den Sensor schickt.
Des Weiteren wird der Sensor im SPI-Modus '*0*' betrieben.(siehe 2.3.2)
Die Verbindung per SPI wird in Abbildung 2.15 erläutert.

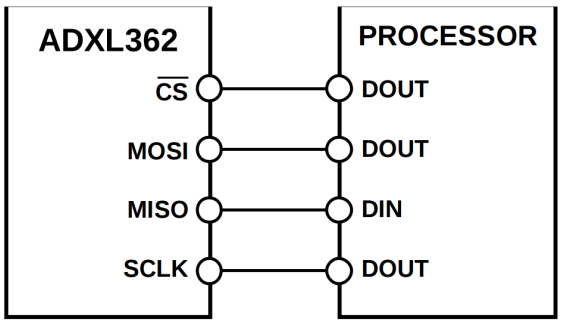


Abbildung 2.15: 4-Leiter-SPI-Anschlussschema aus [11]

### SPI-Befehle

Der SPI-Port nutzt eine *Multibyte*-Struktur, wobei das erste Byte das *Command* beinhaltet. Dafür bestizt der ADXL362 folgenden Befehlssatz:[11]

- **0x0A**: Register schreiben
- **0x0B**: Register lesen
- **0x0D**: First In-First Out (FIFO) lesen

Die *Command*-Struktur lässt sich wie in Abbildung 2.16 und 2.17 darstellen.

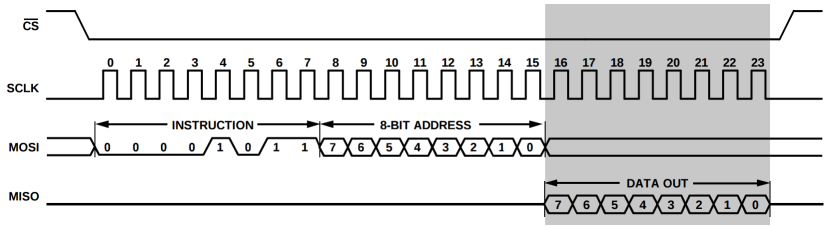


Abbildung 2.16: Register lesen (Befehl: 0x0B) aus [11]

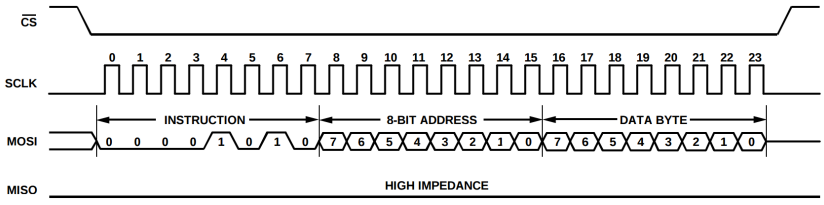


Abbildung 2.17: Register schreiben (Befehl: 0x0A) aus [11]

Ein FIFO-Read wurde in dieser Arbeit nicht praktisch umgesetzt, ist aber in [11] näher erklärt.

### Register Map

Um die gewünschten Daten über die Datensignale zu erhalten, ist das Auslesen der korrekten Register wichtig. Hierfür gibt es die sogenannte *Register-Map*, welche sich in der kompletten Form im Anhang befindet.

Die verwendeten Register und deren Aufgabe werden in Kapitel 3.2.1 näher erläutert. Ergänzend zu der *Register-Map* werden die Funktionen der einzelnen *ADXL362*-Register in den sogenannten *Register-Details* genauer betrachtet.

### Achsen-Daten-Register

Diese Register enthalten die acht Most Significant Bits (MSBs) der (X,Y,Z)-Achsen Beschleunigungsdaten.Angesprochen wird das Register über die jeweiligen Adressen(siehe 2.4).

| Achse | Adresse | Name | Reset |
|---|---|---|---|
| X-Achse | 0x08 | XDATA | 0x00 |
| Y-Achse | 0x09 | YDATA | 0x00 |
| Z-Achse | 0x0A | ZDATA | 0x00 |

Tabelle 2.4: Registertabelle nach [11]

So ergibt sich folgende Abbildung für die Register:

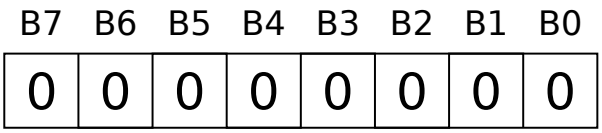


Abbildung 2.18: Achsen Datenregister nach [11]

## 2.8 Linux

Das verwendete System Linux wurde als freies Betriebssystem von Linus Benedict Torvald entwickelt. Es basiert auf dem Unix-Modell, jedoch schrieb Torvald den sogenannten Kernel neu. Seit dem gilt Linux als Open-Source-Projekt und Entwickler versuchen stetig das System zu verbessern.[30]
Die wohl wichtigste Eigenschaft die Linux bzw. Unix für den Einsatz beliebt machen ist die Portabilität, da es weitestgehend rechnerunabhängig läuft. Weitere Eigenschaften die das Betriebssystem zu einem der weitverbreitetsten Systeme überhaupt gemacht haben, sind:
[30]

- **Multi-Tasking** Das parallele Nutzen verschiedener Programme erlaubt jedem Benutzer gleichzeitige Aktionen durchzuführen, ohne auf den Abschluss der letzten Tätigkeit zu warten.

- **Time-Sharing** Das Priorisieren von einzelnen Prozessen erlaubt es, dass mehrere Prozesse gleichzeitig ablaufen, indem abwechselnd Platz im Hauptspeicher beziehungsweise in der CPU zugewiesen wird.

- **Sprachenvielfalt** Neben der Sprache C stellt Linux/Unix viele Programmiersprachen wie C++, Java, Python und viele mehr zur Verfügung. Dadurch, dass es sich um ein Open-Source-Projekt handelt, wird diese Bibliothek, je nach System, ständig erweitert.

- **Grafische Benutzeroberfläche** Linuxrechner, im speziellen die Anwendungen, nutzen in den meisten Fällen GNU Object Model Environment (GNOME) oder K Desktop Environment (KDE) als grafische Oberfläche. Durch die weitverbreitete Nutzung dieser Oberflächen wurden diese kontinuierlich weiterentwickelt und bieten so eine hohe Vielfalt an Anpassungsmöglichkeiten.

- **Netzwerk** Für die Kommunikation zwischen Server und Client hat sich das Betriebssystem bewährt, da es über ein umfangreiches Paket an Software verfügt. Neben den Standardprotokollen TCP/IP (IPv6 wird ebenfalls unterstützt), werden weitere Formate verwendet, wie zum Beispiel Unix to Unix Copy (UUCP), welches eine simple Form der Kommunikation zwischen Linux/Unix-Rechnern darstellt.

Grundsätzliche bestehen Betriebssysteme wie Linux/Unix aus drei Hauptkomponenten. Der **Kern** (engl. *Kernel*) bildet die grundlegenden Funktionen, wie zum Beispiel die Organisation und Verwaltung von Speicher, der Prozesse, sowie Ein- und Ausgänge und sämtliche Kommunikationsaufgaben.
Das **Dateisystem** (engl. *File System*) ermöglicht das Speichern von Dateien und errichtet einen Dateibaum und ist somit für die Datenorganisation zuständig.
Die dritte Komponente stellt der **Befehlsübersetzer** dar (engl. *shell*), welcher anhand einer Befehlssprache die Kommunikation ermöglicht und dem Benutzer erlaubt mit sämtlichen Peripheriegeräten zu interagieren, ohne dass die im Hintergrund laufenden Prozesse berücksichtigt werden müssen. [20]

Grafisch lässt sich der Aufbau wie folgt darstellen:

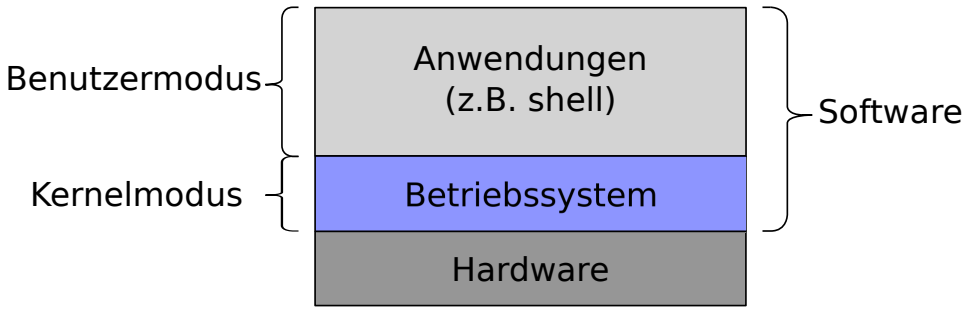


Abbildung 2.19: Aufbau der einzelnen Komponenten nach [25]

Um nun mit den verschiedenen Geräten über eine Vielzahl von Schnittstellen kommunizieren zu können, benötigt jedes Gerät einen eigenen Treiber. Dieser Treiber ist im Betriebssystem als *„Wissen“* hinterlegt und beinhaltet Information über das Gerät und dessen Zugriffsmöglichkeiten und wird im folgenden Unterkapitel näher erklärt. [23]
Der Anwender hat nun die Möglichkeit über die Treiber auf die Schnittstellen zuzugreifen, in dem diese vom Treiber gesteuert werden. Die in dieser Arbeit genutzten Schnittstellen (UART, SPI) werden in den Kapiteln 2.3.1 und 2.3.2 näher beschrieben.

### 2.8.1 Treiber

Der Treiber gilt als *„Lexikon“* der jeweiligen Geräte für das Betriebssystem. Diese Aufgabe ist enorm wichtig für ein funktionierendes System, sodass der Stellenwert eines Treibers sehr hoch ist. Einfach erklärt, besteht ein Treiber aus einer Reihe von Funktionen, die den Zugriff auf das Gerät steuern und so eine Kommunikation ermöglichen. So muss für jedes Gerät ein eigener Treiber implementiert werden, der die offene Schnittstelle im Betriebssystemkern füllt.
Dabei sind zwei Begrifflichkeiten wichtig, in die der Speicherbereich aufgeteilt wird:

- **Kernelspace** : Diese Bereich wird ausschließlich vom Kernel benutzt
- **Userspace** : Dieser Speicher wird von Applikationen genutzt und kann nicht, oder in Ausnahmfällen, nur eingeschränkt vom Kernel genutzt werden

Die Treiber werden grundsätzlich mit in den Kernel kompiliert oder sind als Modul dynamisch angelegt, um zum Kernel, während der Laufzeit, hinzugefügt zu werden. [28]

Bei der Implementierung von Gerätetreibern wird zwischen drei Gerätetypen unterschieden ([28]):

1. **Character-Devices** Auf diese Geräte kann wie auf einen Stream von Bytes zugegriffen werden, indem Portable Operating System Interface (POSIX)-Funktionen genutzt werden, wie beispielsweise *open()*,*write()*,*read()* oder *close()*. Für die Nutzung wird für jedes Gerät eine *Gerätedatei* (vgl. *device nodes*) angelegt.

2. **Block Devices** Im Gegensatz zu den *Character-Devices*, wird auf die *Block-Devices* immer per Random Access zugegriffen. Die Daten werden blockweise gelesen. Diese *Block-Devices* können ein Dateisystem aufnehmen und gemountet werden.

3. **Netzwerkschnittstellen** Netzwerkgeräte erhalten keine Gerätedatei, sondern sind in einer globalen Liste hinterlegt. Der Datenaustausch findet immer asynchron statt, sodass die Netzwerkkarte jederzeit bereit sein muss Daten zu empfangen.

#### Device Tree

Der *Device-Tree* ersetzt beim Advanced RISC Machines (ARM)-Prozessor das, was beim normalen PC das Basic Input/Output System (BIOS) ist. Er beinhaltet Informationen, welche dem Kernel beim Booten helfen. Da das System angeschlossene Geräte nicht automatisch erkennen und die dazu passenden Treiber laden kann, geschieht das in dieser *.dts*-Datei. Durch das Kompilieren wird daraus der sogenannte *Device Tree Blob*. Dieser wird zusammen mit dem Kernel vom Bootloader geladen und im Hauptspeicher abgelegt. Im Anschluss daran wird daraus eine Baumstruktur. In dieser sind die Geräte als Knoten angelegt und die dazugehörigen Treiber werden geladen. Die Verbindung zwischen Device-Tree und dem Treiber wird über eine eindeutige Identifikation per *Compatible* festgelegt.

### 2.8.2 Buildroot

*Buildroot* ist ein Tool, welches das Erstellen eines kompletten Linux-Systems per *Cross-Compiling* für eingebettete Systeme vereinfacht und automatisiert. Neben dem *cross-compiled* Toolchain können ebenfalls das *Root*-Dateisystem, ein *Linux-Kernel-Image* und ein *Bootloader* für das Zielsystem erstellt werden. In einem Konfigurationsprogramm lassen sich vor der Erstellung sämtliche Optionen und Erweiterungen ab- beziehungsweise

anwählen.
Das Tool wird größtenteils dafür genutzt, um die Systeme für andere Prozessoren, außer die klassischen x86-Prozessoren, zu erstellen, wie zum Beispiel ARM-Prozessoren.
Der in Kapitel 2.6 näher beschriebene GPSd lässt sich ebenfalls über das *Buildroot*-Tool dem Gesamtsystem hinzufügen.[17]

# 3 Praktische Implementierung

Das folgende Kapitel beschreibt die eigentliche Arbeit, in welcher das *NEXYS4 DDR Artix-7 FPGA*-Board der Firma Digilent Inc. (siehe Abbildung 3.1) verwendet wurde.
Die explizite Aufgabe dieser Arbeit ist das Anbinden des Pmod-GPS-Receivers über einen Pmod-Port (siehe Nummer 1 in Abbildung 3.1) und des 3-Achsen-Beschleunigungssensors (siehe Nummer 2 in Abbildung 3.2), welcher sich auf der Rückseite des Boards befindet.
Der FPGA ist in Abbildung 3.1 unter der Nummer 3 zu finden.

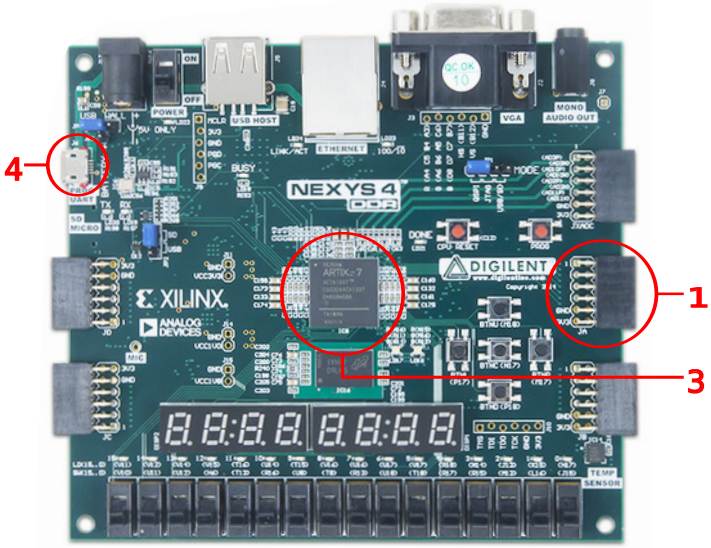


Abbildung 3.1: Vorderseite des *NEXYS4 DDR Artix-7 FPGA-Boards* nach [4]

Um nun die Konfiguration zu realisieren, wurde die Hardware mithilfe von VHDL beschrieben. Bei der Synthese wird nun aus dieser Hardwarebeschreibung eine Konfigurationsdatei.[18]

Über den „Shared UART/JTAG USB Port“ (siehe Nummer 4 in Abbildung 3.1) gelangt diese Konfiguration auf den FPGA.
Das in Kapitel 2.8 angesprochene Betriebssystem, wird mithilfe einer Speicherkarte, über den mit Nummer 5 in Abbildung 3.2 gekennzeichneten *Micro-SD Connector* gebootet,

welcher sich ebenfalls auf der Rückseite befindet.
Wie bereits am Anfang des Kapitels beschrieben, wird der GPS-Sensor nun über den *Pmod-Port* angeschlossen, welcher in Kapitel 3.1.1 näher beschrieben wird, hinsichtlich Nutzung und Pinbelegung.[4]

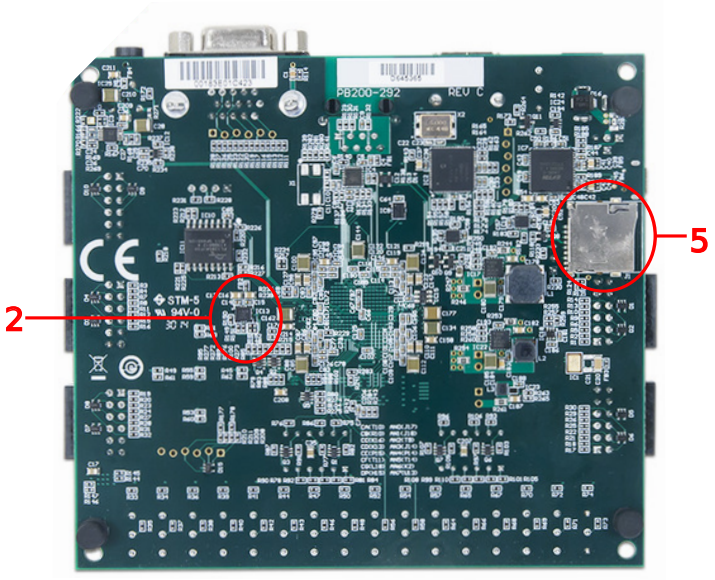


Abbildung 3.2: Rückseite des *NEXYS4 DDR Artix-7 FPGA*-Boards nach [4]

## 3.1 Anbindung des GPS-Sensors

Das Modul, welches eine Positionsbestimmung anhand von GPS-Daten ermöglichen soll, muss, bevor es verwendet werden kann zuerst angebunden werden. Dafür gibt es eine Beschreibung für die physische Anbindung der Hardware in Kapitel 3.1.1 und eine für die logische Anbindung durch einen Treiber in Kapitel 3.1.2.

### 3.1.1 Verbindung des GPS-Receivers mit dem FPGA

Auf der Seite der Hardware wird der Pmod-GPS-Receiver an einen der sogenannten *Pmod-Ports* (siehe Nummer 1 in Abbildung 3.1) gesteckt. Diese Ports bestehen aus insgesamt zwölf '*Female Connectors*', wovon jeweils zwei Pin 3,3 Volt VCC Signale (Pin 6 und 12 nach Abbildung 3.3) darstellen. Des Weiteren verfügt der Port über zwei *Ground*-Signale (Pin 5 und 11 nach Abbildung 3.3), sowie insgesamt acht logische Signale (siehe

Abbildung 3.3). Die bereits genannten *VCC*- und *Ground*-Pins können bis zu einem Ampere zur Verfügung stellen.[5]

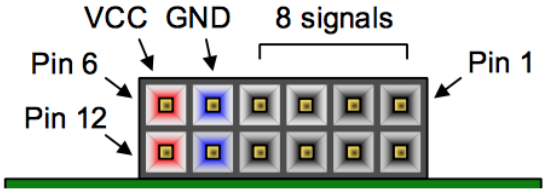


Abbildung 3.3: Übersicht der Pmod-Port-Pins und deren Funktion aus [5]

Um nun die korrekte Verbindung zum Board sicherzustellen, muss über das sogenannte *UCF*-File eine Pinzuweisung getätigt werden. Die benötigte Pinzuordnung lässt sich dem Datenblatt des Boardes entnehmen(siehe Abbildung 3.1)[5]

| **Pmod JA** | **Pmod JB** | **Pmod JC** | **Pmod JD** | **Pmod XDAC** |
|---|---|---|---|---|
| JA1: C17 | JB1: D14 | JC1: K1 | JD1: H4 | JXADC1: A13(AD3P) |
| JA2: D18 | JB2: F16 | JC2: F6 | JD2: H1 | JXADC2: A15(AD10P) |
| JA3: E18 | JB3: G16 | JC3: J2 | JD3: G1 | JXADC3: B16(AD2P) |
| JA4: G17 | JB4: H14 | JC4: G6 | JD4: G3 | JXADC4: B18(AD11P) |
| JA7: D17 | JB7: E16 | JC7: E7 | JD7: H2 | JXADC7: A14(AD3N) |
| JA8: E18 | JB8: F13 | JC8: J3 | JD8: G4 | JXADC8: A16(AD10N) |
| JA9: F18 | JB9: G13 | JC9: J4 | JD9: G2 | JXADC9: B17(AD2N) |
| JA10: G18 | JB10: H16 | JC10: E6 | JD10: F3 | JXADC10: A18(AD11N) |

Tabelle 3.1: Nexys4 DDR Pmod Pinzuordnung aus [5]

Listing 3.1 zeigt die Einträge des GPS-Sensors im *UCF*-File.

```
1  # GPSSensor ; --Connector JA low
2    NET "idGPS_3DF" LOC=D17 | IOSTANDARD=LVCMOS33; # JA7
3    NET "odGPStransmit" LOC=E17 | IOSTANDARD=LVCMOS33; # JA8
4    NET "idGPSreceive" LOC=F18 | IOSTANDARD=LVCMOS33; # JA9
5    NET "idGPS_PPS" LOC=G18 | IOSTANDARD=LVCMOS33; # JA10
6  #GPSSensorRST ; --Connector JB high
7    NET "oGPSReset" LOC=D14 |  IOSTANDARD=LVCMOS33; #JB1
```

Listing 3.1: Pmod-Port Pinzuweisung innerhalb des *UCF*-Files

In Listing 3.1 ist zusehen, dass hierbei sowohl der *JA*-, als auch der *JB*-Connector genutzt wurde. Die eigentliche Verbindung der einzelnen Signale des Pmod-GPS-Receivers werden über den *JA*-Connector realisiert. Hierfür werden jedoch nur sechs Pins benötigt, sodass lediglich die untere Reihe tatsächlich genutzt wurde. Den einzelnen Signalen wird ein Name, sowie der dazugehörige Pin zugewiesen.
Die Nutzung des *JB*-Connectors dient lediglich zu Realisierung eines Reset-Signales, womit der Pmod-GPS-Receiver sich später in der Anwendung resetten lässt.

### Anbindung an den PRHS-Bus

Für das Anbinden des GPS-Moduls an den PRHS-Bus wurde ein Modul angelegt, welches wiederrum aus zwei Komponenten besteht. Diese Anordnung ist in Abbildung 3.4 zu sehen, wobei die Untermodule im folgenden Kapitel näher in der Funktionsweise erläutert werden.

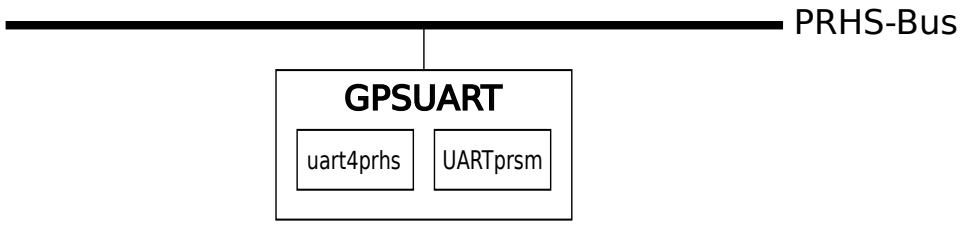


Abbildung 3.4: Aufbau des *GPSUART*-Moduls und dessen Anbindung

### UARTGPS

Die Abbildung 3.4 im vorherigen Kapitel zeigt das sogenannte *UARTGPS*-Modul. Dies wird im Folgenden näher erklärt. Die Einzelmodule werden dann in weiteren Unterkapiteln erläutert.

```
1  entity GPSUART is
2          [...]
3      port
4      (
5          [...]
6        -- serial Rx/Tx signals
```

```
7           idSerialIn      : in  std_logic;          --! Rx line of
                RS232Interface
8           odSerialOut     : out std_logic;          --! Tx line of
                RS232Interface
9
10          idPPS     : in std_logic; --!PPS
11          id3DF     : in   std_logic; --!3DF
12          ocRST     : out  std_logic --! Reset Controller
13      );
14  end entity;
```

Listing 3.2: Entity des *UARTGPS*-Moduls

Hierbei ist zu erkennen, dass die Signale *idSerialIn* und *odSerialOut* lediglich in dem *uart4prhs*-Modul (siehe Kapitel 3.1.1) verwendet werden. Die in Listing 3.2 angegebenen Signale (*idPPS*, *id3DF*, *ocRST*) werden hingegen nur im Modul für das externe Zurücksetzen(Reset) genutzt.(siehe Kapitel 3.1.1)

### UART-Register State Machine(*uart4prhs*)

Bei der Interaktion mit dem PRHS-Bus (siehe Kapitel 2.2) befindet sich der UART-Baustein grundsätzlich in zwei Zuständen. Dieser Baustein ist im genutzten System bereits vorhanden(siehe [7]). Entweder er wartet auf eine Anfrage vom *Master* (*wait4request*) oder er hat eine Anfrage bearbeitet (*requestDone*).

```
1  type uartRegFSMstates is (wait4request, requestDone)
2  signal sState : uartRegFSMstates;
```

Listing 3.3: Zustandsbeschreibung

Befindet er sich im erstgenannten Zustand, überprüft er bei jeder steigenden Taktflanke des Systemtaktes (*iSysClk*), ob eine Anfrage seitens des Masters vorliegt (*iePRHSrequest='1'*). Des Weiteren ist anhand der Adresse zu prüfen, an wen diese Anfrage gerichtet ist.

```
1   scRequest4me <= '1' when iePRHSrequest = '1' and ((icPRHSaddress and
        GEN_AddressMask) = GEN_BaseAddress)
2                   else '0';
```

Listing 3.4: Abgleichen der Adresse

Nachdem die Operation ausgeführt wurde, wechselt der Zustand(*sState*) wieder auf *wait4request* und setzt *rcDriveBus* auf *'0'*.
Befindet sich nun die passende Basisadresse auf dem *GEN_BaseAddress*-Signal, erkennt das System, dass die Anfrage an diese Interaktion gerichtet ist.

```
1              case icPRHSaddress(3 downto 0) is
2                  when "0000" =>
3                      rdPRHSdataSlave <= X"000000" & idDataRcvd;
4                      -- OPread or OPswap --> read from RxFifo
5                      if icPRHSoperation /= OPwrite then
6                          oeReadEn         <= '1';
7                      end if;
8                      -- OPwrite or OPswap --> write to  TxFifo
9                      if icPRHSoperation /= OPread then
10                         oeWriteEn        <= '1';
11                     end if;
```

Listing 3.5: Basisadresse + *Adressoffset='0'*

Nach Listing 3.5 werden, je nachdem welches *icPRHSoperation*-Signal auf dem Bus liegt, entweder Daten vom *Lese-FIFO* gelesen oder Daten in den *Schreibe-FIFO* geschrieben.

```
1                  when "0100" =>
2                      rdPRHSdataSlave <= sdStatus;
3                      if icPRHSoperation /= OPread then
4                          rIntDisable      <= idPRHSdataMaster(12);
5                      end if;
```

Listing 3.6: Basisadresse + *Adressoffset='4'*

Mit dem *Adressoffset='4'* bekommt das *rdPRHSdataSlave*-Signal den derzeitigen *sdStatus* zugewiesen und sofern es sich nicht um eine *OPread*-Operation handelt, wird das 12. Bit des *idPRHSdataMaster* an das Signal *'rIntDisable'* übergeben.

```
1                  when "1000" =>
2                      rdPRHSdataSlave <= rdBaudRate;
3                      if icPRHSoperation /= OPread then
4                          rdBaudRate <= idPRHSdataMaster;
5                      end if;
6                  when others =>
```

Listing 3.7: Basisadresse + *Adressoffset='8'*

Im dritten Fall (*Adressoffset='8'*) wird je nach Operation entweder die Baudrate an den Slave übergeben, oder die neue Baudrate vom Master gesetzt.

Daraus ergibt sich bei der Betrachtung von *icPRHSaddress* und *icPRHSoperation* folgende Tabelle 3.2.

| Adressoffset | Operation | Ereignis |
|---|---|---|
| $0x00_h$ | /= OPwrite | Lesen von Daten aus Lese-Fifo |
| | /= OPread | Schreibe von Daten in die Schreib-Fifo |
| $0x04_h$ | /=OPread | Eintrag ins Steuerregister |
| | | Statusabfrage des Slaves |
| $f0x08_h$ | /= OPread | Baudrate wird vom Master abgefragt |
| | | Baudrate des Slaves wird gesetzt |

Tabelle 3.2: Ereignistabelle für UARTrsm

### Externer Reset(*UARTprsm*)

Des Weiteren ist es möglich über die Adresse Basisadresse + *Adressoffset='100'* die übrigen Signale, wie *3DF*, *PPS*, *Reset-Controller* (siehe Kapitel 2.5), auszuwerten oder gegebenenfalls zu steuern. Die Funktionsweise des Automaten ist analog zu der in dem vorherigen Abschnitt beschriebenen. Hierbei wird jedoch immer das entsprechende Bit an die letzte Stelle von *rdPRHSdataSlave* angehangen.

```
1  when "0000" =>                          -- PPS
2    rdPRHSdataSlave <= X"0000000" & "000" & idPPS;
```

Listing 3.8: Abfrage des PPS-Signals über *Adressoffset='100'*

Wie in Listing 3.8 beschrieben, ist es möglich das PPS-Signal über den *Adressoffset='100'* auf das Signal *rdPRHSdataSlave* zu legen.
Ebenfalls ist es möglich, das 3D-Fix (3DF)-Signal auszuwerten, indem es über den *Adressoffset='104'* auf die Datenleitung des Slaves gelegt wird (siehe Listing 3.9).

```
1  when "0100" =>                          -- 3DF
2    rdPRHSdataSlave <= X"0000000" & "000" & id3DF;
```

Listing 3.9: Abfrage des 3DF-Signals über *Adressoffset='104'*

Die Hauptfunktion jedoch ist, über den *Adressoffset='8'* einen externen *Reset* zu realisieren. Dabei wird das entscheidene Bit hinten angehängt und an die Datenleitung des Slaves übertragen. Sollte jedoch über das *icPRHSoperation*-Signal die Operation *OP-write* aufgerufen werden, so sendet der Master den aktuellen Wert.

```
1                when "1000" =>                    -- Reset
2                  rdPRHSdataSlave <= X"0000000" & "000" & rcReset;
3                  if icPRHSoperation /= OPread then
4                    rcReset <= idPRHSdataMaster(0);
5                  end if;
```

Listing 3.10: Lesen und Setzen des *Reset*-Signals über *Adressoffset='108'*

Daraus ergibt sich folgende Ereignistabelle:

| Adressoffset | Operation | Ereignis |
|---|---|---|
| $0x100_h$ | - | Legen des PPS-Signals auf die Datenleitung des Slaves |
| $0x104_h$ | - | Legen des 3DF-Signals auf die Datenleitung des Slaves |
| $0x108_h$ | - | Legen des *Reset*-Signals auf die Datenleitung des Slaves |
| | /=OPread | Setzen des Resetbits |

Tabelle 3.3: Ereignistabelle für UARTprsm

### 3.1.2 GPS-Treiber

Wie bereits im Kapitel 2.8.1 erwähnt, benötigt jedes Gerät seinen eigenen Treiber, der dem *Betriebssystem* Informationen über das Gerät zur Verfügung stellt. Da bereits ein Baustein für UART zur Verfügung stand, wurde dieser lediglich in der Adressierung aktualisiert, da für den GPS-UART ein eigener Adressenbereich im *Device-Tree* angelegt wurde.
Der nachfolgende Code zeigt die Änderungen auf und erläutert diese.
Um die *OFFSET*-Adressierung anzupassen, wird ein *UART_GPS_OFFSET*-Makro mit dem zugehörigen Wert definiert.

```
1  #define UART_GPS_OFFSET 0x108
```

Listing 3.11: Anpassung des *Offset*-Bereichs im Treiber

Wie in Kapitel 2.8 erläutert, wird der GPS-Sensor resettet, beziehungsweise *aktiv ('1')* und *inaktiv ('0')* geschaltet, indem das ~*RST*-Signal beim Starten und Beenden gesetzt wird. Dieses Setzen ist im Kommentar nochmals angegeben.

```
1  static int PRHSgpsuart_startup(struct uart_port *port){
2    [...]
3   status = readl(port->membase + UART_STATUS_OFFSET);
4   writel(status & ~UART_INT_DISABLE, port->membase + UART_STATUS_OFFSET);
5   writel(1, port->membase + UART_GPS_OFFSET);  --Reset
6
7   return 0;
8  }
9
10 static void PRHSgpsuart_shutdown(struct uart_port *port)
11 {
12   [...]
13   status = readl(port->membase + UART_STATUS_OFFSET);
14   writel(0, port->membase + UART_GPS_OFFSET); -- Reset
15   writel(status | UART_INT_DISABLE, port->membase + UART_STATUS_OFFSET);
       /* disable Interrupts */
16   [...]
17 }
```

Listing 3.12: Anpassung des *Offset*-Bereichs im Treiber

### UART-Eintrag im Device-Tree

Um Geräte über die bereits genannten Schnittstellen (SPI,UART) betreiben und nutzen zu können, müssen Treiber vorhanden sein. Das folgende Listing zeigt den Eintrag in den Device-Tree(siehe Kapitel 2.8.1) für den UART-Baustein.

```
1  -- Copyright (C) 2014 Helmut-Schmidt-University
2  [...]
3 /dts-v1/;
4 /include/ "prhs-base.dtsi"
5
6 / {
7     model = "PRHS SoC on Nexys4 DDR";
8     compatible = "prhs,nexys4ddr";
9
```

```
10      [...]
11        uartgps: serial@f3000000 {
12            compatible = "prhs,prhs-gps-uart";
13            reg = <0xf3000000 0x1000>;
14            interrupts = <42>;
15            id = <1>;
16            interrupt-parent = <&intc>;
17            clocks = <&rootClock 0>;
18          };
```

Listing 3.13: Anlegen eines UART-Device im *Device-Tree*

Wie in Kapitel 3.13 zu sehen ist, wird eine serielle Schnittstelle an der Basisadresse *f3000000* erstellt. Zur eindeutigen Identifizierung wird das *Compatible* genutzt, dieses entspricht der *device_id*, welche im *prhs_gpsuart*-Treiber definiert wurde.
Das Speicherregister erhält eine Größe von '0x1000', beginnend mit der Startadresse.
Das sogenannte *interrupt*-Signal sorgt dafür, dass das Betriebssystem bei der Ausführung einer Anweisung immer noch reagieren kann, wenn ein wichtiges Ereignis eintritt, wie zum Beispiel das Eintreffen von Daten, die sonst verloren gehen würden. Bei den hier verwendeten *interrupts*, handelt es sich um sogenannte *Hardware-Interrupts*, welche genutzt werden um Daten von angeschlossenen Geräten zu empfangen. Dafür erhält jedes Peripheriegerät eine Interrupt Request (IRQ), welche das Gerät identifiziert.[16]

## 3.2 Anbindung des Beschleunigungsensors

Dieses Kapitel bezieht sich auf die Anbindung des in Kapitel 2.7 beschriebenen Beschleunigungssensors. Da der Sensor bereits auf dem Board implementiert ist, war keine Anbindung auf Seiten der Hardware nötig. Jedoch wird in Kapitel 3.2.1 die Anbindung des Sensors auf der Seite der Hardware näher erklärt. Im Kapitel 3.3 folgt dann die Beschreibung des zugehörigen Treibers.

### 3.2.1 Verbindung des Beschleunigungssensors mit dem FPGA

Wie bereits zu Beginn des Kapitels erklärt, benötigt der Beschleunigungssensor keine extra Verbindung zum Board. Dennoch zeigt die Abbildung 3.5 die *Pin*-Konfiguration des Bauteils.
Die Tabelle 3.4 zeigt die jeweiligen Funktionen der einzelnen Pins, die in Abbildung 3.5 dargestellt sind.

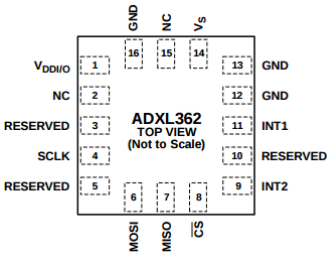


Abbildung 3.5: Pin Konfiguration nach [11]

| Pin Nr. | Mnemonic | Beschreibung |
|---|---|---|
| 1 | $V_{DDI/O}$ | Versorgungsspannung für Input/Output |
| 2 | NC | *No Connect.* Kein Anschluss. |
| 3 | Reserved | Reserviert. Ohne Anschluss oder auf Chassis Ground (GND) gelegt werden. |
| 4 | SCLK | SPI-Communication Clock, entspricht dem Serial Clock (SCK) im Kapitel 2.3.2 |
| 5 | Reserved | Reserviert. Ohne Anschluss oder auf GND gelegt werden. |
| 6 | Master Out-Slave In (MOSI) | Master Output - Slave Input. Serieller SPI-Dateneingang |
| 7 | Master In-Slave Out (MISO) | Master Input - Slave Output. Serieller SPI-Datenausgang |
| 8 | *CS* | Chip Select, entspricht im Kapitel 2.3.2 dem Slave Select (SS) |
| 9 | INT2 | Zweites Interrupt Signal.Ebenfalls Eingang für synchronisiertes Abtasten. |
| 10 | Reserved | Reserviert. Ohne Anschluss oder auf GND gelegt werden. |
| 11 | INT1 | Erstes Interrupt Signal. Ebenfalls für externes *clocking*. |
| 12 | GND | Masse (engl. chassis ground) |
| 13 | GND | Masse (engl. chassis ground) |
| 14 | $V_S$ | Versorgungsspannung |
| 15 | NC | *No Connect.* Kein Anschluss. |
| 16 | GND | Masse (engl. chassis ground) |

Tabelle 3.4: Pin-Funktions-Beschreibung nach [11]

### UCF-File

Für die Signalidentifikation bezüglich des FPGA-Boards, muss, wie beim GPS-Sensor, ein Eintrag in dem *UCF*-File erstellt werden. (siehe Kapitel 3.1.1)

```
1  # Accelerometer
2  NET "acl_miso" LOC=E15 | IOSTANDARD=LVCMOS33; #IO_L11P_T1_SRCC_15 MISO
3  NET "acl_mosi" LOC=F14 | IOSTANDARD=LVCMOS33; #IO_L5n_T0_AD9N_15 MOSI
4  NET "acl_sclk" LOC=F15 | IOSTANDARD=LVCMOS33; #IO_L14P_T2_SRCC_15 Serial
       Clock
5  NET "acl_csn"  LOC=D15 | IOSTANDARD=LVCMOS33; #IO_L12P_T1_MRCC_15 ~CS:
     Slave Select(Active Low)
6  NET "acl_int_one" LOC=B13 | IOSTANDARD=LVCMOS33; #IO_L2P_T0_AD8P_15 Int1
7  NET "acl_int_two" LOC=C16 | IOSTANDARD=LVCMOS33; ##IO_L20P_T3_A20_15
     Int2
```

Listing 3.14: Beschleunigungssensor Pinzuweisung innerhalb des *UCF*-Files

In Listing 3.14 werden alle Input/Output-Signale im *UCF*-File des Systems hinterlegt. Hierbei ist zu beachten, dass die beiden Interrupt-Signale zwar definiert werden, jedoch im weiteren Verlauf nicht genutzt werden.

### Anbindung an den PRHS-Bus

Wie bereits der UART-Baustein, wird der SPI-Baustein ebenfalls an den PRHS-Bus angebunden. Dabei besteht ein solches SPI-Modul aus einem Master und einem Slave, wie bereits im Grundlagenkapitel 2.3.2 erläutert. Ebenfalls ist dort beschrieben, dass für die Implementierung der bereits vorhandene SPI-Baustein genutzt wird.
Dabei werden die beiden *Register-State-Machines* zusammengeführt. Im Folgenden wird anhand des *SPI-Master* die Funktionsweise einer Datenübertragung via SPI verdeutlicht.

```
1   when WAITING =>    --Initial-Zustand
2                   if (icStart= '1') then
3                       odMOSI <= idByteWrite(Gen_DataLength - 1);
4                       state <= DATAPhaseA;
5                       ocReadyToSend <= '0';
6                       rcDataPos <= Gen_DataLength - 1;
```

Listing 3.15: Initialzustand des SPI-Master RSM

Dieser *Initial-Zustand* ist der Ausgangzustand, welcher am Anfang initialisiert wird oder wieder erreicht wird, nachdem eine Datenübertragung erfolgreich abgeschlossen wurde. Hierbei wird das Kommando vom Master an den Slave übermittelt, der zu diesem Zeitpunkt kein anderes Ereignis vorliegen hat.

```
1       when DATAPhaseA =>
2                       state <= DATAPhaseB;
3                       if (icCPHA = '0') then
4                         odByteRead(rcDataPos) <= idMISO;
5                       else
6                         odMOSI <= idByteWrite(rcDataPos);
7                       end if;
8                       SPIClk <= not SPIClk;
```

Listing 3.16: Byteübertragung des Masters in der SPI-Master RSM

Bei dem nächsten Zustand wird ein Bit vom Master an den Slave übermittelt, da es sich beim Übertragen von Daten per SPI um einen Austausch von Bits handelt.

Dabei wird unterschieden, ob das CPHA-Signal '*0*' ist oder nicht, da je nachdem die Daten bei steigender oder fallender Taktflanke eingelesen werden.

```
1  when DATAPhaseB =>
2          if (icCPHA = '0') then
3              odMOSI <= idByteWrite(rcDataPos-1);
4            else
5              odByteRead(rcDataPos) <= idMISO;
6            end if;
7            state <= DATAPhaseA;
8            [..]
9            if rcDataPos = 0 then
10             ocReadyToSend <= '1';
11             state <= WAITING;
12           else
13             rcDataPos <= rcDataPos - 1;
14           end if;
```

Listing 3.17: Datenübertragung des Slaves in der SPI-Master RSM

Hierbei wechselt der Zustand ständig zwischen *DATAPhaseA* und *DATAPhaseB*, da solange Bits übertragen werden, bis der 'Zeiger'(*rcDataPos*) den Wert '*0*' erreicht. Somit wird signalisiert, dass alle Bits übertragen wurden und der Zustand wechselt wieder in den Initialzustand.

In dem übergeordneten *spi4prhs*-Modul werden, je nach Adressoffset die einzelnen Register angesprochen. So ergibt sich aus der Offsetunterscheidung folgende Ereignistabelle:

| **Adressoffset** | **Funktion** | **Operation** | **Ereignis** |
|---|---|---|---|
| $0x00_h$ | Chip Select | OPwrite | Setzen des Chip-Select-Registers |
| $0x04_h$ | Clock Divider | OPwrite | Slave bekommt den Takt vom Master |
| $0x08_h$ | Lesen | OPread | Daten vom Slave lesen |
| | Schreiben | /=OPread | Wenn *ocReadyToSend='1'*, dann schreibt der Master und *requestinprogress* |
| $0x0C_h$ | Control | OPwrite | Setzen des Master von *Power on*, *CPOL* und *CPHA* über einzelne Bits des *idPRHSDataMaster* |
| | Status | /=OPwrite | Abrufen von Informationen von *CardDetect*, *WriteProtect*, *Power on*, *CPOL* und *CPHA* über Bits von *rdPRHSDataSlave* |

Tabelle 3.5: Ereignistabelle für SPIrsm

Innerhalb des *BaseReconfTop*-Moduls[7] wird dann das Modul als *Component* deklariert und es werden über die verschiedenen Bus-Signale die Verknüpfungen hergestellt. Hierbei beschränken sich die Änderungen auf die Zuweisung der Signale, welche spezifisch für den Beschleunigungssensor sind.(Siehe Listing 3.18)

```
1  spi4prhs1 : spi4prhs
2    generic map
3    (
4        GEN_BaseAddress  => X"f4000000" --! base address for register io
             mem mapping
5    )
6    port map
7    (
8        [...]
9        -- SPI SPEZIFISCH
10       odMOSI => acl_mosi, --Accelerometer-Signal
11       idMISO => acl_miso, --Accelerometer-Signal
12
13       oSClk => acl_sclk,
14       iClk => iClkG,
15        ocSS => acl_csn,
16
17        oGND1 => open,
18        icnCardDetect => '0',
19        icnWriteProtect => '0',
20        oVCC1 => open,
21        oGND2 => open,
22        oVCC2 => open,
23        DAT1 => open,
24        DAT2 => open
25        [...]
```

Listing 3.18: Zuweisung der Signale des Beschleunigungssensors

Hier werden die vorher im *UCF*-File deklarierten Signale der SPI-Schnittstelle zugewiesen.

## 3.3 Beschleunigungssensor-Treiber

Für den Treiber des Beschleunigungssensors wurde ein bereits vorhandener SPI-Treiber modifiziert und an den Sensor angepasst. Neben der Anpassung des *compatible* (siehe

Listing 3.19), werden, wie auch schon beim GPS-Sensor, Makros definiert, mit deren Hilfe eine Offsetanpassung stattfindet.

```
1  static struct of_device_id prhscommonfb_of_match[] = {
2    { .compatible = "prhs,prhs-accelmeter", },
3    [...]
```

Listing 3.19: Anpassung des Compatible innerhalb des Treibers

Des Weiteren wird die Struktur der *file_operations* in einem struct eindeutig hinterlegt.(siehe Listing 3.20)

```
1  [...]
2  .open = prhs_accel_open,
3  .release = prhs_accel_release,
4  .read = prhs_accel_read,
5  .write = prhs_accel_write,
6  .compat_ioctl = prhs_accel_ioctl,
7  .unlock_ioctl = prhs_accel_ioctl,
8  [...]
```

Listing 3.20: *file_operations*-Struct

In der Defintion werden Makros für die Signale *Chip-Select(CS)*, *Clock Divider(CLKDIV)* und *Befehl(COMMAND)* angelegt, um später diese einfacher nutzen zu können.

```
1  #define OFFSET_CS       0x00
2  #define OFFSET_CLKDIV   0x04
3  #define OFFSET_COMMAND 0x08
```

Listing 3.21: Definition von Makros

Zusätzlich zu den bereits vorhandenen Funktionen innerhalb des Treibers, werden nun zwei Funktionen (*AccelDataRead*,*AccelDataWrite*) hinzugefügt. In diesen Funktionen werden Unterfunktionen aufgerufen, sodass beim Schließen der Oberfunktion die Daten gelesen, beziehungsweise geschrieben werden.

```
1  int AccelDataRead(struct prhs_accel_privdata *drvdata, char z){
2 [...]
3     writel(0, drvdata->base + OFFSET_CS);          --Set CS 0=active
```

```
4       writel(0x0b, drvdata->base + OFFSET_COMMAND); --Set Instruction (0x0b
            read, 0x0a write)
5       writel(z, drvdata->base + OFFSET_COMMAND);    --Set Register Map
6       writel(0xff, drvdata->base + OFFSET_COMMAND); --Send Data to get Data
7       erg = readl(drvdata->base + OFFSET_COMMAND);  --Read Data
8       writel(1, drvdata->base + OFFSET_CS);         --Set CS 1=inactive
9     [...]
10    }
```

Listing 3.22: Funktion zum Lesen von Daten

In Listing 3.22 ist der genaue Ablauf einer Datenabfrage dargestellt. Nachdem der Slave über das *Chip-Select*-Signal ausgewählt wurde, wird zu erst die Instruktion übermittelt. In der 'Lesen'-Funktion ist es das Register mit dem Offset '*0x0b*'. Je nachdem ob nun X-,Y-,Z- oder Temperaturdaten ausgelesen werden sollen, wird nun das entsprechende Register per die Variable '*z*' übermittelt.(Register Map siehe Anhang)
Nachdem der Master den Slave ausgewählt und die Übertragung eingeleitet hat, werden die Daten synchron zum Takt des Masters an den Datenleitungen ausgegeben. Diese Bits werden als Byte dann in der Variable '*erg*' gespeichert und aus der Funktion zurückgegebenen.

```
1   int AccelDataWrite(struct prhs_accel_privdata *drvdata, char z,char y){
2 [...]
3       writel(0, drvdata->base + OFFSET_CS);              --Set CS 0=active
4       writel(0x0b, drvdata->base + OFFSET_COMMAND);      --Set Instruction
            (0x0b read, 0x0a write)
5       writel(z, drvdata->base + OFFSET_COMMAND);         --Set Register Map
6       writel(y, drvdata->base + OFFSET_COMMAND);         --Send Data to get
            Data
7       writel(1, drvdata->base + OFFSET_CS);              --Set CS 1=inactive
8     [...]
9     }
```

Listing 3.23: Funktion zum Schreiben von Daten

Der Unterschied zwischen Listing 3.22 und Listing 3.23 liegt darin, dass nun eine weitere Variable in die Funktion übergeben wird('*y*'). Im Gegensatz zu der *AccelDataRead*-Funktion, wird nun der Befehl in das gewählte Register ('*z*') geschrieben. Dieser wurde an die Funktion übergeben. Dies wird genutzt um beispielsweise Register zu resetten.[11]

### IOCTL()-Funktion

Mit der *ioctl()*-Funktion ist es möglich, den Treiber unter Linux aus dem *Userspace* direkt anzusprechen. Diese *Input-Output-Control*-Funktion ermöglicht eine gerätespezifische Steuerung und bietet über das Schreiben und Lesen hinaus die Möglichkeit zur Steuerung des Gerätes.[29]
Das Einbinden und die generelle Syntax sind in Kapitel 4 näher erklärt.

In der Funktion *prhs_accel_ioctl* wurden verschiedene Kommandos anhand einer Switch-Case-Funktion angelegt, welche je nach angegebener Register-Map-Adresse die gewünschte Operation ausführt.

```
1  switch (ioctl_num) {
2          case ACCEL_IOCTL_QUERY_DEV_ID: --Ausgabe der Device-ID (Register
               -Map-Adresse: 0x00)(ioctl_num=0x35)
3            [...]
4          case ACCEL_IOCTL_QUERY_XDATA: --Ausgabe der X-Achsen-Daten (
               Register-Map-Adresse: 0x08)(ioctl_num=0x36)
5            [...]
6          case ACCEL_IOCTL_QUERY_YDATA: --Ausgabe der Y-Achsen-Daten (
               Register-Map-Adresse: 0x09)(ioctl_num=0x37)
7            [...]
8          case ACCEL_IOCTL_QUERY_ZDATA: --Ausgabe der Z-Achsen-Daten (
               Register-Map-Adresse: 0x0A)(ioctl_num=0x38)
9            [...]
10         case ACCEL_IOCTL_QUERY_RST_COMMAND: --Reset (ioctl_num=0x39)
11           AccelDataWrite(drvdata,0x1F,0x52);--Soft-Reset (Register-Map-
                 Adresse: 0x1F)
12           AccelDataWrite(drvdata,0x1F,0x00);--Soft-Reset (Register-Map-
                 Adresse: 0x1F)
13           AccelDataWrite(drvdata,0x2D,0x02);--Power_CTL in Measure-Mode
                 (Register-Map-Adresse: 0x2D)
14           AccelDataWrite(drvdata,0x2C,0x14);--FILTER_CTL (Register-Map-
                 Adresse: 0x1F)
15           break;
16         case ACCEL_IOCTL_QUERY_TEMP: --Ausgabe der Temperatur-Daten (
               Register-Map-Adresse: 0x14 und 0x15)(ioctl_num=0x40)
17           [...]
```

Listing 3.24: Switch-Case-Anweisung innerhalb der Funktion *prhs_accel_ioctl*

Wie in Listing 3.24 zu sehen, wird innerhalb der Applikation (siehe Kapitel 4.3) über den Parameter *ioctl_num* die gewünschte Unterfunktion aufgerufen. So lässt sich über *ioctl_num*='0x39' der Sensor resetten und in den *Measure-Mode* schalten. Dies muss vor der ersten Nutzung geschehen. Danach können sowohl die *Device-ID*, als auch die Daten der Achsen und die Temperatur abgefragt werden.

### SPI-Eintrag im Device-Tree

Für die Kommunikation mit dem Board, muss, wie beim GPS-Sensor, ein Eintrag in den *Device-Tree* erstellt werden. Dieser Eintrag ist in Listing 3.25 angegeben, wobei die Bedeutung der einzelnen Parameter bereits in Kapitel 3.1.2 näher beschrieben wurden.

```
1 accelmeter: accel@f4000000{
2 #address-cells = <1>;
3 #size-cells = <0>;
4 compatible = "prhs,prhs-accelmeter";
5 reg = <0xf4000000 0x1000>;
```

Listing 3.25: Device-Tree Eintrag des SPI-Bausteins

# 4 Erprobung

Für die Erprobung wird auf dem Board das L4PRHS als Betriebssystem gebootet, welches zuvor über den SD-Kartenslot auf das Board gebracht wurde. Für die abschließende Erprobung und Demonstration wurde eine Applikation in der Programmiersprache C entwickelt.

## 4.1 GPS-Sensor Erprobung

Für die Erprobung des GPS-Sensors und zur Auswertung der Daten ist es erforderlich, zuerst den GPSd zu starten.(siehe Kapitel 2.6) Dies geschieht über den folgenden Aufruf:

```
1  gpsd -b -n  -D 2 /dev/ttyGPSPRHS1
```

Listing 4.1: Tatsächlicher Aufruf des GPSd in der Kommandozeile

Der genutzte *Device*-Port ist der */dev/ttyGPRSPRHS1*, welcher im System vorher angelegt und definiert wurde. Dieser ist bereits im genutzten System, wie in der Einführung zum *PRHS-Framework*(siehe Kapitel 2.2) erwähnt, vorhanden und wurde nur geringfügig verändert.[7]

Da das GPSd-Paket nicht nur aus dem Daemon an sich besteht, sondern auch aus diversen Applikationen, wie zum Beispiel dem *gpsmon* und dem *gpspipe*, ist es möglich die empfangenen Daten unmittelbar angezeigt zu bekommen, um diese Auswerten zu können.[24]

```
1  gpspipe -r /dev/ttyGPSPRHS1
```

Listing 4.2: Aufruf des *gpspipe* in der Kommandozeile

'*gpspipe*' bietet, wie der GPSd, eine Vielzahl an zusätzlichen Optionen beim Aufruf.[24] Um schnell Daten zu empfangen, beschränkt sich hier die Optionswahl auf '*-r*'(NMEA Datensätze werden ausgegeben) und den zugehörigen Port(/dev/ttyGPSPRHS1). Um die Syntax hinter dem Programm zu verstehen, wurde die Demo-Applikation des Herstellers analysiert. Anhand dieser Analyse war es möglich den Datenabgriff und die richtige Darstellung in der eigenen Applikation zu implementieren (siehe Kapitel 4.3).

## 4.2 Beschleunigungs-Sensor Erprobung

Die bereits im Kapitel 3.3 angesprochene *ioctl()*-Funktion wird nun für den Abruf der Daten genutzt. Diese wird in ein, auf das nötigste reduzierte C-Programm implementiert.

Für diese Implementierung ist das Einbinden der *ioctl*-Headerdatei nötig.

```
1 #include <sys/ioctl.h>
```

Listing 4.3: Einbinden der *ioctl()*-Headerdatei

Des Weiteren lässt sich die Syntax der *ioctl()*-Funktion wie folgt darstellen:

```
1  int ioctl(int fd,int request,...);
```

Listing 4.4: Einbinden der *ioctl()*-Headerdatei

Dabei bezeichnet '*fd*' den Parameter für den geöffneten Filedeskriptor mit dem eine Gerätedatei geöffnet wurde. Dieser ist eindeutig über den *Dateinamen* identifizierbar.
Der zweite Parameter stellt den entsprechenden Befehl dar, welcher vom Gerät abhängt. In der Erprobung ist dieser als *magic* bezeichnet. Der genutze Befehl '*0x40*' ist im Treiber des Beschleunigungssensors als Makro hinterlegt (siehe Kapitel 3.3) und soll für die Ausgabe des *X-Achsen*-Wertes sorgen.
Der dritte und letzte Parameter, welcher übergeben wird, ist *param*. Hierbei wird die Adresse einer Variablen, in diesem Fall *xData* übergeben. Diese Variable enthält nach dem Schließen der Funktion nun die gewünschten Daten.

```
1 [...]
2       int magic = strtol(argv[2], NULL, 16);
3       int param = strtol(argv[3], NULL, 16);
4       if ((fd = open(argv[1], O_RDWR)) < 0) {
```

```
5       perror("open");
6       return -1;
7         }
8         ret = ioctl(fd, magic, &param);
9         printf("ioctl return %d for magic=%x, param=%x\n", ret, magic, param
              );
10        while(1){
11          int xdata;
12          ioctl(fd, 0x40, &xdata);
13          printf("\r xdata = %03d", xdata);
14        }
15  [...]
```

Listing 4.5: Empfangen von Daten mithilfe der *ioctl()*-Funktion

In Listing 4.5 ist der zuvorbeschriebene Funktionsaufruf dargestellt. Dabei wird in einer dauerhaften *while*-Schleife immer wieder die *ioctl()*-Funktion aufgerufen, um die X-Achsen-Beschleunigung kontinuierlich dazustellen.

## 4.3 Demo-Applikation

Für die letztendliche Applikation, welche im Terminal aufgerufen wird und sowohl die Darstellung der GPS-Daten, als auch die des Beschleunigungssensors grafisch realisiert, werden nun beide Teile zusammengefügt.

### ncurses

Für die Text User Interface (TUI)-Umgebung wurde die freie C-Programmbibliothek *ncurses*(new curses) genutzt, welche eine Darstellung unabhängig vom darstellenden Textterminal garantiert. So ergeben sich im fertigen Programm zwei Fenster, in denen die Daten dargestellt werden.(siehe Listing 4.6)

```
1   //Building Monitor
2       initscr();
3       //Accelerometer
4       mvaddstr(0, 0, "+---Accelerometer---+");
5       mvaddstr(1, 0, "|X-Data:            |");
6       mvaddstr(2, 0, "|Y-Data:            |");
7       mvaddstr(3, 0, "|Z-Data:            |");
8       mvaddstr(4, 0, "|Temp:              |");
9       mvaddstr(5, 0, "|Temp.F:            |");
```

```
10        mvaddstr(6, 0, "+-------------------------+");
11      //GPSMonitor
12        mvaddstr(0, 27, "+-------------------GPS-------------------+");
13        mvaddstr(1, 27, "|Latitude:                               |");
14        mvaddstr(2, 27, "|Longitude:                              |");
15        mvaddstr(3, 27, "|Altitude:                               |");
16        mvaddstr(4, 27, "|Speed:                                  |");
17        mvaddstr(5, 27, "|Time:                                   |");
18        mvaddstr(6, 27, "+----------------------------------------+");
19        mvaddstr(8, 27, "+----------------------------------------+");
20        mvaddstr(9, 27, "|Status:                                 |");
21        mvaddstr(10, 27,"|Mode:                                   |");
22        mvaddstr(11, 27,"+----------------------------------------+");
```

Listing 4.6: Darstellung der TUI via ncurses

### Darstellung der GPS-Daten

Um nun die Daten des GPS-Sensors zu empfangen und in die *ncurses*-Umgebung einzupflegen, müssen diese Daten zuallererst in ein passendes struct(*gps_data*) gepackt werden.

```
1 struct gps_data_t gps_data;
```

Listing 4.7: Erstellen eines Structs für die GPS-Daten

Als nächstes wird mithilfe einer *if*-Abfrage sichergestellt, dass der GPSd aktiv ist und Daten an den Port *localhost* sendet. Ist dies nicht der Fall, wird das Programm mit einer Fehlerausgabe beendet.(Listing 4.8)

```
1   if ((rc = gps_open("localhost", "2947", &gps_data)) == -1) {
2       printf("code: %d, reason: %s\n", rc, gps_errstr(rc));
3       return EXIT_FAILURE;
4   }
```

Listing 4.8: Überprüfung ob der GPSd aktiv ist

Nun wird die GPSd-interne Funktion *gps_stream* aufgerufen und die Adresse des structs übergeben, welches am Anfang des Programms initialisiert wurde.

```
gps_stream(&gps_data, WATCH_ENABLE | WATCH_JSON, NULL);
```

Listing 4.9: Aufruf der *gps_stream*-Funktion und Übergabe der Adresse des structs

Alles Weitere nach dem Aufruf der Funktion wird nun innerhalb einer Endlosschleife(*while(1)*) implementiert. Nachdem die Daten des Beschleunigungssensor abgerufen wurden, folgt nun die Abfrage der GPS-Daten.

```
if (gps_waiting (&gps_data, 2000000)) { //wait for 2 seconds to receive
    data
    if ((rc = gps_read(&gps_data)) == -1) {    //read data
        printf("error occured reading gps data. code: %d, reason: %s\n",
            rc, gps_errstr(rc));
    } else {
    [...]
```

Listing 4.10: Abruf der GPS-Daten innerhalb einer Endlosschleife

Nun werden die im struct gespeicherten Daten in die *ncurses*-Umgebung eingebettet.

```
// Display data from the GPS receiver.
    if ((gps_data.status == STATUS_FIX) &&
        (gps_data.fix.mode == MODE_2D || gps_data.fix.mode == MODE_3D) &&
        !isnan(gps_data.fix.latitude) &&
        !isnan(gps_data.fix.longitude)) {
            [...]
            mvprintw(1,39, "%f",gps_data.fix.latitude);
            mvprintw(2,39,"%f",gps_data.fix.longitude);
            mvprintw(3,39,"%f",gps_data.fix.altitude);
            mvprintw(4,39,"%f",gps_data.fix.speed);
            mvprintw(5,34,"%s", asctime(gmtime(&test)));
            mvprintw(9,36,"Receiving Data!");
            switch(gps_data.fix.mode){
              case MODE_2D:
                mvprintw(10,36 , "2D");
                break;
              case MODE_3D:
                mvprintw(10,36 , "3D");
                break;
              default:
                mvprintw(10,36 , "NO FIX");
```

```
22                    break;
23                  }
24
25      } else {
26          mvprintw(9,36,"No GPS Data!");
27 [...]
28   refresh();
29   usleep(1000);
```

Listing 4.11: Implementierung der GPS-Daten in die *ncurses*-Umgebung

### Darstellung der Daten des Beschleunigungssensors

Die Daten des Beschleunigungssensors werden innerhalb der Endlosschleife kontinuierlich abgerufen, indem für die jeweilige Achse, beziehungsweise für die Temperatur die entsprechenden Parameter an die *ioctl()*-Funktion übergeben werden.
Bevor jedoch das Programm die *while*-Schleife erreicht, muss der Sensor in den *Measurement-Mode* geschaltet werden.(siehe dazu Kapitel 3.3)

```
1  // Reset and set Measurement-Mode
2  ioctl(fd, 0x39, &ret);
```

Listing 4.12: Aktivierung des *Measurement-Mode*

Wie zuvor beschrieben werden nun innerhalb der Schleife die jeweiligen Parameter nacheinander an die *ioctl()*-Funktion übergeben.

```
1 [...]
2       ioctl(fd, 0x36, &xdata);    //x-Data Function
3       ioctl(fd, 0x37, &ydata);    //y-Data Function
4       ioctl(fd, 0x38, &zdata);    //z-Data Function
5       ioctl(fd, 0x40, &temp);     //Temperature Function
6       tempf=temp*0.166667;  //The ADXL362 accelerometer temperature data
7                             // will have to be limited and scaled to
                                 pixels: 0 = 0C, 480 = 80C * 16,
```

Listing 4.13: Aktivierung des *Measurement-Mode*

Am Ende werden diese Daten in die *ncurses*-Umgebung implementiert.

```
1  mvprintw(1, 9 , "%03d", xdata);
2  mvprintw(2, 9, "%03d", ydata);
3  mvprintw(3, 9, "%03d", zdata);
4  mvprintw(4, 9, "%03d", temp);
5  mvprintw(5, 9, "%03d", tempf);
```

Listing 4.14: Aktivierung des *Measurement-Mode*

# 5 Zusammenfassung und Ausblick

In dieser Arbeit wird beschrieben, wie es ermöglicht wurde, per GPS und mithilfe des Beschleunigungssensors ADXL 362 eine Positionsbestimmung innerhalb des PRHS-Framework durchzuführen. Dazu wurden zuerst sämtliche, einzubindende Komponenten ausführlich beschrieben und dann auf der Hardwareseite angebunden.
Der GPS-Sensor wurde hierbei an einen sogenannten Pmod-Port angeschlossen und die vorhandene Pinkonfiguration in das System eingetragen. Des Weiteren wurde für den GPS-Sensor ein Modul entwickelt, welches per UART-Schnittstelle mit dem System kommunizieren kann. In diesem Modul wurden zwei Untermodule vereint und an den PRHS-Bus gekoppelt. Um nun die Kommunikation zwischen diesem Modul und dem System sicherzustellen, wurde ein Treiber entwickelt, der es ermöglicht, die vom Sensor gesendeten Daten zu verstehen und etwaigen Applikationen zur Verfügung zu stellen. Dabei kam unter anderem der GPSd zum Einsatz, der die Daten dem Betriebssystem, bereits im richtigen Format, zur Verfügung stellt.
Der Beschleunigungssensor, welcher sich bereits auf dem FPGA-Board befindet, wurde per SPI-Schnittstelle angebunden. Es wird ein neues SPI-Modul angelegt und implementiert. Für dieses wird ebenfalls ein Treiber entwickelt, welcher die Kommunikation herstellt. So kann das System die Daten empfangen und interpretieren. Nachdem Erprobungen für die einzelnen Sensoren getätigt wurden, wurde eine Demo-Applikation entwickelt, welche dem Nutzer per TUI die GPS- und Beschleunigungsdaten darstellt.
Es wurde erreicht, dass der Nutzer des Systems die GPS-Daten, wie zum Beispiel Längen- und Breitengrad, sowie die Höhe über Normalnull und Geschwindigkeit, nicht nur in der Applikation einsehen, sondern sie auch über den angelegten Deviceport weiter nutzen kann. Die Möglichkeit bietet der Beschleunigungssensor ebenfalls, welcher neben den Daten der drei Achsen, zusätzlich die Temperatur anzeigt und diese für weitere Anwendungen nutzbar sind.
So können diese Daten in der Zukunft in sämtlichen Programmen und Anwendungen genutzt werden, wie zum Beispiel zur Lokalisierung eines autonomen Systems. Um das gesamte Potenzial des Systems zu nutzen ist eine Weiterentwicklung möglich und nötig.

Der GPS-Sensor, als auch der Beschleunigungssensor bieten die Möglichkeit weitere Daten und Parameter zu empfangen und weiterzuverarbeiten. So stellt der GPS-Sensor, neben den oben genannten und in der Applikation angezeigten Daten, etliche weitere Parameter zur Verfügung. Diese sind im Anhang in Form des Datenblattes hinterlegt. Das bedeutet, dass sowohl der Umfang der Applikation, als auch die Einsatzgebiete dieser Sensoren deutlich vergrößert werden können.

# 3 Output Sentences

| Example | Description |
|---|---|
| $GPGGA | Message ID |
| 064951.000 | UTC Time (hhmmss.sss) |
| 2307.1256 | Latitude (ddmm.mmmm) |
| N | N/S indicator |
| 12016.4438 | Longitude (dddmm.mmmm) |
| E | E/W indicator |
| 1 | Position Fix Indicator |
| 8 | Satellites used |
| 0.95 | HDOP |
| 39.9 | MSL Altitude |
| M | Units |
| 17.8 | Geoidal Separation |
| M | Units |
| | Age of Diff. Corr. |
| *65 | Checksum |
| <CR><LF> | End of message indicator |

*Table 2. GGA.*

$GPGGA, 064951. 000, 2307. 1256, N, 12016. 4438, E, 1, 8, 0. 95, 39. 9, M, 17. 8, M, , *65<CR><LF>

| Example | Description |
|---|---|
| $GPGSA | Message ID |
| A | Mode1 (see GlobalTop manual) |
| 3 | Mode2 (see GlobalTop manual) |
| 29 | Satellite used (CH1) |
| 21 | Satellite used (CH2) |
| .... | |
| | Satellite Used (Ch12) |
| 2.32 | PDOP |
| 0.95 | HDOP |
| 2.11 | VDOP |
| *00 | Checksum |
| <CR><LF> | End of message indicator |

*Table 3. GSA.*

$GPGSA, A, 3, 29, 21, 26, 15, 18, 09, 06, 10, , , , ,  2. 32, 0. 95, 2. 11*00<CR><LF>

| Example | Description |
|---|---|
| $GPGSV | Message ID |
| 3 | Number of Messages |
| 1 | Message Number |
| 09 | Satellites in View |
| 29 | Satellite ID (CH1) |
| 36 | Elevation (CH1) |
| 029 | Azimuth (CH1) |
| 42 | SNR (C/No) |
| …. | |
| 15 | Satellite ID CH(4) |
| 21 | Elevation (CH4) |
| 321 | Azimuth (CH4) |
| 39 | SNR (C/No) |
| *7D | Checksum |
| <CR><LF> | End of message indicator |

*Table 4. GSV.*

$GPGSV,3,1,09,29,36,029,42,21,46,314,43, 26,44,020,43,15,21,321,39*7D<CR><LF>

| Example | Description |
|---|---|
| $GPRMC | Message ID |
| 064951.000 | UTC Time (hhmmss.sss) |
| A | Status (A = data valid) |
| 2307.1256 | Latitude (ddmm.mmmm) |
| N | N/S indicator |
| 12016.4438 | Longitude (dddmm.mmmm) |
| E | E/W indicator |
| 0.03 | Speed over ground (knots) |
| 165.48 | Course over ground (degrees) |
| 260406 | Date (ddmmyy) |
| 3.05 | Magnetic Variation (degrees) |
| W | E/W indicator |
| A | Mode (see GlobalTop manual) |
| *55 | Checksum |
| <CR><LF> | End of message indicator |

*Table 5. RMC.*

$GPRMC, 064951. 000, A, 2307. 1256, N, 12016. 4438, E, 0. 03, 165. 48, 260406, 3. 05, WA*55<CR><LF>

| Example | Description |
|---|---|
| $GPVTG | Message ID |
| 165.48 | Course (degrees) |
| T | Reference (true or false) |
| | Course (degrees) |
| M | Reference (Magnetic) |
| 0.03 | Speed |
| N | Units (N = knots) |
| 0.06 | Speed |
| K | Units (K = km/hr) |
| A | Mode (see GlobalTop manual) |
| *37 | Checksum |
| <CR><LF> | End of message indicator |

*Table 6. VTG.*

$GPVTG,165.48,T,,M,0.03,N,0.06,K,A*37<CR><LF>

# REGISTER MAP

**Table 11. Register Summary**

| Reg | Name | Bits | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Reset | RW |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x00 | DEVID_AD | [7:0] | DEVID_AD[7:0] | | | | | | | | 0xAD | R |
| 0x01 | DEVID_MST | [7:0] | DEVID_MST[7:0] | | | | | | | | 0x1D | R |
| 0x02 | PARTID | [7:0] | PARTID[7:0] | | | | | | | | 0xF2 | R |
| 0x03 | REVID | [7:0] | REVID[7:0] | | | | | | | | 0x01 | R |
| 0x08 | XDATA | [7:0] | XDATA[7:0] | | | | | | | | 0x00 | R |
| 0x09 | YDATA | [7:0] | YDATA[7:0] | | | | | | | | 0x00 | R |
| 0x0A | ZDATA | [7:0] | ZDATA[7:0] | | | | | | | | 0x00 | R |
| 0x0B | STATUS | [7:0] | ERR_USER_REGS | AWAKE | INACT | ACT | FIFO_OVER-RUN | FIFO_WATER-MARK | FIFO_READY | DATA_READY | 0x40 | R |
| 0x0C | FIFO_ENTRIES_L | [7:0] | FIFO_ENTRIES_L[7:0] | | | | | | | | 0x00 | R |
| 0x0D | FIFO_ENTRIES_H | [7:0] | UNUSED | | | | | | FIFO_ENTRIES_H[1:0] | | 0x00 | R |
| 0x0E | XDATA_L | [7:0] | XDATA_L[7:0] | | | | | | | | 0x00 | R |
| 0x0F | XDATA_H | [7:0] | SX | | | | | XDATA_H[3:0] | | | 0x00 | R |
| 0x10 | YDATA_L | [7:0] | YDATA_L[7:0] | | | | | | | | 0x00 | R |
| 0x11 | YDATA_H | [7:0] | SX | | | | | YDATA_H[3:0] | | | 0x00 | R |
| 0x12 | ZDATA_L | [7:0] | ZDATA_L[7:0] | | | | | | | | 0x00 | R |
| 0x13 | ZDATA_H | [7:0] | SX | | | | | ZDATA_H[3:0] | | | 0x00 | R |
| 0x14 | TEMP_L | [7:0] | TEMP_L[7:0] | | | | | | | | 0x00 | R |
| 0x15 | TEMP_H | [7:0] | SX | | | | | TEMP_H[3:0] | | | 0x00 | R |
| 0x16 | Reserved | [7:0] | Reserved[7:0] | | | | | | | | 0x00 | R |
| 0x17 | Reserved | [7:0] | Reserved[7:0] | | | | | | | | 0x00 | R |
| 0x1F | SOFT_RESET | [7:0] | SOFT_RESET[7:0] | | | | | | | | 0x00 | W |
| 0x20 | THRESH_ACT_L | [7:0] | THRESH_ACT_L[7:0] | | | | | | | | 0x00 | RW |
| 0x21 | THRESH_ACT_H | [7:0] | UNUSED | | | | | THRESH_ACT_H[2:0] | | | 0x00 | RW |
| 0x22 | TIME_ACT | [7:0] | TIME_ACT[7:0] | | | | | | | | 0x00 | RW |
| 0x23 | THRESH_INACT_L | [7:0] | THRESH_INACT_L[7:0] | | | | | | | | 0x00 | RW |
| 0x24 | THRESH_INACT_H | [7:0] | UNUSED | | | | | THRESH_INACT_H[2:0] | | | 0x00 | RW |
| 0x25 | TIME_INACT_L | [7:0] | TIME_INACT_L[7:0] | | | | | | | | 0x00 | RW |
| 0x26 | TIME_INACT_H | [7:0] | TIME_INACT_H[7:0] | | | | | | | | 0x00 | RW |
| 0x27 | ACT_INACT_CTL | [7:0] | RES | | | LINKLOOP | INACT_REF | INACT_EN | ACT_REF | ACT_EN | 0x00 | RW |
| 0x28 | FIFO_CONTROL | [7:0] | UNUSED | | | | AH | FIFO_TEMP | FIFO_MODE | | 0x00 | RW |
| 0x29 | FIFO_SAMPLES | [7:0] | FIFO_SAMPLES[7:0] | | | | | | | | 0x80 | RW |
| 0x2A | INTMAP1 | [7:0] | INT_LOW | AWAKE | INACT | ACT | FIFO_OVER-RUN | FIFO_WATER-MARK | FIFO_READY | DATA_READY | 0x00 | RW |
| 0x2B | INTMAP2 | [7:0] | INT_LOW | AWAKE | INACT | ACT | FIFO_OVER-RUN | FIFO_WATER-MARK | FIFO_READY | DATA_READY | 0x00 | RW |
| 0x2C | FILTER_CTL | [7:0] | RANGE | | RES | HALF_BW | EXT_SAMPLE | ODR | | | 0x13 | RW |
| 0x2D | POWER_CTL | [7:0] | RES | EXT_CLK | LOW_NOISE | | WAKEUP | AUTOSLEEP | MEASURE | | 0x00 | RW |
| 0x2E | SELF_TEST | [7:0] | UNUSED | | | | | | | ST | 0x00 | RW |

# Literaturverzeichnis


[1] Abel, D. M. H.: *GPS: Global Positioning System. Funktionsweise und mathematische Grundlagen.* S. 57–60, 2001. https://www2.hs-esslingen.de/ abel/gps/Abel-gps.htm.

[2] Büch, C.: *SPI - Serial Peripheral Interface*, Juni 2006. https://www.uni-koblenz.de/~physik/informatik/MCU/SPI.pdf.

[3] Cömert, E.: *PModCLS - UART Interface with VHDL*, 2011.

[4] Digilent: *Nexys4 DDR Artix-7 FPGA*, 2016. http://store.digilentinc.com/nexys-4-ddr-artix-7-fpga-trainer-board-recommended-for-ece-curriculum/.

[5] Digilent: *Nexys4 DDR FPGA Board Reference Manual*, Apr. 2016. https://reference.digilentinc.com/_media/pmod:pmod:pmodgps_rm.pdf.

[6] Digilent: *PModGPS Reference Manual*, Apr. 2016. https://reference.digilentinc.com/_media/pmod:pmod:pmodgps_rm.pdf.

[7] Eckert, M.: *FPGA-Based System Virtual Machines.* Dissertation, Helmut Schmidt Universität - Universität der Bundeswehr Hamburg, 2014.

[8] Electronics, D. K.: *Linx Technologies Inc. ANT-GPS-SH-SMA*, Nov. 2016. http://www.digikey.com/product-detail/en/linx-technologies-inc/ANT-GPS-UC-SMA/ANT-GPS-UC-SMA-ND/613967.

[9] Electronics, D. K.: *Linx Technologies Inc. CONSMA003.062*, Nov. 2016. http://www.digikey.com/product-detail/en/linx-technologies-inc/CONSMA003.062/CONSMA003.062-ND/1577208.

[10] Engelke, M.: *Anbindung eines AC-97 Codecs an das PRHS Framework*, Jan. 2015.

[11] Inc., A. D.: *ADXL362*, 2012. https://analog.com/media/en/technical-documentation/data-sheets/ADXL362.pdf.

[12] Inc., G. T.: *FGPMMOPA6h GPS Standalone Module Data Sheet*, 2011. https://reference.digilentinc.com/_media/reference/pmod/pmodgps/globaltop-fgpmmopa6h-datasheet-v0a.pdf.

[13] Inc., M. E.: *MEMS Beschleunigungsmesser - ADI.* https://mouser.de/new/Analog-Devices/adi-mems-accelerometers/?gclid=CKm3nsu359ACFUu6GwodITABGw.

[14] Instruments, N.: *Wie funktionieren FPGAs?*, Mai 2013. http://www.ni.com/white-paper/6983/de/.

[15] ITWissen.info: *GPS-Frequenz :: GPS Frequency*, Nov. 2016. http://www.itwissen.info/definition/lexikon/GPS-Frequenz-GPS-frequency.html.

[16] Kirk, A.: *Interrupt.* https://computerlexikon.com/was-ist-interrupt.

[17] Korsgaard, P.: *The Buildroot user manual.* https://buildroot.org/downloads/manual/manual.pdf.

[18] Mikrocontroller.net: *FPGA.* http://www.mikrocontroller.net/articles/FPGA.

[19] N, A., Joseph, G., Oommen, S. S. und Dhanabal, R.: *Design and implementation of a high speed Serial Peripheral Interface.* 2014.

[20] Noisefloor: *Ubuntuusers Wiki*, Mai 2016. https://wiki.ubuntuusers.de/Dateisystem.

[21] Plate, P. J.: *Schnittstellen : Serielle Schnitstelle, USB. Feldbusse, SPI, $I^2C$, 1-Wire etc.* November 2016. http://www.netzmafia.de/skripten/hardware/Control/schnittstellen.pdf.

[22] Pommersheim, A., Rayapati, P. und Yuan, C.: *CLB Block Diagram.* http://venividiwiki.ee.virginia.edu/mediawiki/images/c/ca/.

[23] Quade, J. und Kunst, E. K.: *Linux-Treiber entwickeln.* dpunkt.Verlag, 2015, ISBN 9783864902888.

[24] Raymond, E. S.: *gpsd - a GPS service daemon*, Sep. 2016. http://catb.org/gpsd.

[25] Tanenbaum, A. S.: *Modern Operating Systems.* Pearson Education Inc., 2009, ISBN 9780138134594.

[26] Trimberger, S. und Gertl, S.: *Die vier Zeitalter der FPGA-Entwicklung*, Juli 2016. http://www.elektronikpraxis.vogel.de/meilensteine-der-elektronik/articles/541059/.

[27] Verle, M.: *Architecture and programming of 8051 MCUs*. MikroElektronika, Jan. 2007, ISBN 9788684417093.

[28] Wietzke, J.: *Embedded Technologies*. Springer-Verlag, 2012, ISBN 9783642239960.

[29] Wolf, J.: *Linux-Unix-Programmierung*. Rheinwerk Computing, 2006, ISBN 3898427498.

[30] Wolfinger, C.: *Keine Angst vor Linux/Unix: Ein Lehrbuch für Linux- und Unix-Anwender*. Springer-Verlag, Berlin Heidelberg, 2013, ISBN 9783642320791.

[31] Yenigül, I., Siseci, N. E., Taskiran, A. und Kaya, M.: *Unix Daemon Server Programming*, Mai 2001. http://enderunix.org/docs/eng/daemon.php.