Inhaltsverzeichnis

1 Einleitung

1.1 Stand der Technik

Das Projekt *deep*[1] ist eine Cross Development Plattform, die es erlaubt, ein Java Programm direkt auf einem Prozessor auszuführen. Es ermöglicht einem Entwickler eine Java Programm zu schreiben, welches direkt auf einem Prozessor läuft und Echtzeit-Fähigkeiten hat. Zur Zeit wird dieses Projekt in der NTB für die Ausbildung Von Systemtechnik Studenten verwendet. Es erlaubt einfach und schnell Robotersteuerungen und Regelungen zu implementieren, ohne dass man sich mit den Eigenarten von C und C++ Programmen auseinandersetzen muss.

deep unterstützt einige grundlegende Debuging-Funktionalitäten. Mit einer mehreren tausend Franken teuren Abatronsonde kann der Speicher und die Register des Prozessors ausgelesen und auch geschrieben werden. Der aktuelle Debugger unterstützt keine *Breakpoints* oder *Source Code Navigation*, wie man es aus bekannten Debuggern wie dem *gdb*[2] kennt.

1.2 Motivation

Aktuell ist *deep* nur mit der PowerPC Architektur kompatibel. PowerPC Prozessoren sind aber nicht mehr weit verbreitet und sehr teuer. Die an der NTB verwendeten PowerPC-Prozessoren sind zwar leistungsstark, aber teuer und veraltet.

Aus diesem Grund wird *deep* für die ARM-Architektur erweitert. Da die ARM-Architektur bei eingebetteten Prozessoren am weitesten verbreitet ist, ist auch die Auswahl an günstiger und leistungsstarker Hardware sehr gross. Mit grosse Flexibilität bei der Auswahl von ARM-Prozessoren können sehr günstige oder auch sehr leistungsstarke Prozessoren verwendet werden.

[1] http://www.deepjava.org/start
[2] https://www.gnu.org/software/gdb/

deep ist ein Open-Source-Projekt welches auch für den Unterricht verwendet wird. Damit nicht für jeden Student teure Debugging-Hardware gekauft werden muss, ist eine kostengünstige Alternative wünschenswert.

Java ist im Gegensatz zu C und C++ eine sehr zielorientierte Sprache. Bei Java muss man sich nicht so detailliert um Ressourcen wie Speicher und Hardwareschnittstellen kümmern wie in C-orientierten Sprachen. Dieser Aspekt soll auch beim Debugger beibehalten werden. Zusätzlich zum direkten Speicher Auslesen sollen auch Variablen gelesen und geschrieben werden können. Eine native *Source Code Navigation* in Eclipse vereinfacht die Entwicklung einer *deep*x-Applikation sehr.

1.3 Zielsetzung

Bei dieser Arbeit werden mehrere Ziele verfolgt, die aufeinander aufbauen.

1. Passende Hardware (Experimentierboard) finden, welche auch im Unterricht verwendet werden kann.

2. Grundlegendes Debug-Interfeace, welches bereits für PowerPC existiert, für die ausgewählte Hardware anpassen. Dieses Interface soll für die Entwicklung von deep möglichst bald einsatzbereit sein.

3. Den GNU-Debugger (gdb) mit einem Programm verwenden, dass vom deep-Compiler übersetzt wurde. Dazu soll vorerst das Command-Line-Interface (CLI) des gdb genutzt werden.

4. Den gdb in das Eclipse Plug-In von deep integrieren, damit der Debugger direkt aus Eclipse verwendet werden kann.

2 Auswahl der Hardware

2.1 Einleitung

Die Auswahl von Hardware mit ARM Prozessoren ist extrem gross. Ende September 2016 sind bereits über 86 Milliarden ARM basierte Prozessoren verkauft worden.[1] Diese Zahl reflektiert zwar nicht direkt die Diversität von den verschiedenen Prozessoren, aber sie zeigt recht gut wie enorm weit ARM Prozessoren verbreitet sind.

In diesem Kapitel soll in dem riesigen Angebotsdschungel die richtige Hardware ausgewählt werden, auf der diese Arbeit aufbauen kann. Die ausgewählte Hardware soll nicht nur für diese Arbeit genutzt werden, sondern auch für den Robotik Unterricht. Zusätzlich sollte der Prozessor auch leistungsstark und auch flexibel genug sein, um ihn in anspruchsvollen Robotikprojekten verwenden zu können.

2.2 Soll- und Muss-Kriterien bei der Auswahl der Hardware

Für die Hardware sind folgende Soll- und Muss-Kriterien ermittelt worden.

2.2.1 Muss-Kriterien

- Systemebene
  - FPGA: Der Prozessor muss mit einem FPGA kommunizieren können.
  - Hardware Debugger: Der Prozessor muss für die Entwicklung von *deep* einen Hardware Debugger wie beispielsweise das JTAG Interface BDI3000[2] von Abatron unterstützen.

[1] Elektronischer/Anhang/ARM-media-fact-sheet-2016.pdf
[2] http://www.abatron.ch/fileadmin/user_upload/news/BDI3000-Brochure.pdf

  - Günstiger Programmierer: Wenn zusätzliche Hardware benötigt wird um die *deep*-Applikation auf das Target zu schreiben, dann muss diese möglichst günstig sein.
  - Grosses Ökosystem: Das ausgewählte Produkt muss von einem grossen Ökosystem unterstützt werden. Aussterbende Produkte oder Nischenprodukte sind nicht akzeptabel.
  - Als fertiges Modul erhältlich: Eigenes PCB entwickeln und herstellen ist keine Option.
  - Einbettbar: Der Prozessor muss auch bei einem selbst entwickelten PCB verwendet werden können. Wahlweise als SOM (*System On Module*) oder direkt als Prozessor in eigenem Package.
  - Noch lange erhältlich.

- Prozessorebene
  - ARMv7: Der Prozessor muss auf einer ARMv7 ISA (*Instruction Set Architecture*) basieren.
  - ARM Instruktionen: Der Prozessor muss ARM Instruktionen unterstützen. *Thumb* Instruktionen sind nicht ausreichend.
  - FPU (*Floating Point Unit*): Für Gleitzahlenarithmetik.
  - Netzwerkschnitstelle: RJ-45 inklusive MAC[3] und *Magnetics*.
  - USB: USB Schnittstelle als Host und als Slave.
  - Flash: Mehr als 50kByte Flash.
  - RAM: Mehr als 100kByte RAM.

[3] Media Access Control

2.2.2 Soll-Kriterien

- Systemebene
  - Einfach einbettbar: Der Prozessor ist als Prozessormodul erhältlich, so dass das Design von einem selbst entwickelten PCB einfacher wird.
  - Günstiger Hardwaredebugger: Der Hardwaredebugger kann auch für Applikationsentwicklung mit *deep* eingesetzt werden.
  - Möglichst schneller Download der Applikation.
- Prozessorebene
  - Memory Mapped Bus für FPGA Schnittstelle.
  - FPU unterstützt *Double Precision*.
  - Integerdivision
  - Prozessortakt über 500MHz.

2.3 Hardware Debugger

Der Begriff *Hardware Debugger* ist nicht eindeutig definiert. Im einfachsten Fall kann ein Hardware Debugger nur ein *Boundary Scan* durchführen wie es ursprünglich für JTAG vorgesehen war. Bei *Boundary Scan* können die I/O Pins von einem Prozessor gelesen und auch gesetzt werden. Mit so einem Scan kann in der Produktion bei der Bestückten PCBs überprüft werden, ob alle Lötstellen Kontakt herstellen und dabei keine Kurzschlüsse bilden. Für diesen Scan wird der Prozessor Kern nicht verwendet, sondern separate Peripherie im Prozessor. Über das JTAG Interface kann der Scan ausgeführt werden, ohne dass eine Software auf dem Prozessor ausgeführt werden muss.

Moderne Prozessoren erweitern diese grundlegende Funktionen mit einigen sehr hilfreichen Features. So bieten ARM Prozessoren mit der *CoreSight* Technologie noch viel mehr als nur einen *Boundary*

Scan. Die untenstehende Liste zeigt einige Funktionen von dieser Technologie, aber nicht alle. Die für diese Arbeit relevanten Funktionen sind **fett** geschrieben.

- **Prozessor Register lesen und schreiben**
- **RAM lesen und schreiben**
- **Externer Flash Speicher lesen und schreiben**
- **Hardware Breakpoint auf den Program Counter**
- **Hardwaer Breakpoint auf einer Speicherstelle (Watchpoint)**
- Debug Trace (ETM Program Trace)
- Debug Trace Buffer

Da ein Hardware Debugger keine funktionsfähige Software auf dem Prozessor benötigt, kann er auch gut verwendet werden, um die grundlegendsten Funktionen, wie beispielsweise der Bootvorgang, vom *deep* Laufzeit System zu entwickeln.

2.4 Übersicht über die ARM Mikroarchitekturen

2.4.1 Cortex-A

Sehr gut geeignet für die Verwendung mit einem vollen Betriebssystem wie Windows, Linux oder Android. Cortex-A Prozessoren bieten dem umfangreichsten Support für externe Peripherie wie USB, Ethernet und RAM. Sie sind auch leistungsstärksten ARM Cortex Prozessoren.

2.4.2 Cortex-R

Cortex-R werden entwickelt für Echtzeitanwendungen und Sicherheitskritische Applikationen wie Festplattenkontrolle und medizinische Geräte. Sie sind normalerweise nicht mit einer MMU *Memory Management Unit* ausgerüstet. Mit einer Taktrate von über 1GHz und einem sehr schnellen Interruptverhalten eignen sich Prozessoren mit einem Cortex-R sehr gut um auf externe Stimuli schnell zu reagieren.

2.4.3 Cortex-M

Cortex-M sind mit einer Taktrate um 200Mhz relativ langsam. Sehr stromsparend und durch die kurze Pipeline haben sie eine deterministische und kurze Interrupt Verzögerung. Die Prozessoren aus der Cortex-M Reihe unterstützen nur die Thumb-Instruktionen und kommen deshalb nicht in Frage.

2.4.4 ARM Prozessoren ausserhalb der Cortex Reihe

Seit 2004 werden die meisten Kerne in eine der Cortex Gruppen eingeteilt. Ältere Kerne, sogenannte *"Classic cores"*, haben Namen wie z.b. ARM7 oder ARM1156T2F-S. Da solche Designs meist aus einer Zeit vor 2004 stammen, gilt das Design als veraltet und wird bei dieser Arbeit nicht berücksichtigt.

2.4.5 Fazit über die ARM Mikroarchitekturen

Prozessoren die auf der Cortex-A Mikroarchitektur basieren bieten die grösste Flexibilität. Zusätzlich ist auch das Angebot bei den Cortex-A Prozessoren am grössten. Die anderen Cortex Reihen bieten keine Vorteile, die für dieses Projekt von Nutzen sind. Aus diesen Gründen wird die Auswahl auf die Prozessoren aufs der Cortex-A Reihe begrenzt.

Tabelle 2.1: Übersicht ARM Mikroarchitekturen

	Vorteile	Nachteile
A	* Sehr Leistungsstark * Support für vollwertige Betriebssysteme * Grosse Variation erhältlich (Energiesparend / sehr Leistungsstark) * Reichhaltiger Funktionsumfang * NEON und FPU Unterstützung	* Langsamer Context-Switch * Relativ hoher Stromverbrauch * Relativ teuer * Mit GPU erhältlich * Keine DSP Unterstützung * Keine HW-Division
B	* Sehr gut geeignet für Echtzeitanwendungen * Sehr schneller Context-Switch * DSP Unterstützung	* Kleiner Funktionsumfang * Nicht so leistungstark wie Cortex A * Keine Linux Unterstützung
C	* Sehr schneller Context-Switch * Sehr energiesparend * DSP Unterstützung	* Geringe Rechenleistung * Keine Linux Unterstützung * Unterstützt nur Thumb-Instruktionen

2.5 Anbindung des FPGAs

2.5.1 Einleitung

FPGAs haben typischerweise einen sehr hohen *Pin-Count* und werden in *BGA-Packages* ausgeliefert.

Es gibt verschiedene Möglichkeiten, wie ein FPGA mit einem Prozessor verbunden werden kann. Die Vor- und Nachteile der verschiedenen Bauarten werden in diesem Kapitel abgewogen.

2.5.2 FPGA als Zusatzplatine zum Prozessorboard - Bauweise "Modular"

Das *FPGA Development Board CAPE for the BEAGLEBONE*[4] ist eine Aufsteckplatine für den *Beaglebone Black*. Wenn es auf den *Beaglebone Black* aufgesteckt wird, erweitert es den ARM basierten Linux PC um *Spatran 6 LX9* FPGA inklusive einiger I/O-Peripherie und SDRAM.

Vorteile:

- Relativ günstig.
- Funktioniert "Out of the Box"
- Schnelles GPMC[5] Interface (bis zu 70 MB/s) zwischen Prozessor und FPGA.

[4] https://www.element14.com/community/docs/DOC-69215/l/fpga-development-board-cape-for-the-beaglebone
[5] General-Purpose Memory Controller

Tabelle 2.2: Übersicht Bauformen

Bauweise	Vorteile	Nachteile
Modular	* Günstig wenn nur Prozessor verwendet wird * Unterschiedliche FPGAs können verwendet werden	* Datenbus evt. nicht Memory mapped
SOB	* Sauberes, abgeschlossenes System	* FPGA ist fix
SOC	* Potenziell sehr schnelle Datenverbindung zwischen FPGA und Prozessor * Sauberes, abgeschlossenes System	* FPGA ist fix * Relativ teuer
FPGA	* Flexibel	* Sehr teuer

Nachteile:

- Verwendet ein modifiziertes Linux-Image, das LOGI-Image.
- Der eMMC[6] Speicher des Beaglebone kann nicht gleichzeitig mit dem GPMC verwendet werden.
- Die Verfügbarkeit vom Cape ist nicht garantiert.
- Nur ein FPGA und Prozessor erhältlich.

Fazit - Bauweise "Modular"

2.5.3 FPGA auf dem gleichen Board wie der Prozessor (System On Board) - Bauweise "SOB"

2.5.4 FPGA im gleichen Gehäuse wie der Prozessor (System On Chip - Bauweise "SOC"

2.5.5 ARM als Softcore in FPGA - Bauweise "FPGA"

STM23

STM

[6] Embedded Multi Media Card

3 System

3.1 Einleitung

Dieses Kapitel bietet eine grobe Übersicht über das ganze System, um die Zusammenhänge zwischen einzelnen Komponenten aufzuzeigen. Auf einzelne Komponenten wird in den folgenden Kapitel genauer eingegangen.

3.2 Schematische Übersicht

In der Abbildung 3.1 ist das ganze System abgebildet. Auf dem *Windows PC* wird die *deep*-Applikation in Eclipse geschrieben, kompiliert und debuggt. Plug-Ins erweitern Eclipse um die notwendige Funktionalitäten, die für die Entwicklung von Deep Applikationen notwendig sind. Es sind beide Debug Toolchains, die ”klassische” Abatron Toolchain und die neue OpenOCD Toolchain in dieser Übersicht abgebildet.

Bei der Abatron Toolchain wird das *Abatron BDI 3000* über die rote TCP/IP Verbindung angesprochen. Das BDI kommuniziert dann über eine JTAG Verbindung direkt mit dem Zynq Chip.

Die grünen Pfeile zeigen den Kommunikationsweg für die neuen OpenOCD-Toolchains. OpenOCD bildet zusammen mit der richtigen Hardware, hier ist es der FT2232 Chip, einen kompletten Debugger und ist somit eine Alternative zum BDI. OpenOCD stellt einen GDB Server und auch ein CLI (*Command Line Interface*) zur Verfügung. Das Eclipse Plugin *openOCDInterface* verwendet das CLI über den TCP/IP Port 4444 (dunkelgrüner Pfeil). Der GDB Client kommuniziert mit dem GDB Server mit dem GDB Protokoll über den TCP/IP Port 3333 (hellgrüner Pfeil). OpenOCD verwendet dann den *WinUSB* Treiber um mit dem FT2232 Chip zu kommunizieren. Der FT2232 verwendet den selben JTAG Bus wie das BDI 3000 als Verbindung mit dem Zynq.

Das *Zybo* beinhaltet neben dem FT2232 auch noch diverse I/O Peripherie die in einer *deep*-Applikation genutzt werden kann. Der FT2232 Chip übernimmt zwei verschiedene Funktionen. Zum einen wird er als USB zu UART Brücke verwendet, damit man mit dem Windows PC einfach eine serielle Verbindung mit dem Prozessor aufbauen kann. Zusätzlich fungiert er ebenfalls als Brücke zum JTAG Bus. Das bedeutet, er erhält Befehle von OpenOCD über USB und übersetzt diese elektrisch und auch logisch für das JTAG Interface.

3.3 Debugger Toolchains

3.3.1 Abatron-Toolchain

Die Abatron-Toolchain benötigt weder OpenOCD noch den FT2232, dafür aber das teure BDI 3000. Diese ”klassische” Toolchain wird für die Entwicklung von Deep Applikationen für den PowerPC verwendet. In dieser Arbeit wird sie aber nicht direkt verwendet.

3.3.2 CLI-OpenOCD-Toolchain

Das teure BDI wird für diese Toolchain nicht mehr benötigt. Da das CLI[1] von OpenOCD ist aber sehr ähnlich wie das CLI des BDI. Eine Portierung ist somit relativ einfach. Die CLI-OpenOCD-Toolchain lehnt sich deshalb sehr stark an die bestehende Abatron Toolchain an.

Mit dieser Toolchain ist *Source Code Debugging* aber nicht möglich. Das bedeutet, es ist nicht möglich im Source Code Breakpoints zu setzten, oder durch einzelne Zeilen im Source Code zu steppen wie man es von Debuggern wie dem GDB gewohnt ist. Bestehende Möglichkeiten aus der alten Abatron-Toolchain wie *Target Commands* bleiben aber erhalten.

[1] Command Line Interface

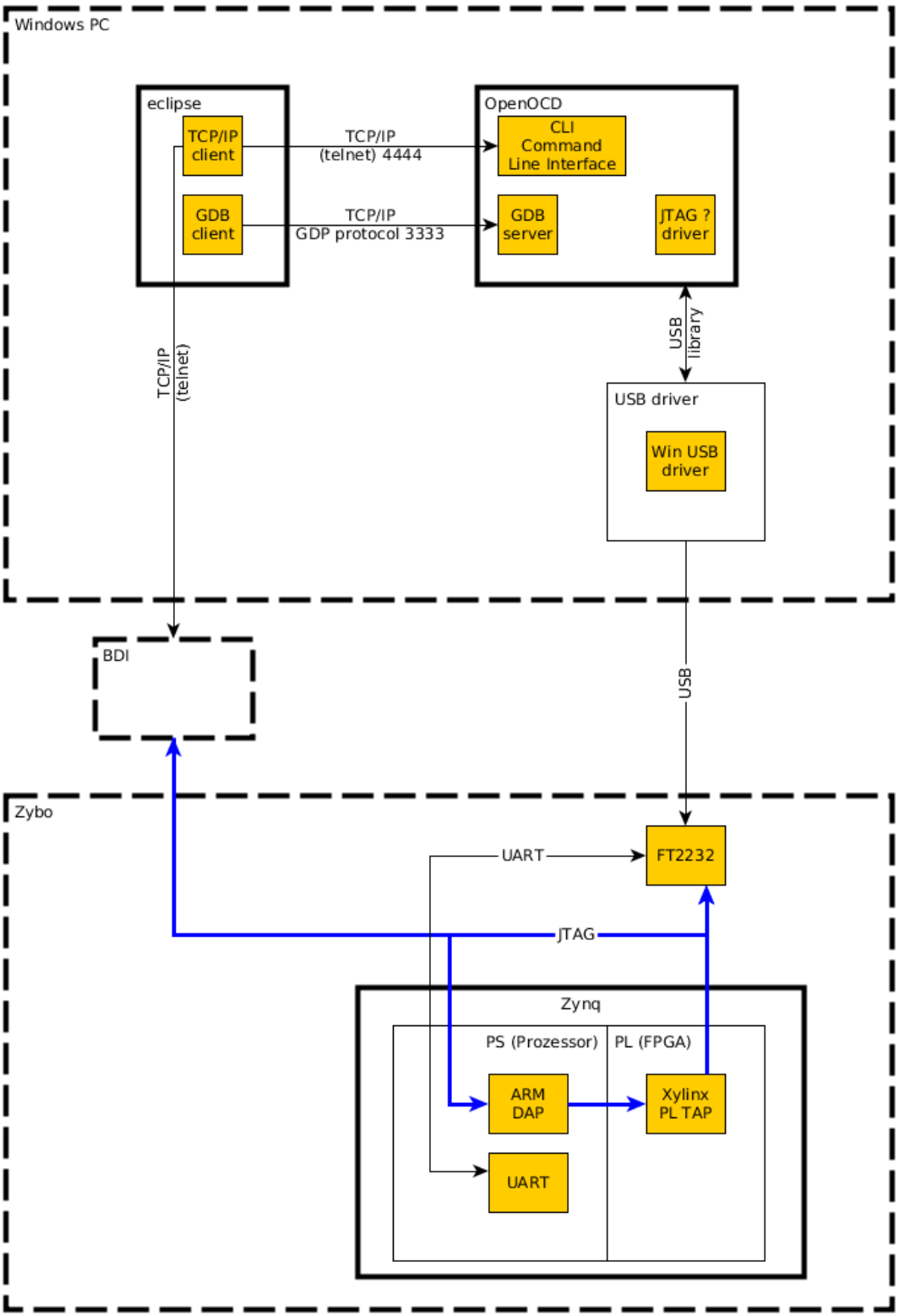

Abbildung 3.1: Systemübersicht Debugger Toolchain

3.3.3 GDB-OpenOCD-Toolchain

In der GDB-OpenOCD-Toolchain wird, wie bei der obigen Toolchain, ebenfalls die OpenOCD Software und der FT2232 Chip verwendet. Es wird aber nicht mehr ein Interface bestehend auf der ”klassischen” Abatron Toolchain verwendet, sondern es wird direkt der bekannte GDB in Eclipse verwendet. Dadurch kann *Source Code Debugging* direkt in Eclipse eingesetzt werden.

4 OpenOCD

OpenOCD[1] bildet den Software-Teil eines Debuggers. Zusammen mit einem Hardware-Adapter bildet OpenOCD einen vollständigen Debugger und kann als Ersatz für einen teuren Debugger wie beispielsweise dem BDI 3000 von Abatron verwendet werden.

Der Adapter bildet dabei das elektrische Interface zum Prozessor und muss auch auf den Prozessor abgestimmt sein. Relevant sind dabei unter anderem der Transport Layer (JTAG/SWD), das elektrische Potential und natürlich auch der Physikalischer Stecker. In den vielen Fällen basieren solche Adapter, wenn sie zusammen mit OpenOCD verwendet werden, auf dem FT2232 Chip von FTDI. Solch ein generischer Adapter ist in der Abbildung 4.1 zu sehen.

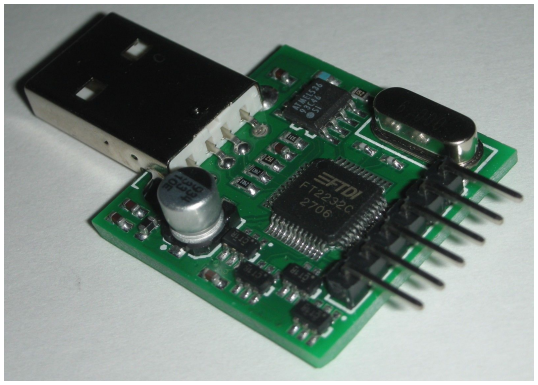

Abbildung 4.1: Generischer JTAG Adapter mit einem FTDI FT2232[2]

Bei Experimentierboards ist der FT2232 oft auch direkt auf das Board aufgelötet. So kann eine einfache USB-Verbindung genutzt werden, um den Prozessor zu debuggen. Beim Zybo wurde ebenfalls dieser Ansatz verfolgt. Aus diesem Grund reicht ein einfaches USB Kabel um den Prozessor des Zybos auf einer Hardwareebene debuggen zu können.

[1] http://openocd.org/about/

[2] https://www.ebay.com/itm/FPU1-FTDI-FT2232-USB-JTAG-XILINX-FPGA-CPLD-programmer-cable-/181635528314 Seite 5

4.1 Installation

Um OpenOCD nutzen zu können, muss auch der richtige USB-Treiber installiert sein. In den folgenden Kapitel wird erklärt, wie der Treiber und auch OpenOCD selber installiert werden kann.

4.1.1 Installation - OpenOCD

OpenOCD kann direkt aus dem Sourcecode kompiliert werden[3] oder es können vorkompillierte Binaries verwendet werden. Für diese Arbeit wurde das vorkompilierte Windows Binaries[4] für ARM Cores mit der Version 0.10.0 verwendet.

Das eigentliche Binary befindet sich im Ordner:

`/openocd-0.10.0/bin-x64/`

Das Open OCD User Manual[**?**] befindet sich im Ordner:

`/openocd-0.10.0/`

4.1.2 Installation - USB Driver WinUSB

Damit OpenOCD mit dem FT2232 Chip kommunizieren kann, werden die richtigen USB Treiber benötigt. Die Installation der Treiber ist am einfachsten mit den *USB Driver Tool*[5].

Das Zybo muss per USB mit dem PC verbunden sein, damit der Treiber installiert werden kann. Wenn der Jumper '*J15*' auf USB gesetzt ist, dann wird keine zusätzliche Stromversorgung für das Zybo benötigt.

Öffnet man das *USB Driver Tool* werden alle USB Devices aufgelistet. Das Device mit der *Vendor ID=0403, Device ID=6010* und *Interface 0* ist das JTAG Interface des FT2232. Mit einem Rechtsklick darauf kann man den *Install WinUSB* Treiber auswählen und installieren. Abbildung 4.2 zeigt die Liste

[3] http://sourceforge.net/p/openocd/code/
[4] http://www.freddiechopin.info/en/download/category/4-openocd?download=154%3Aopenocd-0.10.0
[5] http://visualgdb.com/UsbDriverTool/

mit allen USB Devices und das Kontextmenü für die Installation des richtigen Treibers. Um den Standardtreiber wieder zu installieren, kann einfach *"Restore default driver"* ausgewählt werden. Nachdem das Zybo einmal aus- und wieder einschaltet wird, ist der Treiber einsatzbereit.

Das Device mit der *Vendor ID=0403, Device ID=6010* und *Interface **1*** ist die UART Verbindung zum Prozessor. Dieser Treiber darf **nicht** ersetzt werden.

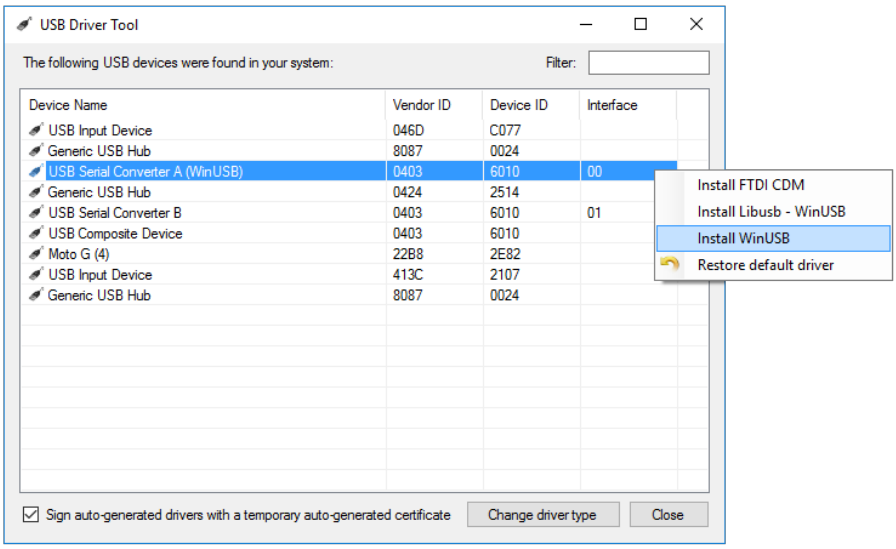

Abbildung 4.2: Installation des *WinUSB* Treibers mit dem *USB Driver Tool*

4.2 OpenOCD CLI - Command Line Interface

Das CLI (*Command Line Interface*) ist eine einfache Methode um mit dem Debugger zu kommunizieren. Sobald OpenOCD gestartet wurde, kann über den Port 4444, z.B. mit *Putty*, auf dem *Localhost* eine Telnet-Verbindung aufgebaut werden. Der Befehl "`help`" listet alle zulässigen Befehle auf.

In den folgenden Kapitel wird folgende Notation verwendet, um einen CLI-Befehl zu beschreiben:

`(CLI: Befehl)`

4.3 OpenOCD Konfiguration - Einleitung

OpenOCD unterstützt eine Vielzahl von Adaptern und Targets (Prozessoren). Beim Start muss die Software für die verwendete Hardware konfiguriert werden. Die Konfiguration erfolgt mit Konfigurationss-

cripts (*.cfg) in der Scriptsprache *Jim-Tcl*[6]. *Jim-Tcl* ist eine abgespeckte Version von *Tcl*[7].

Normalerweise werden die Scripts in die drei Gruppen *interface, board* und *target* aufgeteilt. So kann einfach ein Script ausgewechselt werden, wenn man den gleichen Adapter aber einen anderen Prozessor verwenden will. Im Pfad `openocd-0.10.0/scripts` befinden sich eine Sammlung von Konfigurationsscripts für Standardhardware.

Mit folgendem Befehl kann OpenOCD mit der passenden Konfiguration für das Zybo gestartet werden:

```
openocd -f zybo-ftdi.cfg -f zybo.cfg
```

4.4 OpenOCD Konfiguration - Interface

Die Interface Konfiguration beschreibt hauptsächlich den verwendeten Adapter. Da beim Zybo kein Adapter verwendet wird, sondern der aufgelötete FT2232, wird mit diesem Script der FTDI Chip und dessen Anbindung an den Zynq konfiguriert.

Da ein FTDI-Chip als Interface verwendet wird, sollte ein passender Script unter *openocd-0.10.0/scripts/interface/ftdi/* zu finden sein. Keiner der Scripts passt von Namen her auf *Zybo* oder *FT2232*. Eine Google Suche nach einem passenden Script war erfolgreicher. Ein Github User mit dem Namen *emard* hat folgenden Script in einem von seinen Repositories[8] gespeichert:

zybo-ftdi.ocd:

```
1  #
2  # ZYBO ft2232hq usbserial jtag
3  #
4
5  interface ftdi
6  ftdi_device_desc "Digilent Adept USB Device"
7  ftdi_vid_pid 0x0403 0x6010
8
9  ftdi_layout_init 0x3088 0x1f8b
10 #ftdi_layout_signal nTRST -data 0x1000 -oe 0x1000
```

[6] http://jim.tcl.tk/index.html/doc/www/www/index.html
[7] http://www.tcl.tk
[8] https://github.com/f32c/f32c/blob/master/rtl/proj/xilinx/zybo/xram_bram_hdmi_ise/zybo.ocd

```
11 # 0x2000 is reset
12 ftdi_layout_signal nSRST -data 0x3000 -oe 0x1000
13 # green MIO7 LED
14 ftdi_layout_signal LED -data 0x0010
15 #ftdi_layout_signal LED -data 0x1000
16
17 reset_config srst_pulls_trst
```

Zeile 5 bis 7 konfigurieren das Interface als ein standard-FTDI Interface. Von OpenOCD werden neben dem FT2232 auch noch andere Chips unterstützt. Zeile 7 definiert die *Vendor* und *Device-ID* des USB Devices.

4.4.1 Resetverhalten

Liest man aus einer unerlaubten Speicheradresse (`CLI: mdw 0x40000000`), dann hängt sich die Debug-Peripherie des Zynq auf. Nach so einem unerlaubten Speicherzugriff können auch keine erlaubten Speicherstellen mehr gelesen werden. Beim Versuch erscheint die Fehlermeldung:
`Timeout waiting for cortex_a_exec_optcode`.
Wahrscheinlich ist die *CoreSight* Debug-Peripherie abgestürzt oder in einem undefinierten Zustand. Aus diesem Grund bekommt OpenOCD keine Antwort vom Zynq, wenn versucht wird, eine Speicheradresse zu lesen. Mit einem manuellen Powercycle vom Zybo kann die Hardware wieder zurückgesetzt werden.

Im Supportbereich der Xilinx Homepage[9] ist eine Mögliche Erklärung für dieses Verhalten zu finden. In diesem Artikel wird beschrieben, dass die Fehlermeldung "*Invalid address - it can hang PS interconnect*" erscheint, wenn mit dem XSDB (*Xilinx System Debugger*) auf bestimmte Adressbereiche zugegriffen wird. Die Vermutung liegt nahe, dass der XSDB merkt, wenn auf eine *"Invalid address"* zugegriffen werden soll. Dieser Befehl wird abgefangen und stattdessen wird die Fehlermeldung angezeigt, so dass der *"PS interconnect"*, also der Bus innerhalb des Zynq, nicht abstürzen kann. OpenOCD fängt so einen invaliden Zugriff nicht ab, was dann zum Absturz des *"PS interconnect"* führt. Da auch

[9] https://www.xilinx.com/support/answers/63871.html

die Peripherie für den Debugger im Zynq von diesem *Interconnect* abhängig ist, stürzt auch die Debug-Peripherie ab, sobald auf einen ungültigen Adressbereich zugegriffen wird.

Mit OpenOCD ist es grundsätzlich möglich, einen Reset automatisch durchzuführen. Dabei wird zwischen einen SRST (*System Reset*) und dem TRST (*TAP Reset*) unterschieden. Der SRST führt dabei einen Powercycle vom ganzen System durch, der TRST setzt mit einem JTAG-Befehl nur den TAP (*Test Access Port*) zurück

Beim obigen Script ist aber das Resetverhalten nicht sauber definiert. Mit dem Befehl ''`CLI: reset halt`'' sollte der FT2232 einen Reset des ganzen Zynq durchführen. Der Befehl führt aber zur Fehlermeldung:

```
...
zynq.cpu0: how to reset?
...
```

Im OpenOCD User Manual[**?**] in *"Kapitel 9: Reset Configuration"* ist beschrieben, wie das Resetverhalten konfiguriert werden kann. Mit dem Script-Befehl ''`reset_config srst_only`'' wird der TAP Reset ignoriert. Da jetzt nur noch der SRST und nicht mehr der TRST verwendet wird, kann das Problem auf den SRST begrenzt werden.

Wenn OpenOCD mit der neuen Konfiguration neu gestartet wird, dann scheint der Befehl ''`CLI: reset halt`'' zu funktionieren. Greift man vorher aber wieder auf eine ungültige Speicherstelle zu, dann erscheint beim Reset die Fehlermeldung:

```
...
Timeout waiting for dpm prepare
...
```

Der erneute Timeout legt die Vermutung nahe, dass der Zynq nicht ordentlich zurück gesetzt wurde.

Zeile 12 ''`ftdi_layout_signal nSRST -data 0x3000 -oe 0x1000`'' konfiguriert die I/O Pins des FT2232 welche für den System Reset verwendet werden. Im elektrischen Schema des Zybos (siehe Anhang **??**) könnte man überprüfen, welche I/Os des FT2232 effektiv für den Reset verwendet werden. Die

Seite mit dem Schema für den FT2232, Seite 7, ist aber als einzige Seite im Schema nicht veröffentlicht worden. Die korrekten I/O Pins lassen sich also nicht mit dem Schema ermitteln.

Im OpenOCD User Manual[**?**] wird der für ''`ftdi_layout_signal nSRST` genauer beschrieben. Der Switch *-data 0x3000* definiert alle relevanten Pins für den SRST und *-oe 0x1000* konfiguriert alle Ausgänge. In einem Versuch wurden diverse Kombinationen für die beiden Switches ausprobiert. Keine Kombination mit nur einem Pin (z.B. *-data 0x2000* mit *-oe 0x2000*) hat funktioniert. Es hat sich herausgestellt, dass die Kombination *-data 0x3000* mit *-oe 0x3000* tatsächlich einen System Reset ermöglicht.

Weil der Debugger direkt nach dem SRST versuch mit dem Zynq zu kommunizieren, tritt folgende Fehlermeldung auf:

```
...
Invalid ACK (7) in DAP response
JTAG-DP STICKY ERROR
...
```

Mit dem Kommando ''`adapter_nsrst_delay 40`'' wartet der Debugger nach dem SRST zusätzliche 40 Millisekunden. Diese Wartezeit genügt, damit die Debug Peripherie wieder betriebsbereit ist, wenn der Debugger zu kommunizieren versucht.

4.5 OpenOCD Konfiguration - Board

Da beim Zybo der Adapter direkt auf dem Board ist, ist die Bordkonfiguration bereits im Konfigurationsscript für das Interface enthalten.

4.6 OpenOCD Konfiguration - Target

Für das Target, in diesem Fall der Zynq 7000 SOC, ist bereits ein Script unter *openocd-0.10.0/scripts/target/zynq_7000.cfg* enthalten. In diesem Script werden nicht nur beide Kerne des Prozessors definiert,

sondern auch ein TAP für das FPGA. Es ist also auch möglich, den FPGA mit dieser Toolchain zu laden.

5 Eidesstattliche Erklärung

Der unterzeichnende Autor dieser Arbeit erklärt hiermit, dass er die Arbeit selbst erstellt hat, dass die Literaturangaben vollständig sind und der tatsächlich verwendeten Literatur entsprechen.

St. Gallen, 10. August 2018

Marcel Gehrig

Anhang