Inhaltsverzeichnis

1 STUFF TO SORT

1.1 Zybo

1.2 OpenOCD

1.3 Was ist ein Debugger

2 Einleitung

2.1 Stand der Technik

Das Projekt *deep* [1] ist eine Cross Development Plattform, die es erlaubt, ein Java Programm direkt auf einem Prozessor auszuführen. Das *deep* Projekt ermöglicht es 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. Auch werden keine *Breakpoints* oder *Source Code Navigation* unterstützt, so wie man es aus bekannten Debuggern wie dem *gdb*[1] kennt.

2.2 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 ARM-Prozessoren 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 kann wahlweise sehr günstige oder auch sehr leistungsstarke Prozessoren verwendet werden.

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-Applikation sehr.

2.3 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.
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.

[1] https://www.gnu.org/software/gdb/

3 Auswahl der Hardware

3.1 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 zu finden, 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.

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

Um die richtige Hardware im riesigen Angebotsdschungel zu finden, sind Soll- und Muss-Kriterien ermittelt worden.

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.
  - 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 Modul oder direkt als Prozessor in eigenem Package.
  - Noch lange erhältlich.
- Prozessorebene
  - ARMv7: Der Prozessor muss auf der ARMv7 ISA[3] basieren.
  - ARM Instruktionen: Der Prozessor muss ARM Instruktionen unterstützen. *Thumb* Instruktionen sind nicht ausreichend.
  - FPU: Für Gleitzahlenarithmetik.
  - Netzwerkschnitstelle: RJ-45 inklusive MAC[4] und *Magnetics*.
  - USB: USB Schnittstelle als Host und als Slave.
  - Flash: Mehr als 50kByte Flash.
  - RAM: Mehr als 100kByte RAM.

Soll-Kriterien

[1] Elektronischer/Anhang/ARM-media-fact-sheet-2016.pdf
[2] http://www.abatron.ch/fileadmin/user_upload/news/BDI3000-Brochure.pdf
[3] Instruction Set Architecture
[4] Media Access Control

- 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.

3.3 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 bei der Produktion des Bestückten PCBs überprüft werden, ob alle Lötstellen Kontakt herstellen und 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 laufen 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**
- **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 installierte Software auf dem Prozessor benötigt, kann er auch gut verwendet werden, um die grundlegendsten Funktionen vom *deep* Laufzeit System zu entwickeln.

3.4 Übersicht über die ARM Mikroarchitekturen

3.4.1 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. Die leistungsstärksten ARM Cortex Prozessoren.

Tabelle 3.1: 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

3.4.2 Cortex-R

Cortex-R werden entwickelt für Echtzeitanwendungen und Sicherheitskritische Applikationen wie Festplattenkontrolle und medizinische Geräte. Sie sind normalerweise nicht mit einer MMU 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.

3.4.3 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 nicht die standard-ARM Instruktionen.

3.4.4 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.

3.4.5 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.

3.5 Anbindung des FPGAs

3.5.1 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.

Tabelle 3.2: Übersicht Bauformen

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

3.5.2 FPGA als Zusatzplatine zum Prozessorboard - Bauweise "Modular"

Das *FPGA Development Board CAPE for the BEAGLEBONE*[5] 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[6] Interface (bis zu 70 MB/s) zwischen Prozessor und FPGA.

Nachteile:

- Verwendet ein modifiziertes Linux-Image, das LOGI-Image.
- Der eMMC[7] 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"

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

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

3.5.5 ARM als Softcore in FPGA - Bauweise "FPGA"

3.5.5.1 STM23

STM

[5] https://www.element14.com/community/docs/DOC-69215/l/fpga-development-board-cape-for-the-beaglebone
[6] General-Purpose Memory Controller
[7] Embedded Multi Media Card

4 System

4.1 Einleitung

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

4.2 Schematische Übersicht

In der Abbildung 4.1 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 *Command Line Interface* (CLI) 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.

4.3 Debugger Toolchains

4.3.1 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.

4.3.2 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

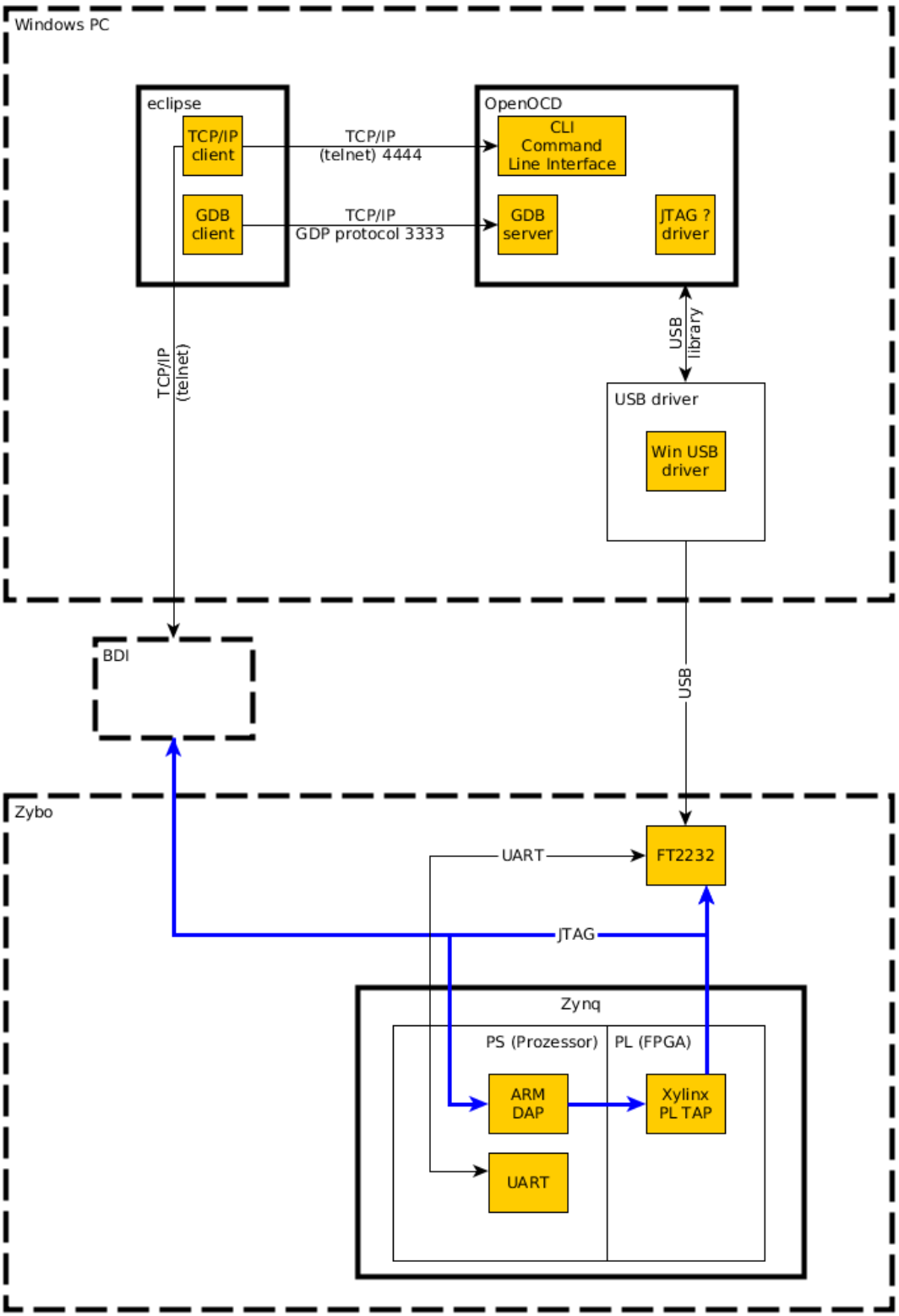

Abbildung 4.1: Systemübersicht Debugger Toolchain

4.3.3 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.

5 Auswahl der Hardware

5.1 Einleitung

6 Debugger

Es gibt diverse Debugger auf dem Markt. In dieser Arbeit beschränke ich mich aber auf den GNU-Debugger (*gdb*). Der *gdb* steht unter der PGL Lizenz und ist somit Open Source. Bei den meisten Linux Distributionen wird der *gdb* direkt mitgeliefert und kann sofort verwendet werden.

6.1 Anwendungsbeispiel gdb auf einem Linux-System

Mit diesem Beispiel will ich zum einen ein kurzes Tutorial bieten um den Umgang mit dem *gdb* zu lernen. Zum anderen will ich damit aber auch die verschiedenen Funktionen vom Debugger zeigen, damit die beschriebenen Probleme später im Kapitel besser in einen Kontext gestellt werden können.

Für dieses Tutorial verwende ich ein Linux Mint 18.1 (basierend auf einem Ubuntu 16.01). Solange *gdb* installiert ist, ist das verwendete Betriebssystem aber nicht relevant.

6.1.1 Grundlegende Funktionsweise

Auf Linux verwendet der *gdb* den System Call *ptrace* (Kurzform für "process trace"). Dieser System Call erlaubt dem *gdb* einen anderen Prozess zu inspizieren und zu manipulieren. Im Hardwaredebugger, den wir später bearbeiten, verwenden wir stattdessen JTAG in Verbindung mit der Debugginghardware im Prozessor.

6.1.2 Vorbereitung

Für dieses Tutorial verwenden wir folgendes Beispielprogramm:

```
1  #include <iostream>
2  using namespace std;
3
4  int divint(int, int);
5  int main()
6  {
7      int x = 5, y = 2;
8      cout << divint(x, y);
9
10     x =3; y = 0;
11     cout << divint(x, y);
12
13     return 0;
14 }
15
16 int divint(int a, int b)
17 {
18     return a / b;
19 }
```

Diese Applikation können wir jetzt Kompilieren und mit *gdb* starten:

```
# g++ gdbTest.cpp -o gdbTest
# gdb gdbTest
# run
```
// startet die Applikation im *gdb*

```
(gdb) run
Starting program: /home/mgehrig2/projects/gdbTest/gdbTest
```

Program received signal SIGFPE, Arithmetic exception.
0x00000000004007b5 in divint(int, int) ()

Obwohl die Applikation nicht mit Debug-Symbolen kompiliert wurde, wird nicht nur die Adresse des Ursprungs der Floating Point Exception angezeigt, sondern auch der Name der Methode.

6.2 Unterschied zwischen einem Software- und Hardwaredebugger

Auf einem Linux-System ist es sehr einfach ein Debugger einzusetzen.

6.3 Funktionen eines Debuggers

Ein Debugger kann verschiedene Funktionen besitzen. Die grundlegenden Funktionen sind sehr einfach und brauchen keine grosse Ïntelligenz"vom Debugger selber.

Erweiterte Funktionen erwarten vom

6.4 Erstellen einer Dummy-Applikation mit Debug-Informationen

6.4.1 Vorgehen

Das Ziel von diesem Kapitel ist es, eine Deep-Applikation zu erzeugen, die mit *gdb* und *OpenOCD*

7 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

Quellenverzeichnis

[1] Urs Graf: *deep - a Cross Development Platform for Java*, Juni 2018, http://www.deepjava.org/start

Anhang