

# Laborpraktikum: Software Debugging in eingebetteten Echtzeitsystemen

Skript

Laurenz Borchers, Kai-Björn Gemlau

16. April 2020



# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Lehrziel . . . . .	1
1.3. Anwendungsfall . . . . .	2
<b>2. Hardware und Aufbau</b>	<b>3</b>
2.1. Hardware . . . . .	3
<b>3. Grundlagenwissen</b>	<b>5</b>
3.1. Echtzeitsysteme . . . . .	5
3.2. Debugging . . . . .	5
<b>4. Coding Guidelines</b>	<b>8</b>
4.1. Regeln . . . . .	8
<b>5. Aufgabe 1</b>	<b>11</b>
5.1. Wissen . . . . .	11
5.2. Pre-Kolloquium . . . . .	13
5.3. Aufgabenstellung . . . . .	13
<b>6. Aufgabe 2</b>	<b>18</b>
6.1. Wissen . . . . .	18
6.2. Aufgabenstellung . . . . .	19
6.3. Post-Kolloquium . . . . .	21
<b>7. Aufgabe 3</b>	<b>23</b>
7.1. Wissen . . . . .	23
7.2. Pre-Kolloquium . . . . .	30
7.3. Aufgabe . . . . .	31
7.4. Post-Kolloquium . . . . .	33
<b>8. Aufgabe 4</b>	<b>34</b>
8.1. Wissen . . . . .	34
8.2. Pre-Kolloquium . . . . .	35
8.3. Aufgabe . . . . .	35
8.4. Post-Kolloquium . . . . .	38

<b>9. Aufgabe 5</b>	<b>40</b>
9.1. Wissen . . . . .	40
9.2. Aufgabe . . . . .	40
<b>10. Aufgabe 6</b>	<b>43</b>
10.1. Tracing . . . . .	43
10.2. Aufgabenstellung . . . . .	47
<b>11. Aufgabe 7</b>	<b>49</b>
11.1. Aufgabenteil 1 . . . . .	49
11.2. Aufgabenteil 2 . . . . .	51
<b>12. Aufgabe 8</b>	<b>53</b>
12.1. Wissen . . . . .	53
12.2. Aufgabenstellung . . . . .	53
<b>Akronyme</b>	<b>54</b>
<b>A. Anhang</b>	<b>55</b>
<b>Literaturverzeichnis</b>	<b>59</b>

# 1 Einleitung

## 1.1. Motivation

Im Programmieralltag wird man immer wieder damit konfrontiert, dass Software nicht einwandfrei funktioniert. Neben der Tatsache, dass es bei größeren Projekten schnell unmöglich wird den Fehler nur durch Analyse des Quellcodes zu finden, kommt es vor allem in eingebetteten Systemen vor, dass nicht immer Fehler in der Programmiersyntax oder -Logik vorliegen. Dabei geht es um Fehler, die nicht beim Kompilieren oder Linken des Programms auftreten, sondern erst zur Laufzeit der Anwendung. Um ein Programm zur Laufzeit zu debuggen gibt es mehrere Möglichkeiten. Dieses Praktikum wird Ihnen die klassischen Debug-Varianten darlegen und an Beispielen nachvollziehen lassen um diese Fehler effizient zu finden und zu korrigieren.

Dazu werden zunächst die Grundlagen des Programmbaus vom C-Code zum Maschinencode und zur Makefile behandelt, um später auftretende Fehler richtig einordnen und beheben zu können. Außerdem wird ein Grundverständnis von Betriebssystemen für eingebettete Systeme vermittelt.

Im nächsten Teil des Praktikums werden Ihnen Grundlagen des Software Debuggings von eingebetteten Systemen dargelegt und praktisch auf die entsprechenden Probleme angewandt. Dabei werden Sie lernen, welche Vor- und Nachteile die einzelnen Debugmethoden haben und wann es sinnvoll ist, welche Debugmethode zu verwenden. Die Arten des Debuggings, die in diesem Praktikum behandelt werden beinhalten Printf Debugging, Debugging via Programmablauf-Counter/Single-Stepping, die Nutzung von Break-/Trace- und Watchpoints, die Nutzung direkten Speicherzugriffs zur Laufzeit des Programms und Möglichkeiten und Anwendung des Tracing. Die Aufgaben fördern das Verständnis zur Arbeitsweise eines Betriebssystems vor allem in Bezug auf Tasks und deren Zustände, Scheduling und das Wissen um das Zeitverhalten in Echtzeitsystemen. Im Laufe der Veranstaltung wird auf Inter-Core Kommunikation in Multicore-Systemen eingegangen.

## 1.2. Lehrziel

Die Studierenden kennen am Ende des Praktikums die klassischen Varianten des Software Debuggings von eingebetteten Systemen. Sie können mit Software Debugging Verfahren wie zum Beispiel JTAG Debugging umgehen, kennen sich mit der Lauterbach Debugumgebung aus und wissen, welche Möglichkeiten sowie Vor- und Nachteile die jeweiligen

Debugmethoden mit sich bringen. Sie sind in der Lage Probleme zu beurteilen und die am besten geeigneten Methoden des Debuggings auf diese anzuwenden.

### 1.3. Anwendungsfall

Das Praktikum vermittelt seinen Lehrinhalt an einem Anwendungsfall. Es soll die Software für einen sich selbst balancierenden zweirädrigen Roboter entwickelt werden, der nach dem Vorbild eines Segways funktioniert. Der Roboter soll selbstständig sein Gleichgewicht halten und seine Position über eine Eingabe durch Fahrbewegung verändern können. Die dafür benötigte Hardware ist bereits vorhanden und wird den Studierenden zur Verfügung gestellt. Dazu sollen zu den Praktikumsterminen einzelne Komponenten entwickelt, beziehungsweise zum Teil vorgegebene Software mit Lauterbach-Debuggern debuggt werden. Die Roboterhardware wird über ein Zynq-7000 Board angesteuert. Die Teilnehmer müssen eine inertielle Messeinheit auswerten und die ausgewerteten Daten in Echtzeit verarbeiten. Die daraus berechnete benötigte Lagekorrektur des Roboters wird an die Motortreiber übermittelt. Hintergrund dieses Anwendungsfalls ist die gegebene Nähe zum Auslesen von Sensorik, Echtzeitverarbeitung von Daten, sowie Parallelen zur Automobilindustrie im Bereich der Multiprozessor-Datenverarbeitung im späteren Verlauf des Praktikums.

## 2 Hardware und Aufbau

### 2.1. Hardware

Der Aufbau und die Zusammenhänge der Hardwarekomponenten sind in Diagramm 2.1 dargestellt. Der Debugger Lauterbach PowerDebug Pro mit der Erweiterung PowerTrace-II ist ein leistungsfähiges Debugging-System. Besonderheiten sind unter anderem die eingebauten Features zum Debugging von Echtzeitbetriebssystemen und die umfangreichen Trace Funktionen. Genauere Informationen zum Lauterbach Debugger finden sich in der Quelle [lau].

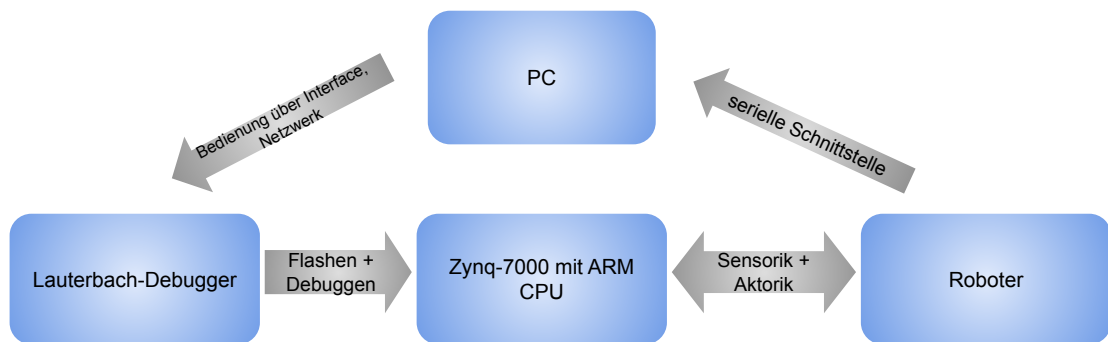


Abbildung 2.1.: Komponenten und Zusammenhänge des Versuchsaufbaus im Praktikum

Das Xilinx ZC706 Evaluation Kit ist auf das Entwickeln mit dem SoC Zynq 7000 optimiert. Das SoC ist mit einem Dual Core ARM Cortex A9 Prozessor ausgestattet. Außerdem findet sich ein FPGA in dem System, welcher eng mit dem Hauptprozessor zusammen arbeitet, um etwa die Ansteuerung der GPIOs flexibler zu gestalten (siehe Abbildung 2.2). Genauere Informationen zum Xilinx Zynq-7000 SoC ZC706 Evaluation Kit finden sich in der Quelle [zc7, Xilinx ZC706].

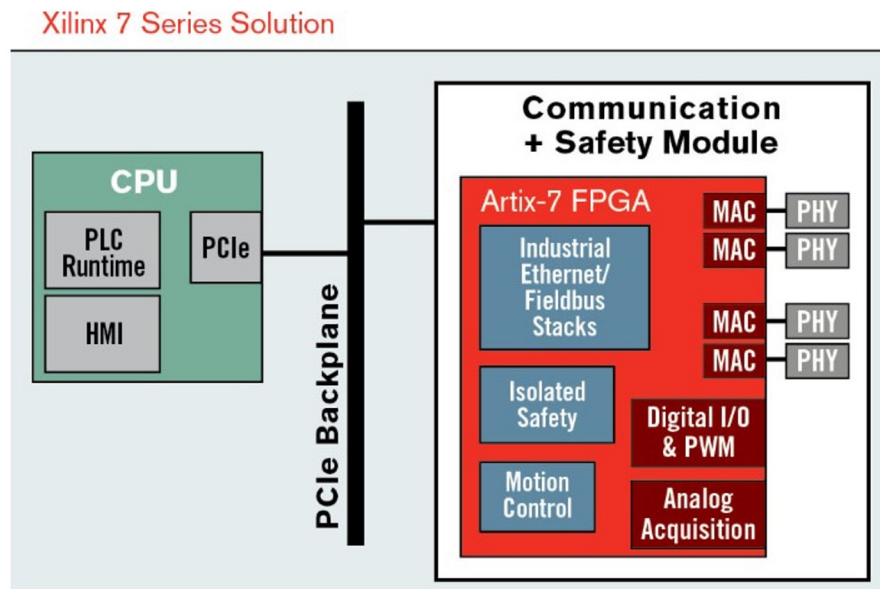


Abbildung 2.2.: Nutzung des Programmable Logic Controllers im Zynq7000 SoC



## 3 Grundlagenwissen

### 3.1. Echtzeitsysteme

Ein Echtzeitsystem wird nicht durch seine Schnelligkeit definiert, sondern durch das Einhalten von Zeitschranken und damit verbundene *Rechtzeitigkeit*. Das Ergebnis der Berechnung eines Echtzeitsystems ist nur dann brauchbar, wenn es rechtzeitig vorliegt. Echtzeitanforderungen können in *harte* und *weiche* Echtzeitanforderungen unterschieden werden. Entsprechend Ihrer Betitelung sind *harte* Anforderungen unter allen Umständen einzuhalten, da sonst das Ergebnis des Systems als nicht brauchbar gilt. Als Beispiel hierfür gilt die Auslösung des Airbags im Auto. *Weiche* Echtzeitanforderungen hingegen werden durch die bei ihrer Verletzung verursachten Kosten definiert. Je stärker die vorgegebene Zeitschranke verletzt wird, desto höher fallen die damit verbundenen Kosten in Form von Rechenzeit oder materiellem Verlust aus. Die Eigenschaft der *Gleichzeitigkeit* impliziert die korrekte Verarbeitung paralleler Arbeitsabläufe. Als Beispiel kann die Regelung der Lage des im Praktikum verwendeten Roboters gesehen werden, der Bewegungssensordaten und Motorsteuerung gleichzeitig bearbeiten muss. Diese Aufgaben werden in Form von Tasks umgesetzt. Wichtig ist dabei, dass ein höherpriorer Prozess einen niederprioreren Prozess verdrängen kann (*preemptiv*), und dass das System nicht überlastet ist. So wird *Vorhersagbarkeit* garantiert.

### 3.2. Debugging

Es gibt nicht nur eine bestimmte Methode des Debuggings, sondern mehrere Möglichkeiten. Sie unterscheiden sich in ihrer Leistungsfähigkeit, dem Anwendungsgebiet, den Kosten, den Anforderungen an die Hardware und auch in der Pinanzahl. Außerdem gibt es große Unterschiede in ihrem Einfluss auf das Zeitverhalten des Debuggee (zu debuggendes System). Die einfachste Möglichkeit einen Einblick in ein laufendes Programm zu bekommen, ist die Ausgabe von Variablen zur Laufzeit über eine serielle Schnittstelle. Das Printf Debugging ist simpel und reicht für einfache kleine Programme ohne Taskstruktur und ohne strikte Anforderungen an Echtzeitfähigkeit aus. Die nächst-mächtigere Methode des Debuggings stellt JTAG-Debugging dar. Es ermöglicht die Steuerung des Programmablaufs via Single-Stepping, also das kontrollierte Anhalten des Programms und das Setzen von Halte-, Verfolgungs- und Überwachungspunkten. Außerdem ermöglicht es den vollständigen Lese- und Schreibzugriff auf den Speicher des Programms. Mit diesen Werkzeugen lassen sich einfach Fehler finden und beheben. Problematisch wird es dann, wenn unvorhergesehene Fehler zur Laufzeit auftreten, deren Ursprung und Wirkung im Code entweder zeitlich oder programmierhierarchisch weit auseinander liegen. Die bis-

her bekannten Debugmethoden können ihre Möglichkeiten aufgrund für den Menschen schwierig erkennbarer Zusammenhänge nicht ausspielen, was die Fehlersuche sehr erschwert. Außerdem ist es auch hier nahezu unmöglich, das Zeitverhalten des Programms umfangreich zu analysieren. An dieser Stelle hilft es Trace-Tools als Debugmethode zu verwenden. Zusätzlich zur Steuerung der CPU wird nun der Programmablauf teilweise oder vollständig aufgezeichnet. Aus diesem kann dann wahlweise auf Assemblerebene bis hin zum Ablauf in der Hochsprache nachvollzogen werden in welcher Reihenfolge das Programm ablief. Diese Daten helfen zu rekonstruieren wo das Programm anders als gewünscht ablief und ermöglichen es den Stand des Programms vor dem Absturz zu betrachten und damit auch eine mögliche Fehlerquelle zurück zuschließen.

### 3.2.1. Debugging-Prozess

Um Programme zu debuggen, sollte zunächst definiert sein, was ein Fehler ist und wie an ihn herangegangen wird. Der Begriff Fehler ist im Deutschen mehrdeutig belegt und lässt sich mit Hilfe der englischen Begriffe differenzierter betrachten (siehe Abbildung 3.1). Ein *Error* ist ein nicht korrekt programmierter Code oder eine falsch implementierte Nutzeranforderung. Dieser *Error* im Programmcode kann einen *Fault* auslösen. Es kann sich zum Beispiel um einen Speicherüberlauf handeln. Der *Fault* muss nicht nach außen sichtbar sein. Das extern zu beobachtende Fehlverhalten des Systems wird *Failure* genannt. Ist ein *Failure* erkannt worden, muss die Stelle im Quellcode gefunden werden, die den *Fault* auslöst und den *Error* beschreibt. Die Ursache des *Fault* muss analysiert und korrigiert werden. Sollten mehrere Fehlverhalten gleichzeitig auftreten, deren Symptome sich überlagern, kann es vorkommen, dass die Betrachtung eines einzelnen nicht funktionierenden Programmteils alleine nicht zur Lösung des Problems führt. An dieser Stelle kann viel Zeit gespart werden, indem nicht sofort versucht wird herauszufinden was nicht richtig arbeitet, sondern was überhaupt funktioniert. Je konsequenter das Programm in modularisierter Form geschrieben wurde, desto einfacher und schneller geht dieser Schritt vonstatten. Es ist möglich Eingabedaten in ihrer Größe oder in ihrem Inhalt zu variieren, um ihren Einfluss auf Fehler zu untersuchen. Es sollten Hypothesen zur Fehlerursache aufgestellt und das Programm systematisch darauf getestet werden. Sofern es sich nicht um triviale Fehler handelt, ist es sinnvoll, das Wissen um die Ursache des *Fault* festzuhalten, um den Hergang des Problems sowie seine Lösung zu dokumentieren.

### 3.2.2. Grundlagen des Debuggens eingebetteter Systeme

Debugging in der Softwareentwicklung für Desktopanwendungen ist wesentlich zugänglicher und komfortabler als im Kontext der eingebetteten Systeme. So programmieren die meisten Anwendungsentwickler auf den Systemen, auf denen die von ihnen entwickelte Software später auch ausgeführt wird. Die Steuerung des Programmablaufs, das Betrachten von Variablen und andere Debugwerkzeuge sind ohne große Umwege nutzbar. Um eingebettete Systeme zu programmieren, kommt oft die Technik des Cross-Compilings und des Cross-Debuggings zum Einsatz. Es wird also nicht auf der Zielhardware kompiliert und debuggt, sondern auf einem Anwendungscomputer. Das zu programmie-

rende System hat eventuell nicht ausreichend leistungsfähige Hardware, um ein Anwenderbetriebssystem samt Entwicklungsumgebung auszuführen oder es ist schlicht nicht gewünscht das System zu beeinflussen. Auch die Architektur der Systeme unterscheidet sich für gewöhnlich. Diese Trennung von Entwicklungsplattform und Laufzeitplattform erschwert die Beobachtbarkeit und Steuerbarkeit eingebetteter Systeme und verkompliziert somit deren Debugging. Dennoch ist die Anwendung von Debugging notwendig, weil es unter Umständen zu Fehlern kommt, die erst während der Laufzeit des Systems auftreten, deren Entstehung aus dem Code nicht ersichtlich ist oder das Projekt zu umfangreich ist.

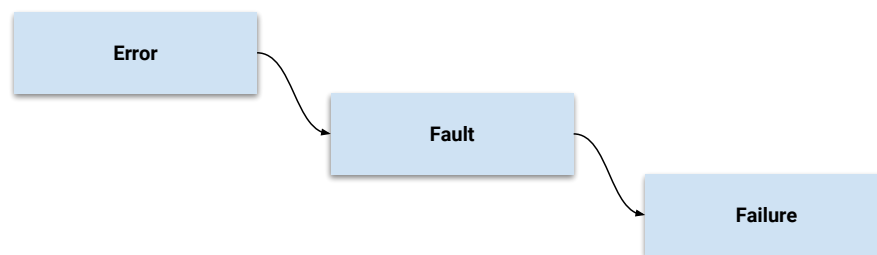


Abbildung 3.1.: Entwicklung eines Fehlers: Error->Fault->Failure

# 4 Coding Guidelines

## 4.1. Regeln

Im folgenden finden Sie einige Regeln wie der C-Code in diesem Praktikum formatiert sein muss. Sauber strukturierter Quellcode trägt entscheidend dazu bei, dass Fehler entweder von vorneherin vermieden werden oder beim Debugging leichter eingrenzbar sind.

### 4.1.1. Modul- und Funktionsnamen

Ein Modul (z.B. eine C-Datei mit Funktionen die als Gruppe eine Teilfunktion erfüllen) hat einen prägnanten Namen. Dieser wird sowohl im Dateinamen verwendet, wie auch als präfix für alle Funktionen und Variablen. Beispiel: Die Datei 'imu.c' soll alle Funktionen zum auslesen der IMU enthalten. Diese heißen dann entsprechend 'imu\_init()', 'imu\_readGyro()' usw. und nutzen globale Variablen wie 'imu\_config'

### 4.1.2. Interne Funktionen und Variablen

Als Grundsatz gilt: Alles was nicht für andere Module als Interface nötig ist wird mit 'static' und einem Unterstrich als Präfix versehen. Somit ist es nur innerhalb dieser einen C-Datei bekannt und es wird klar wie die Schnittstellen nach außen aussehen.

```
1 /** Dies ist eine interne variable */
2 static int32_t _internalTimerCount;
3 /** Dies ist eine interne Funktion */
4 static void _imu_handleError(int err);
```

Quellcode 4.1: Deklaration von static Variablen und Funktionen

### 4.1.3. Kommentare

Jede Funktion bekommt einen Kommentar im Doxygen-Stil, der den Inhalt der Funktion sowie ihre Parameter und Rückgabewerte beschreibt.

```
1 /*****
2  */
3  * This function initiates an interrupt-driven receive in master mode.
4  *
5  * It sets the transfer size register so the slave can send data to us.
6  * The rest of the work is managed by interrupt handler.
7  *
8  * @param InstancePtr is a pointer to the XlicPs instance.
9  * @param MsgPtr is the pointer to the receive buffer.
```

```

10 * @param ByteCount is the number of bytes to be received.
11 * @param SlaveAddr is the address of the slave we are receiving from.
12 *
13 * @return None.
14 *
15 * @note This receive routine is for interrupt-driven transfer only.
16 *
17 *****/
18 void XlicPs_MasterRecv(XlicPs *InstancePtr, u8 *MsgPtr, s32 ByteCount,u16 SlaveAddr){
19     ....
20 }

```

Quellcode 4.2: Beispielhaftes Doxygen Kommentar aus der Xilinx Library

#### 4.1.4. Aufbau der C-Datei

Jede C-Datei hat einen klaren Aufbau der sich in die Bereiche “Includes”, “Inputs”, “Outputs”, “interne Variablen”, “interne Konstanten”, “interne Funktions-Prototypen” und die eigentliche Implementierung aufteilt. So ist klar ersichtlich wo man eingreifen muss um z.B. konstante Parameter zu verändern.

```

1  /** Includes */
2  #include " .... h"
3
4  /** module input variables */
5  extern int16_t accData[3];
6
7  /** module output variables */
8  int32_t pidValue=0;
9
10 /** module internal variables */
11 static float _pid_errorSumAngle=0;
12
13 /** module internal constans */
14 const int16_t ACC_MIN[3] = {-10, -20, -30};
15
16 /** internal function prototypes */
17 static int16_t _pid_map(int16_t x, int16_t in_min, int16_t in_max, int16_t out_min, int16_t out_max);
18 ...
19 /** functions */
20 void pid_init(void){
21     ...
22 }
23
24 static int16_t _pid_map(int16_t x, int16_t in_min, int16_t in_max, int16_t out_min, int16_t out_max){
25     ...
26 }

```

---

Quellcode 4.3: Aufbau einer C-Datei

# 5 Aufgabe 1

Die in dem Skript stehenden Aufgaben dienen der Anweisung was programmiert werden soll. Zusätzlich zu den Programmieraufgaben halten Sie Ihre Ergebnisse in einem vorgefertigten .odt Dokument fest, wo sich Fragen zur Vorbereitung auf die Hauptaufgabe finden.

## 5.1. Wissen

### 5.1.1. Toolchain

Jedes in Hochsprache geschriebenes Programm muss in Maschinencode übersetzt werden. Die *Toolchain* ist eine Aneinanderreihung von Routinen zur Erstellung eines ausführbaren Programms aus dem Code einer Hochsprache wie zum Beispiel der Sprache C. Damit der Code unserer Hochsprache auf unserem Zielsystem ausgeführt werden kann, muss er zuerst in die dem Zielsystem entsprechende Maschinsprache übersetzt werden. Der *Compiler* unserer Toolchain übersetzt den von uns geschriebenen Code aus der Hochsprache in Assembler-Code. Dieser Code hängt bereits von dem verwendeten Ziel ab. Je nach Befehlssatz der Zielarchitektur (ARM, AVR, x86, ...) sieht der Assemblercode verschieden aus. Der vom Compiler erstellte Assembly-Code wird vom Assembler in Maschinencode umgewandelt. Als Basis für das weitere Verständnis zur Toolchain kann das Dokument *How a Compiler Works* [RS14, Kapitel 2, S. 2] genutzt werden. In dem Diagramm 5.1 (siehe auch in Ihrer Dokumentation) können zum Verständnis die Arbeitsschritte der Toolchain eingetragen werden. Nutzen Sie dafür das zur Verfügung gestellte .odt Dokument.

### 5.1.2. Makefile

Das Programm *Make* wird verwendet, um den Buildprozess zu automatisieren. Dafür liest Make die *Makefile* aus und gibt die entsprechenden Anweisungen an die Toolchain weiter (z.b. Compiler, Linker, ...). Dies ist besonders bei größeren Projekten hilfreich, da der Buildprozess aus vielen Einzelanweisungen bestehen kann. Außerdem können unterschiedliche Build-Konfigurationen und *Targets* benutzt werden.

Eine Erklärung von Makefiles bietet die Website [mak]. Schauen Sie sich den Inhalt an und verinnerlichen Sie den Sinn und die Arbeitsweise von Makefiles.

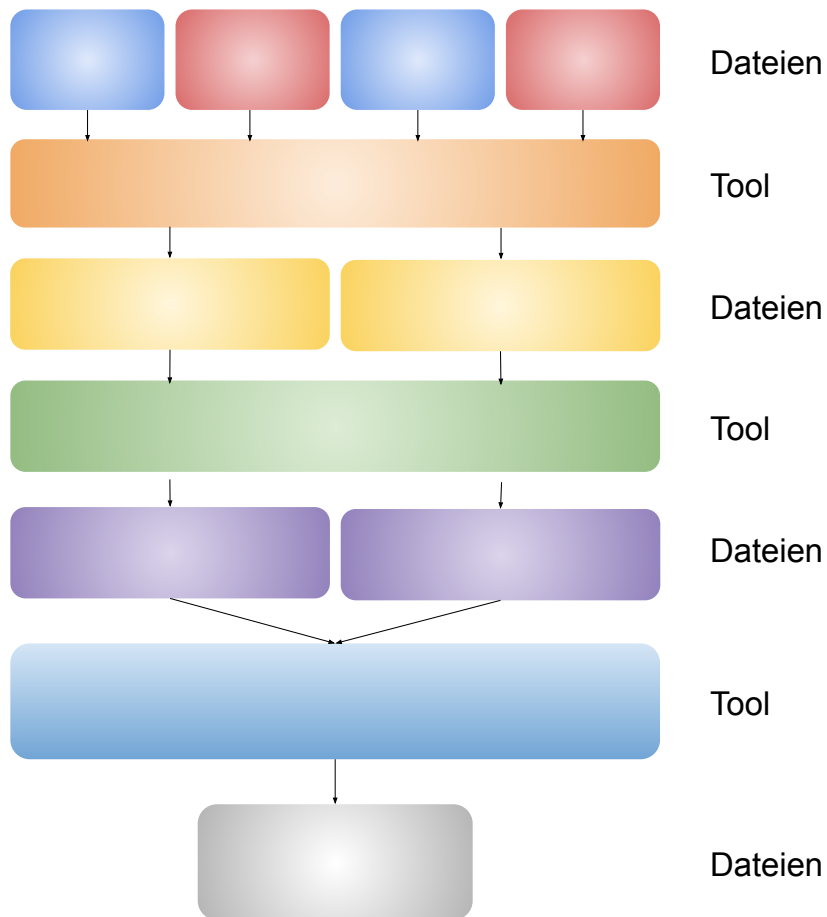


Abbildung 5.1.: Toolchain Diagramm zum Ausfüllen und zur Vorbereitung auf das Prekolloquium

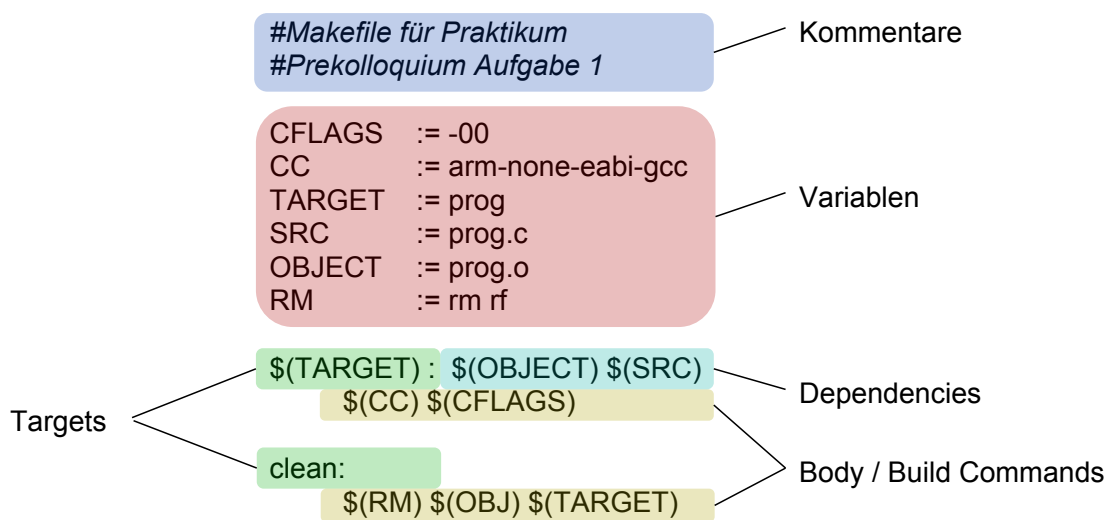


Abbildung 5.2.: Beispiel zum Aufbau einer Makefile



### 5.1.3. Printf-Debugging

#### Funktionsdefinition

Eine intuitive und einfache Methode des Debuggens ist die Ausgabe von Werten oder Nachrichten über eine serielle Schnittstelle. Zum Lesen solcher Nachrichten auf der Entwicklungsplattform ist es meist nötig, einen Seriell-zu-USB Adapter einzusetzen. Diese sind günstig und weit verbreitet. Als Software dient ein serieller Monitor.

#### Anwendung

Möchte man die Implementierung einer Berechnung überprüfen, so kann dies über einen Printf-Befehl getan werden, der das Ergebnis ausgibt. Auch eignet sich diese Debugging-Methode gut als Indikator, ob bestimmte Stellen im Programmcode erreicht werden. Besonders hilfreich ist diese Methode zum Überprüfen von Variablenwerten in Schleifendurchläufen. Auf die gleiche Art und Weise können falsche Übergabe- oder Rückgabeparameter erkannt werden.

#### Grenzen

Der Rahmen in dem Printf-Debugging zum Erfolg führt, ist stark begrenzt. Es bietet keinerlei Hilfe bei Problemen, die mit Speicherallozierung oder Interrupts zu tun haben. Printf nutzt die langsame serielle Schnittstelle und verändert das Zeitverhalten des Programmes stark. Das kann dazu führen, dass sich Fehler in einem auf Echtzeitfähigkeit ausgelegten System anders verhalten, wenn eine Printf-Anweisung in den Code eingefügt wurde. Es ist dadurch nicht möglich zeitkritische Anwendungen zu debuggen. Als Beispiel dafür gilt die Kommunikation über Bussysteme. Dazu kommt der Aufwand und die Dauer des wiederkehrenden Build-Prozesses, da nach dem Verschieben der Printf-Anweisung an eine andere Stelle im Programm, das Programm erneut kompiliert werden muss. Vor allem bei großen Projekten bedeutet dies lange Wartezeiten und ist nicht praktikabel.

## 5.2. Pre-Kolloquium

Für das Kolloquium sollte klar sein, wie die Toolchain funktioniert und welche Werkzeuge in den Programmbau involviert sind. Als Visualisierung können Sie das Diagramm zur Toolchain ausfüllen. Machen Sie sich mit den einzelnen Schritten vertraut. Die Regeln, nach denen das Programm gebaut wird, finden sich in dem Makefile. Ihnen sollte klar sein, welche grundlegenden Bestandteile sich in der Datei finden.

Sie sollten erklären können welche Teile im Makefile welche Funktion erfüllen und ggf. schon die Fehler im gegebenen Makefile gefunden haben.

## 5.3. Aufgabenstellung

### 5.3.1. Toolchain

Die erste Aufgabe formt den Einstieg in die Programmierung des Praktikumsboards. Es wird sich mit der Ordnerstruktur und der Entwicklungsumgebung vertraut gemacht und

die Struktur des Gits verstanden. Es soll das erste Programm gebaut und geflasht sowie der Umgang mit Makefiles geübt werden. Außerdem soll die Funktionsweise der Toolchain verinnerlicht werden.

Erstellen Sie zunächst einen Fork des Repositories von

```
1 https://git.ida.ing.tu-bs.de/IDA_Lehre/sdes_student
```

Auf diesem Fork können Sie dann arbeiten. Dazu clonen Sie zunächst Ihren Fork mit dem Link aus dem Webinterface angezeigt in Ihren lokalen Ordner.

In dem bereit gestellten git-repository finden sich alle für die Aufgabe benötigten Dateien. In dem Ordner `src/APP` befinden sich der Quellcode der Aufgaben des Praktikums. Der Ordner `src/APP/Aufgabe1/ps7/core0/` enthält den Quellcode der Aufgabe 1 für den die Architektur `ps7` und den Kern `0`. In ihm finden sich weitere Unterordner. In `Aufgabe1/ps7/core0/src` findet man alle `.c`-Dateien außer der `main.c`. Der Ordner `Aufgabe1/ps7/core0/cfg` enthält die Header-Dateien und der Ordner `linker` das Linkerscript.

Wichtig ist außerdem der `Aufgabe1/ps7/core0/build` Ordner. In den Dateien `config.mk`, `includes.mk` und `sources.mk` festgelegt, welche Pfade beim Build-Prozess überhaupt berücksichtigt und welche Compilerflags gesetzt werden. Neue Dateien in bisher nicht inkludierten Pfaden müssen in den entsprechenden Dateien eingetragen werden. In dem Ordner `out` finden sich geordnet nach den Aufgaben, Architektur und Core die entsprechenden Zielpfade für die aus dem Build-Prozess entstehenden Objekt-Dateien.

Der Ordner `Debug/ps7` im Oberverzeichnis enthält das Skript `"start_amp_session.sh"`, welches die Debug-Umgebung lädt. Zur Anwendung des Skripts wird ein Terminal in dem Ordner `ps7` geöffnet und der Befehl

```
1 ./start_amp_session.sh lauterbach"Lauterbachnummer"
```

ausgeführt. Die "Lauterbachnummer" muss angepasst werden und entspricht der Gruppennummer.

Nach dem Klonen des Gits ist die Entwicklungsumgebung einzurichten:

- Terminal in Verzeichnis der Wahl (aber nicht in dem Git Repositoryordner) öffnen
- `setup-lm eclipse cdt-oxygen` ausführen
- `setup-lm gcc gcc-arm-none-eabi-7-2018-q2` ausführen
- Eclipse mit `eclipse&` starten
- Neuen Workspaceordner erstellen und Ordner auswählen

- File->New->Makefile Project with existing Code
- sdes\_student als Projektnamen wählen
- unter Existing Code Location den soeben geklonten git-Ordner auswählen
- Toolchain = Cross GCC und Setup fertigstellen
- Links in der Leiste restore auswählen
- Rechtsklick auf das Projekt und Properties auswählen
- unter C/C++ Build als Build Command `make APP=Aufgabe1 ARCH=ps7 CORE=0` eingeben
- Unter C/C++ General -> Indexer project specific settings aktivieren
- Indexer aktivieren und alle anderen Optionen deaktivieren
- Use active build configuration auswählen
- Reindex project on change of active build configuration auswählen
- C/C++ General -> Code Analysis , projektspezifische Optionen auswählen und alles andere abwählen, wir verlassen uns nur auf die Fehlermeldungen des Compilers.
- Unter C/C++ General -> Preprocessor Includes -> Providers Tab alles außer CDT User Setting Entries und CDT GCC Build output parser abwählen
- Bei CDT GCC Build output parser das Compiler command pattern setzen:  
`(g?cc)|([gc]++)|(clang)|(arm-none-eabi-gcc)|(aarch64-elf-gcc)`
- Container to keep discovered entries => Project

Ihre Aufgabe ist es, das Programm zu kompilieren und in die Laufzeitumgebung zu laden. Dabei auftretende Fehler sind zu beheben und **der Arbeitsvorgang zu dokumentieren**. Zunächst ist es sinnvoll sich den Programmcode der Datei `main.c` anzusehen und zu verstehen. Um das Programm zu kompilieren gehen sie in den Repositoryordner, öffnen ein Terminal und führen den Befehl `make` aus. Es ist nötig dem Programm `make` mitzuteilen, für welche Aufgabe und Architektur das Programm kompiliert werden soll. Die Syntax des Aufrufs ist folgende:

```
1 make ARCH=ps7 APP=Aufgabe1 CORE=0
```

Falls Fehler auftreten, analysieren Sie diese und versuchen Sie ihre Gründe herauszufinden. Es lohnt sich die Ausgabe genauer anzusehen und mit dem Wissen über die Toolchain und Makefile zu verknüpfen. So kann der Ursprung des Fehlers schnell eingegrenzt werden. Nutzen sie die Ausgabe des Kompiliervorgangs und lösen Sie die auftretenden Fehler, um den Build-Prozess zu ermöglichen.

Tipp: Alle von der Toolchain benötigten Abhängigkeiten finden sich in dem Ordner `Aufgabe1/ps7/core0/build`. Suchen Sie auch in der Makefile nach Fehlern!

Nach dem erfolgreichen Kompilieren muss das Programm auf das Board geflasht werden. Dafür starten wir die Lauterbach Software "Trace 32" wie oben beschrieben. Für jede Aufgabe gibt es im Ordner `Debug/ps7/` einen entsprechenden Unterordner, welcher ein Lauterbach-Skript enthält. Diese kann über die Befehlszeile in Trace32 ausgeführt werden:

```
1 do Aufgabe1/zc706_onchip_trace.cmm
```

Generell arbeitet Trace32 vollständig skript-basiert und jedes GUI Kommando kann auch in einem Skript eingesetzt werden. Dies können Sie sich für spätere Aufgaben merken um wiederkehrende Befehle zu automatisieren.

Das zu Aufgabe 1 gehörige Skript sorgt dafür, dass das Board durch die Software geflasht wird, sich aber keine weiteren Fenster in der Software öffnen. Ziel dieser Aufgabe ist das Debuggen über printf. Nachdem Sie das den Befehl zum Flashen im richtigen Verzeichnis ausgeführt haben, sollte sich ein Fenster ähnlich der Abbildung 5.3 öffnen. Das Programm wird nun durch einen Klick auf "Go" gestartet. Sie können dieses Fenster nun ignorieren und in einer freien Konsole die Verbindung zur Ausgabe des Boards aufbauen:

```
1 telnet ida-ser2net 800X
```

Der Port hängt von Ihrer Gruppennummer und dem verwendeten Lauterbach ab. Ersetzen Sie das 'X' durch die Nummer ihres Lauterbach-Debuggers.

Bei Überprüfung der Arbeitsergebnisse sollten Sie auftretende Fehler dem Linker oder dem Compiler zuordnen können.

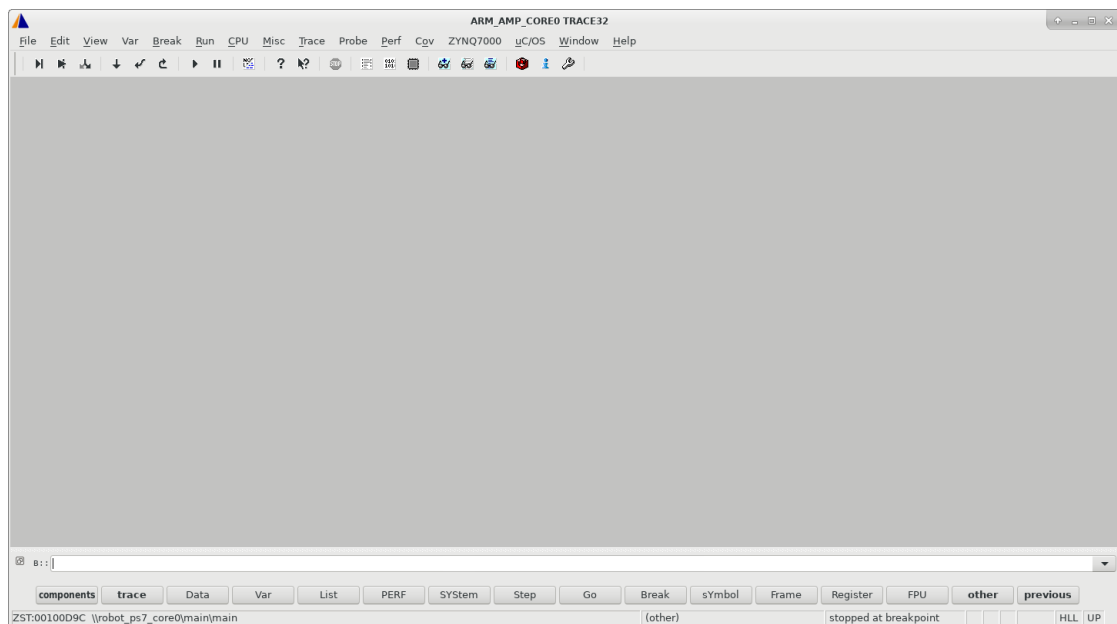


Abbildung 5.3.: Lauterbachumgebung zum Flashen des Boards in Aufgabe 1

# 6 Aufgabe 2

## 6.1. Wissen

### 6.1.1. Tasks

Ein Task ist für gewöhnlich eine Endlosschleife einer Funktion, die von dem Scheduler der CPU zugeteilt wird. Es können mehrere Tasks gleichzeitig laufen, die dann entsprechend ihrer Priorität CPU-Zeit zugeteilt bekommen. In  $\mu$ C/OS-II wird ein Task mit der Methode

```
1 INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)
```

erstellt. Der Task benötigt eine Priorität, die gleichzeitig auch seine Identifikation darstellt. Kleinere Zahlen bedeuten eine höhere Priorität. Mit dem Funktionsaufruf von

```
1 UCOSStartup (CPU_FNCT_PTR initial_func)
```

wird unter anderem der Scheduler und somit das Multitasking gestartet. Ein Task benötigt seinen eigenen Stack mit der entsprechenden Stack-Größe. Dieser sollte statisch alloziert werden.

Empfohlene Literatur:

- *$\mu$ C/OS-II Micrium Documentation* zum Thema Task Management [[mica](#),  $\mu$ C/OS-II Quick Reference, Task Management]
- *$\mu$ C/OS-II Micrium Documentation* zum Thema Task Control Blocks [[micb](#),  $\mu$ C/OS-II Quick Reference, Task Management]
- *$\mu$ C/OS-II Micrium Documentation* zum Thema Message Queues [[micc](#),  $\mu$ C/OS-II Quick Reference, Message Queues]
- *Micrium: Inter Process Communication via Message Queues* [[wik](#), Message Queues Quick Start Guide, S. 9]
- *Technical Reference Manual : Zynq-7000 ZC-706 im Repository Ordner*

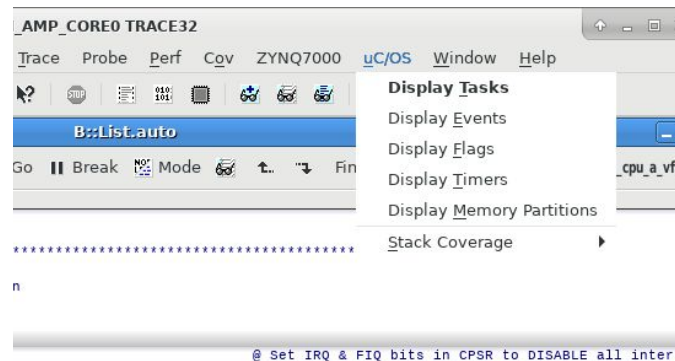


Abbildung 6.1.: Menü zum Inspizieren von Betriebssystemfunktionen

## 6.2. Aufgabenstellung

### 6.2.1. Teil 1

Das in der ersten Aufgabe programmierte Programm soll nun als Task ausgeführt werden. Zusätzlich sollen in einem weiteren Task Fibonaccizahlen berechnet und ausgegeben werden.

- Das Programm weist zahlreiche Fehler auf, die es zu debuggen gilt
- Eine sinnvolle Hilfe stellt die UCOS-2 Dokumentation zu dem Thema Task Management dar (siehe oben: "Empfohlene Literatur")
- Tipp: Die UART Schnittstelle ist fehlerhaft konfiguriert, die gewünschte Frequenz sollte auf 50000000 eingestellt sein, die erste der beiden UART Instanzen ausgewählt sein. (siehe TRM Zynq 7000)

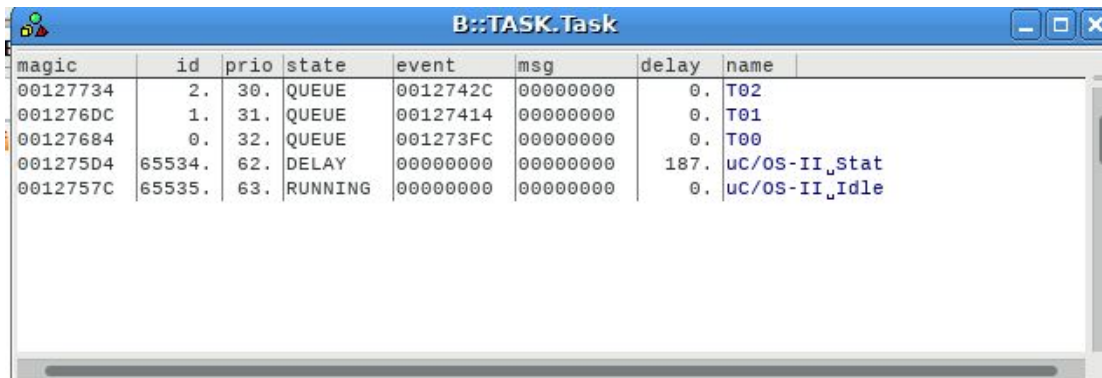
Dokumentieren Sie die gefundenen Fehler.

Die Lauterbach Umgebung bietet mehrere Möglichkeiten das Multitasking in UCOS zu überwachen (siehe Abbildung 6.1):

- Überwachung der Tasks und ihrer Prioritäten (Beispiel siehe Abbildung 6.2)
- Überwachung von Stackgrößen, auch die der Tasks (Beispiel siehe Abbildung 6.3)  
Diese Überwachung basiert auf der Überprüfung auf Nullen im Stack.

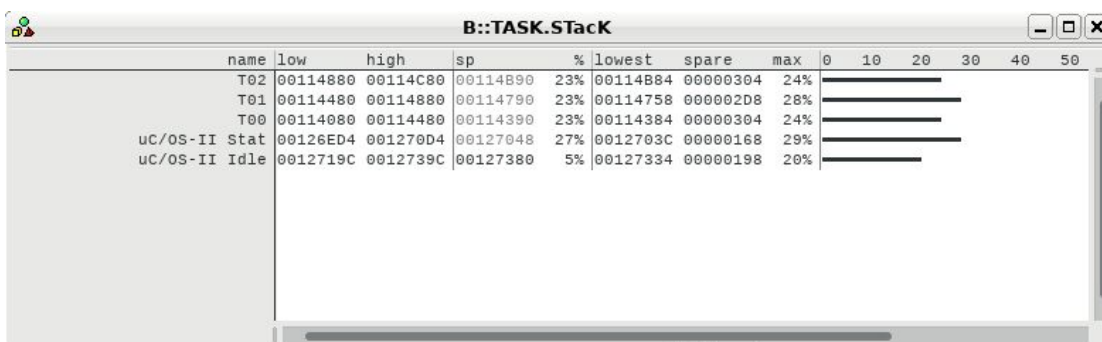
### 6.2.2. Teil 2

Es sollen zwei Tasks erstellt werden, die über eine Message Queue miteinander verbunden sind.



magic	id	prio	state	event	msg	delay	name
00127734	2.	30.	QUEUE	0012742C	00000000	0.	T02
001276DC	1.	31.	QUEUE	00127414	00000000	0.	T01
00127684	0.	32.	QUEUE	001273FC	00000000	0.	T00
001275D4	65534.	62.	DELAY	00000000	00000000	187.	uC/OS-II_Stat
0012757C	65535.	63.	RUNNING	00000000	00000000	0.	uC/OS-II_Idle

Abbildung 6.2.: Fenster zum Inspizieren der Stackausnutzung



name	low	high	sp	%	lowest	spare	max	0	10	20	30	40	50
T02	00114880	00114C80	00114B90	23%	00114B84	00000304	24%						
T01	00114480	00114880	00114790	23%	00114758	00000208	26%						
T00	00114080	00114480	00114390	23%	00114384	00000304	24%						
uC/OS-II Stat	00126ED4	001270D4	00127048	27%	0012703C	00000168	29%						
uC/OS-II Idle	0012719C	0012739C	00127380	5%	00127334	00000198	20%						

Abbildung 6.3.: Fenster zum Inspizieren von laufenden Tasks, ihren Prioritäten und IDs



- Erstellen Sie zwei Tasks
- Task 2 soll auf eine Nachricht von Task 1 mit einer Ausgabe über printf reagieren
- Nutzen Sie eine Message Queue um Task 2 zu informieren

Es gilt zu dokumentieren, was zu einem Task gehört, wie ein Task erstellt wird und wie entschieden werden kann, welcher Task von dem Betriebssystem als nächstes ausgeführt wird.

### 6.3. Post-Kolloquium

- Welche Bedeutung hat *UCOSStartup()* ?  
(Tipp: Suchen Sie in Eclipse nach der Funktion (Strg+H) und machen Sie sich mit dem Inhalt vertraut. Ist der Aufruf dieser Funktion für die korrekte Funktion des Programms notwendig?)
- Wie ist der Task Control Block aufgebaut? (Siehe Abbildung 6.1)
- Was ist eine Message Queue und warum wird sie genutzt?
- Welche Vorteile hat eine Message Queue?
- Über welche Parameter wird die UART Schnittstelle konfiguriert?
- In Aufgabe 1 haben Sie sich eingehend mit der Toolchain beschäftigt. Schauen sie sich nun einmal die Dateien `out/Aufgabe2_ps7_core0.lst` und `out/Aufgabe2_ps7_core0.map` an. Was steht in diesen Dateien und welche Informationen könnten Sie hier heraus ziehen?

```
1 typedef struct os_tcb {  
2  
3     OS_STK      *OSTCBStkPtr;  
4  
5     void        *OSTCBExtPtr;  
6  
7     OS_STK      *OSTCBStkBottom;  
8     INT32U      OSTCBStkSize;  
9  
10    INT16U      OSTCBOpt;  
11  
12    INT16U      OSTCBIId;  
13  
14    struct os_tcb *OSTCBNext;  
15    struct os_tcb *OSTCBPrev;  
16  
17    OS_FLAGS     OSTCBFlagsRdy;  
18  
19    INT8U        OSTCBStat;  
20    INT8U        OSTCBPrio;  
21  
22 } OS_TCB;
```

Quellcode 6.1: Ausschnitt aus dem Task Control Blocks in  $\mu C/OS-II$

# 7 Aufgabe 3

## 7.1. Wissen

### 7.1.1. JTAG-Debugging

#### Funktionsdefinition

Der Joint Test Action Group (JTAG)-Debugger ermöglicht es, Eingriffe in den Programmablauf vorzunehmen. Außerdem unterstützt er den Entwickler dabei, den Programmzustand zu inspizieren. Dazu lässt sich der Speicher auslesen und die daraus gewonnenen Informationen werden ausgewertet und zu Analysezwecken aufbereitet. Plattformen, die Multitasking unterstützen, bieten Übersichten zu laufenden Tasks. Damit wird das Überwachen der Nebenläufigkeit vereinfacht. Fortgeschrittene Debuggerprogramme bieten die Möglichkeit die Interprozesskommunikation, zum Beispiel Semaphoren und Nachrichten, auszuwerten. Der Standard in dem Bereich des Debuggings für eingebettete Systeme ist der GNU-Debugger (GDB), welcher Teil der GNU Compiler Collection (GCC) ist. Das JTAG-Interface hat den Zweck ein Verfahren zu ermöglichen, mit dem Schaltungen getestet werden können, während sie sich verlötet auf der Leiterplatte befinden [jta]. JTAG-kompatible Systeme haben im Normalbetrieb abgetrennte Komponenten, die erst dann aktiviert werden, wenn das JTAG-Interface genutzt werden soll. Technisch gesehen ist die Schnittstelle als Schieberegister verwirklicht. Das Zielsystem ist über das JTAG-Interface mit der Debugginghardware verbunden. Die Kommunikation zwischen der Entwicklungsplattform auf dem PC und der Debuggingplattform findet über USB statt (Abbildung 7.1).

#### Quelltextansicht

Im Gegensatz zu Assembler-Debugging kann der Code via High Level Language (HLL)-Debugging in der Quelltextansicht inspiziert werden (siehe Abbildung 7.2). Der Programmablaufzähler wird eingeblendet und es lässt sich nachvollziehen, an welcher Stelle im Quelltext sich das Programm gerade befindet. Diese Möglichkeiten bieten sich, weil der Compiler beim Erzeugen der Executable and Linking Format (elf)-Datei Debuginformationen hinzufügt. Diese werden von dem Debugwerkzeug interpretiert und die Assembler-Instruktionen werden den Zeilen im Quelltext zugeordnet.

#### (Single-)Stepping

Das Programm kann in Einzelschritten ausgeführt werden. Dabei können entweder die Schritte der Hochsprache oder der Assembler-Ebene einzeln ausgeführt werden. Außerdem ist es möglich, die aktuelle Methode zu Ende laufen zu lassen oder in die auszu-

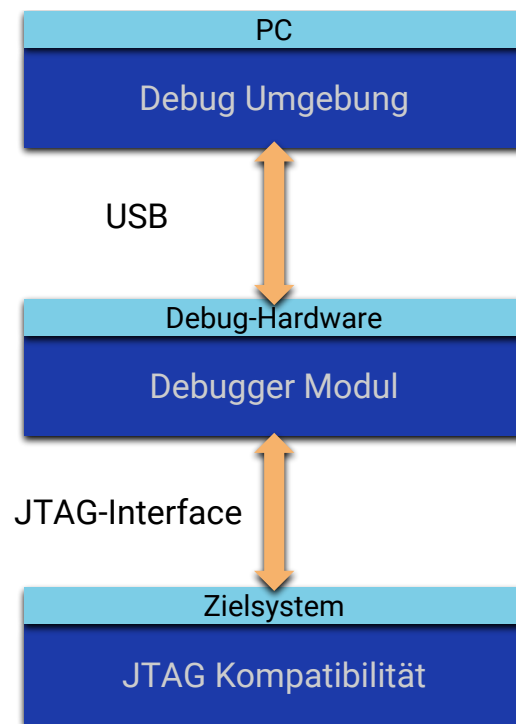


Abbildung 7.1.: Cross Debugging via Joint Test Action Group (JTAG)-Debugging

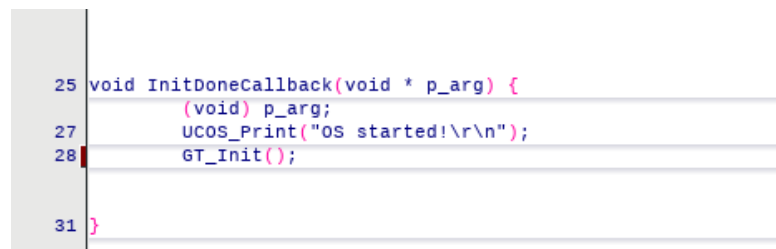
The screenshot shows the Lauterbach environment with the source code view of `main.c`. The code is as follows:

```
[ B::List.auto]
Step Over Diverge Return Up Go Break Mode Find: main.c
addr/line source
#include <mmu.h>
#include <stdio.h>
#include "ucos_uartps.h"
#include "ucos_ii.h"
#include "ucos_bsp.h"
#include "xil_testmem.h"
#include "xil_printf.h"

9 void InitDoneCallback(void * p_arg){
    (void) p_arg;
    while(1){
12         OSTimeDly(OS_TICKS_PER_SEC * 5);
13         UCOS_Print("I'LL BE BACK!\r\n");
    }
}

17 int main(void) {
18     MMUInit();
19     UCOSStartup(InitDoneCallback);
    //this should never been reached
21     while (1);
22 }
```

Abbildung 7.2.: Quelltextansicht in der Lauterbach Umgebung



```

25 void InitDoneCallback(void * p_arg) {
    (void) p_arg;
27     UCOS_Print("OS started!\r\n");
28     GT_Init();
    }
31

```

Abbildung 7.3.: Setzen eines Breakpoints in Zeile 28.

führende Unteroutine hineinzuspringen, beziehungsweise erst bei deren Rückkehr zu stoppen. Diese Möglichkeiten ergeben sich, wie auch die Quelltextansicht, aus den vom Compiler der elf-Datei hinzugefügten Debuginformationen.

### Starten und Stoppen des Programmablaufs

Das Programm kann angehalten werden. Dies kann hilfreich sein, wenn man an bestimmten Stellen im Programm Variablen auslesen möchte. Das manuelle Stoppen des Programmflusses ist allerdings sehr ungenau. Daher sollten für das gezielte Anhalten des Programms Breakpoints genutzt werden.

### Haltepunkt (Breakpoint)

Das Setzen eines Breakpoints beschreibt die Auswahl einer Stelle im Programmfluss, an der die Ausführung des Programms gestoppt wird, bevor der markierte Befehl ausgeführt wird. Aus [Gra, S. 115], Vorgehen beim Debuggen mit Breakpoints:

- Aufstellen einer These über die mögliche Position des Defekts
- Setzen eines Haltepunkts vor der vermuteten Position
- Annäherung mit Hilfe von Breakpoints / Stepping, dabei: Überprüfung des Programmzustands
- These falsch / Korrigieren des Defekts

Es wird zwischen Software und Hardware Breakpoints unterschieden [arm]. Erstere werden temporär in den RAM des Zielsystems geschrieben und ersetzen bis zum Eintritt des Breakpoints die ursprüngliche Instruktion. Diese wird durch eine Breakpoint-Instruktion überschrieben und die CPU geht bei der Ausführung in einen Debugstatus. Hardware Breakpoints werden durch das Überprüfen des Instruction Fetch von einer spezifischen Speicheradresse aus umgesetzt (siehe Abbildung 7.5). Im Gegensatz zu Software Breakpoints können Hardware Breakpoints auch auf Befehle aus dem ROM angewendet werden. Sollte eine Memory Management Unit (MMU) Adressbereiche neu zuordnen, so kann es zum Überschreiben von Software Breakpoints kommen.

Das Setzen eines Breakpoints erfolgt über einen Doppelklick neben die Programmzeile (siehe Abbildung 7.3).

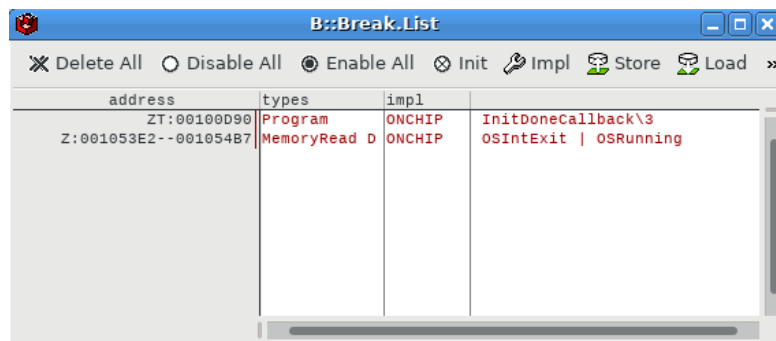


Abbildung 74.: Breakpoint-Übersichtsfenster in der Lauterbach Umgebung

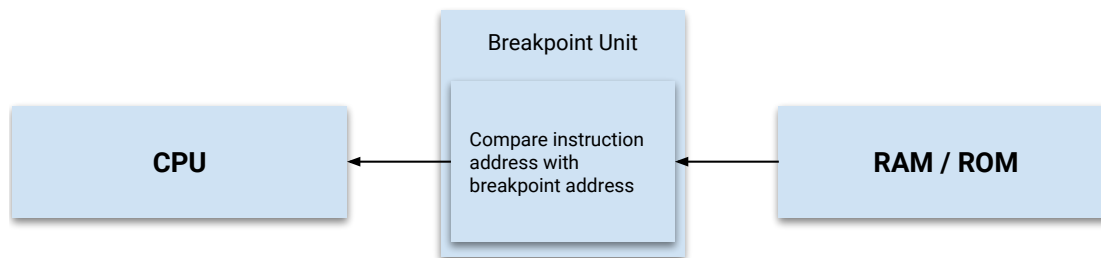


Abbildung 75.: Hardware Breakpoint Realisierung

### Überwachungspunkt (Watchpoint)

Ein Watchpoint überwacht eine gewünschte Variable und hält die Ausführung des Programms an, wenn diese verändert werden. Die Möglichkeiten der Überwachung hängen von dem genutzten Debugprogramm ab. Möglich ist zum Beispiel die Überprüfung auf einen Wertebereich oder auf Lese/Schreibzugriffe auf eine Variable. Nicht überwachen lassen sich alle Datenströme, die an der CPU vorbei laufen. Sollten Speicherbereiche zum Beispiel durch Direct Memory Access (DMA) verändert werden, so kann dies nicht mit Watchpoints an der CPU detektiert werden. Die gesetzten Break- und Watchpoints erscheinen im Übersichtsfenster, wo auch ihr Typ näher spezifiziert ist.

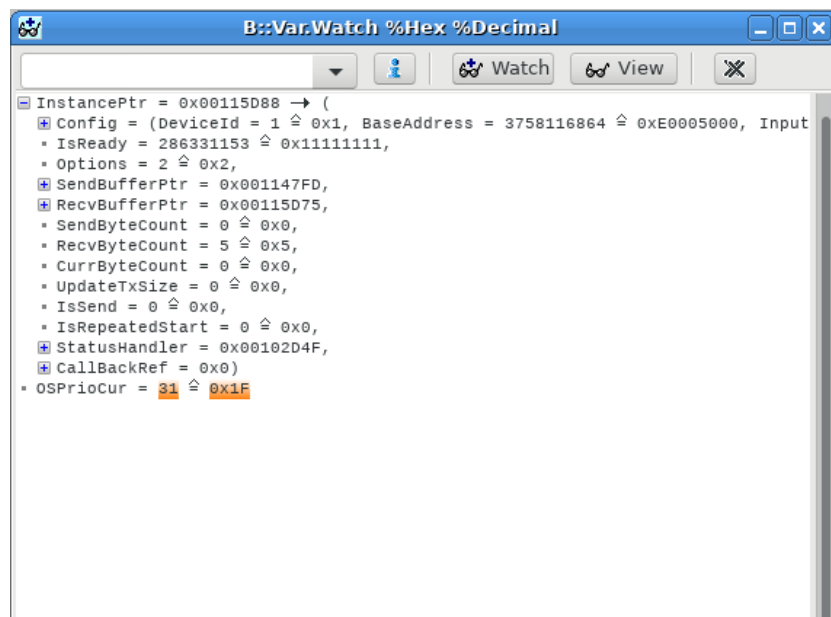


Abbildung 7.6.: Watch-Fenster Breakpoint Realisierung

### Speicherzugriff

Der Speicherzugriff ermöglicht das Auslesen des Speichers. Die Daten können in verschiedenen Formatierungen angezeigt werden. Es ist möglich den Inhalt direkt als ASCII String, hexadezimal, dezimal und binär darzustellen.

### Watch Fenster

Mit Hilfe der Debuginformationen aus der elf-Datei können die, aus dem Speicher gelesenen, Daten im Watch Fenster geordnet und entsprechend der zugehörigen Datenstrukturen dargestellt werden (siehe Abbildung 7.6). Es ist möglich, sich die Daten in Arrayform oder in anderen Formatierungen anzeigen zu lassen. Konstanten werden von dem Debugger nicht aufgelöst.

### Auswertung des Call Stacks

Die im Call Stack enthaltenen Daten lassen sich auswerten und eine Aufrufliste daraus rekonstruieren (siehe Abbildung 7.7). Außerdem werden beim Verlassen einer Unteroutine die lokalen Variablen auf den Call Stack gelegt und lassen sich von vielen Debuggern auswerten.

### 7.1.2. Beispiel: Schreiboperation auf Variablen überprüfen

Wenn es zu nicht nachvollziehbaren Änderungen von Variablen kommt ist, ist es sinnvoll diese mit Watchpoints zu überwachen. Es ist möglich, dass die Variable durch einen fehlerhaften Schreibvorgang einer anderen Variable beeinflusst wird. Ein Beispiel, wie es zu solch einer Situation kommen kann, ist in dem Quellcode 7.1 zu betrachten. Die Tabelle 7.1 zeigt, dass der in der globalen Variable *a* gespeicherte Wert durch die Schreiboperation

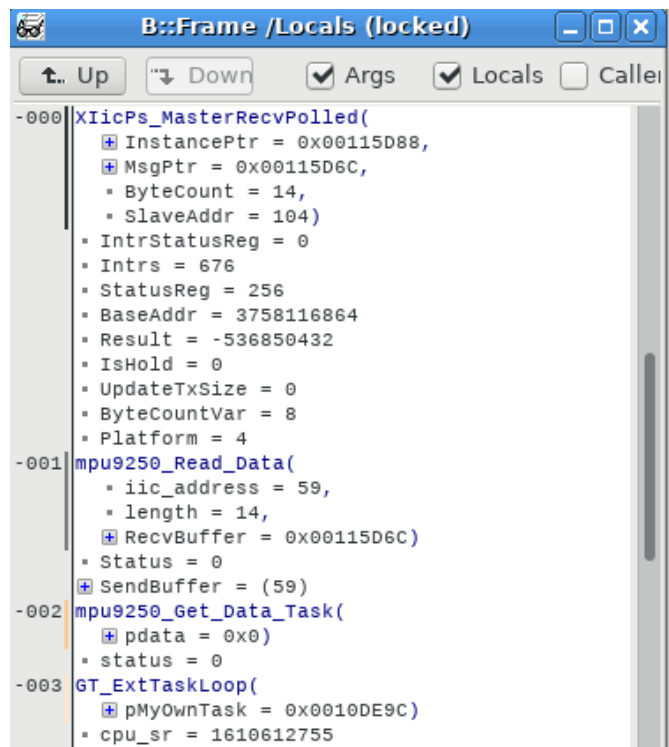


Abbildung 7.7.: Watch-Fenster Breakpoint Realisierung

auf `numbers[4]` überschrieben wurde. Das Ergebnis der Addition in Zeile 25 des Quellcodes 7.1 ist somit falsch.



```

1  /*
2  ** zur Veranschaulichung:
3  ** alle Variablen global und im gleichen Speichersegment
4  */
5
6  uint8_t numbers[4];
7  uint8_t a = 10;
8  uint8_t b = 15;
9  uint8_t result;
10
11 void main(void) {
12
13     /*
14     ** for-Schleife mit Fehler in Abbruchbedingung
15     ** Schreiboperation auf numbers[4]
16     ** -> fehlerhafte Daten in Speicherbereich der Variable a
17     */
18     for(uint8_t i = 0; i <= 4; i++){
19         numbers[i] = i;
20     }
21
22
23     /* falsches Ergebnis durch fehlerhafte Daten in Variable a */
24     result = a + b;
25
26 }

```

Quellcode 7.1: Beispielcode zum Überschreiben von Speicherbereichen und dadurch entstehende Folgefehler

Speicherbereich für globale Variablen							
Byteweise adressiert	0	1	2	3	4	5	6
Inhalt - soll	0	1	2	3	10	15	25
Inhalt - ist	0	1	2	3	4	15	19
Zugeordnete Variable	numbers[4]				a	b	result

Tabelle 7.1.: Visualisierung des Speicherinhalts bei Ausführung des Programms 7.1

### 7.1.3. Grenzen des JTAG-Debuggings

JTAG-Debugging eignet sich nur dann, wenn das System zur Erfassung des Fehlers auch pausiert werden kann. Sobald eine Analyse des Systems ausgeführt wird, wird das Zeitverhalten stark verändert, weil das System angehalten werden muss. Fehler, die auf Zeitverhalten beruhen, lassen sich damit nur schwer untersuchen. Dazu gehören auch Interrupts oder Unterbrechungen durch höherprioritäre Tasks. Es ist außerdem nicht möglich, im Programmablauf zurück zu gehen und sich den Hergang des Fehlers genau anzusehen. Dafür bedarf es der Aufzeichnung des Programmflusses.

### 7.1.4. Schrittmotoren

Schrittmotoren können schrittweise und somit sehr genau gesteuert werden. In unserem Anwendungsfall ist die schrittweise Ansteuerung allerdings nicht wichtig. Es ist interessant, in welcher Frequenz die Schritte ausgelöst werden, denn damit wird die Geschwindigkeit des Motors geregelt. Um die Schrittmotoren einfacher anzu steuern zu können, werden Schrittmotortreiber genutzt. Das Datenblatt zu dem Treiber A4988 findet sich dabei in der Quelle [All]. Für jeden Schritt, den der Motor machen soll, muss ein Puls an den Eingang des Schrittmotortreibers gesendet werden. Schauen Sie sich im Datenblatt zu dem Zynq-7000 [Xil18] das Kapitel zu dem Thema *Triple Timer Counter* an. Hier finden sich Informationen, wie man diese Pulse erstellen könnte, ohne dass man sich in der Software um das Zählen direkt kümmern müsste. Die Drehrichtung wird von einem Signal über GPIOs gesetzt. 1

## 7.2. Pre-Kolloquium

Versuchen Sie herauszufinden, wie man die Motoren mit Hilfe der Timer mit verschiedenen Geschwindigkeiten ansteuert. Halten Sie Ihre Ergebnisse zunächst schriftlich fest. Anregungen:

- Welche Vorteile bieten Timer gegenüber dem Zählen in Software?
- Schauen Sie sich die verschiedenen Zählmodi an: Wann startet der Timer wieder bei 0?
- Ist es möglich Signale auf GPIOs zu geben, wie kann davon Nutzen gemacht werden? Können diese Signale auch von Timern erzeugt werden?
- Finden Sie die Bedeutung von *Match Value* und *Interval Length* heraus
- Wie lang muss der Puls sein? Schauen Sie sich das Datenblatt des Motortreibers an.
- Wie können Sie in der Lauterbach Skriptsprache Breakpoints generieren, laden und speichern?

## 7.3. Aufgabe

Schreiben Sie die Software für das Ansteuern der Motortreiber. Es soll auf einen PID Wert zwischen -100 und 100 reagiert werden und die Motorleistung, sowie Drehrichtung entsprechend geregelt werden. Die Motoren drehen bis zu einer Pulsfrequenz von 500Hz flüssig. Als Mindestgeschwindigkeit sollen etwa 60 Schritte pro Sekunde gesetzt werden.

- Initialisieren Sie einen Timer, erstellen Sie dafür eine init-Methode

- Timer Control Struct definieren

```
1 typedef struct TimerStruct{
2     u32 OutputHz; /* Output frequency */
3     XInterval Interval; /* Interval value */
4     u8 Prescaler; /* Prescaler value */
5     u16 Options; /* Option settings */
6 } TmrCntrSetup;
```

- Timer Instanz erstellen (global)

```
1 XTtcPs (...)
```

- *TmrCntrSetup*-Struct erstellen (global)

- Erstellen Sie eine Konfigurationsinstanz für den Timer

```
1 XTtcPs_Config (...)
```

- Modus des Timers setzen, dazu das Feld Options des *TimerCntrSetup* Structs bearbeiten (Tipp: Welcher Timermodus soll gewählt werden? Wann soll ein HIGH/LOW am Ausgang erzeugt werden?)

- Die Konfigurationsinstanz des Timers füllen dazu die folgende Methode nutzen

```
1 XTtcPs_LookupConfig(XPAR_PS7_TTC_0_DEVICE_ID);
```

- Den Timer initialisieren

```
1 XTtcPs_CfgInitialize (...)
```

- Die in dem *TimerCntrSetup* gewählten Optionen anwenden

```
1 XTtcPs_SetOptions(...);
```

- Den Prescaler auf Basis der gewünschten Frequenz automatisch herausfinden

```
1 XTtcPs_CalcIntervalFromFreq(Timerinstanz, TimerCntrSetup->OutputHz,  
2   &(TimerCntrSetup->Interval), &(TimerCntrSetup->Prescaler));
```

- Prescaler setzen

```
1 XTtcPs_SetPrescaler (...)
```

- Match Value erstellen und setzen

```
1 XTtcPs_SetMatchValue(...)
```

- Intervalllänge erstellen und setzen

```
1 XTtcPs_SetInterval (...)
```

- Schreiben Sie Methoden zum Starten und Stoppen der soeben erstellten Timer (Hinweise finden sich in der Datei xtttps.h)
- GPIO Initialisierung für das Festlegen der Drehrichtung des Motors

- Erstellen Sie eine Konfigurationsinstanz für GPIOs

```
1 XGpioPs_Config (...)
```

- Erstellen Sie eine GPIO Instanz

```
1 XGpioPs (...)
```

- Implementieren Sie die Funktion

```
1 timer_gpio_Init()
```

- Füllen Sie die Konfigurationsinstanz (ähnlich wie bei I2C und der Erstellung des Timers)
    - Initialisieren Sie die GPIO-Instanz (Tipp: XPAR\_PS7\_GPIO\_0\_DEVICE\_ID)
    - Definieren Sie die Pins als Outputs mit

```
1 XGpioPs_SetDirectionPin(gpioInstanz, Pinnummer, 1);  
2 XGpioPs_SetOutputEnablePin(gpioInstanz, Pinnummer, 1);
```

- Erstellen Sie eine Funktion, die die Drehrichtung der Schrittmotoren in Abhängigkeit des PID Values bestimmt. Nutzen Sie dabei die Methode

```
1 XGpioPs_WritePin()
```

Weitere Informationen und Tipps:

- Ein Timer Tick entspricht einer Mikrosekunde
- Nutzen Sie den Timer `TTC0_0`
- Die GPIO-Pinnummer ist 54

Die Geschwindigkeit des Motors soll sich nun nach jedem Ablauf eines Intervalls ändern. Dazu bietet sich die Nutzung von Interrupts an! Jedes mal wenn ein Intervall zu Ende ist, wird der nächste Wert gelesen und die Intervalllänge aktualisiert. Auch die Drehrichtung soll gesetzt werden.

Wir erstellen einen Interrupt der dann ausgelöst wird, wenn der Timer das Ende des Intervalls erreicht hat. Die Interrupt ID ist `TTC_PWM_INTR_ID`. Dazu können die beiden folgenden Methoden genutzt werden.

```
1 UCOS_IntVectSet(...);
2 UCOS_IntSrcEn (...);
```

Die Interrupts werden nur dann ausgelöst, wenn der Timer entsprechend konfiguriert wird. Tipp: Nutzen Sie `XTTCPS_I XR_INTERVAL_MASK`

```
1 XTtcPs_EnableInterrupts (...)
```

Es muss nun eine Methode erstellt werden, die ausgeführt wird wenn der Interrupt aufgerufen wurde. In dem Interrupt muss das Interruptflag wieder zurückgesetzt werden, damit der normale Programmablauf fortgeführt werden kann.

```
1 u32 StatusEvent;
2 StatusEvent = XTtcPs_GetInterruptStatus (...);
3 XTtcPs_ClearInterruptStatus (...);
```

## 7.4. Post-Kolloquium

- Zeichnen Sie mit der Funktion `"iprobe.timing"` die Pulse auf die Sie mit dem Timer generieren.
- Wie können in Trace32 Breakpoints erstellt werden? Wie kann eine Bedingung angegeben werden?

## 8 Aufgabe 4

Nachdem in der vorherigen Aufgabe das Thema JTAG bereits angerissen wurde, soll es auch in Aufgabe 4 behandelt werden. Es wird eine weitere Komponente des Anwendungsfalls debuggt. Damit der Roboter jederzeit seinen Winkel zur Horizontalen kennt, benötigt er eine echtzeitfähige Lagemessung. Der Treiber für die Ansteuerung der inertialen Messeinheit über I<sup>2</sup>C wird mit Hilfe des JTAG-Debuggings und der Nutzung des Lauterbach Logik Analysators in einen fehlerfreien Zustand gebracht. Die Werte des Sensors werden den anderen Programmmodulen über globale Variablen zur Verfügung gestellt.

Die Aufgabe ist es, ein Programm zu schreiben, dass die aktuellen Werte des Gyroskops und des Beschleunigungssensors ausliest und in zwei globalen Variablen speichert. Maßgeblich für die Funktion des Moduls ist die korrekte Initialisierung des I<sup>2</sup>C Busses, die richtige Konfiguration der inertialen Messeinheit und das korrekte Umwandeln der Daten. Bei dieser Aufgabe wird neben dem JTAG-Debugging unterstützend ein Logik Analysator genutzt. Die Studierenden sollen mit Hilfe des Lauterbach Logic Analysers die übertragenen Daten auf dem I<sup>2</sup>C Bus einsehen und die gesendeten Informationen extrahieren. Zur Kontrolle kann die von Lauterbach zur Verfügung gestellte automatische Protokollanalyse des Logik Analysators genutzt werden.

### 8.1. Wissen

#### 8.1.1. Inertiale Messeinheit und Sensorfusion

Die MPU9250 von InvenSense bietet eine 9-Achsen Messeinheit mit Accelerometer, Gyroskop und Magnetometer. Die MPU9250 wird auch als Bewegungs- und Lagesensor in Smartphones genutzt. Die aus der Sensoreinheit ausgelesenen Werte müssen für die weitere Verwendung bearbeitet werden. Dabei ist es sinnvoll sich auf die Kippachse nach vorne und hinten zu konzentrieren. Diese Achse wird im Folgenden als Pitch bezeichnet. Um die Lage richtig einschätzen zu können, benötigt man zwei Messwerte von dem Sensor. Die Beschleunigung sowie die Winkelgeschwindigkeit. Um einen möglichst fehlerfreien Wert für den Pitch zu bekommen, müssen die Messwerte gefiltert werden. Die Probleme, die sich dabei auftun, sind folgende: Der Beschleunigungssensor ist sehr anfällig für Rauschen, also für kurzfristige Fehler. Dafür kann er die Winkelmessung nicht relativ, sondern absolut ausführen. Die Winkelgeschwindigkeit wird sehr genau gemessen und kaum von äußeren Einflüssen gestört. Der aus dem Gyroskop resultierende Winkel unterliegt durch die Integration der Messwerte einem gewissen Drift. Es ist nötig diese Sensordaten zu filtern und zu kombinieren und die Schwächen der beiden einzelnen Methoden damit auszugleichen.

Neben der in Aufgabe 3 erlernten Arbeitsweise mit JTAG-Debugging, ist Grundlagenwissen über die Funktionsweise von I<sup>2</sup>C nötig. Dazu können Sie sich [i2c] ansehen. Es sollte bekannt sein, wie ein I<sup>2</sup>C Gerät angesprochen wird, welche Pins dafür nötig sind und wie Registern im Zielgerät gelesen oder geschrieben werden. Für die Analyse mit dem Lauterbach Logic Analyser finden sich in [Lau14, S. 48-49] Informationen zur automatischen Protokollanalyse von I<sup>2</sup>C. Diese Informationen sind in dem Datenblatt [Inva] und der Register Map [Invb] der inertialen Messeinheit zu finden. Sie sollten über Wissen zu inertialen Messeinheiten verfügen und sich darüber im Klaren sein, warum eine Sensorfusion nötig ist und wie man sie realisiert [Pie].

## 8.2. Pre-Kolloquium

- Wie liest man aus einem I<sup>2</sup>C Gerät?
- Welche Register aus der IMU sind interessant für uns? Schauen Sie sich das Registerdatenblatt der IMU an und suchen Sie Register, die für unsere Anwendung interessant sind
  - Wie werden rauschende Signale geglättet?
  - Was wollen wir aus der IMU lesen?
  - Müssen wir die IMU erst aufwecken?
  - Wie konfiguriert man Gyroskop und Accelerometer so, dass beim Gyroskop Dps = 500 ist, ACC Skala = 4g ?
- Werten Sie den gegebenen I<sup>2</sup>C Datenstrom (siehe 8.1) aus und geben Sie Adresse und Inhalt der Nachricht wieder, handelt es sich um Lesen oder um Schreiben? (Tipp: Abgebildet ist nur der Datenstrom ausgehend vom Master)
- Warum benötigt man zur sicheren Winkelberechnung eine Sensorfusion aus Beschleunigungs- und Gyroskopdaten? (Machen Sie sich mit dem Komplementärfilter vertraut)

## 8.3. Aufgabe

Erstellen Sie einen neuen Task und starten Sie ihn so, wie in Aufgabe 2 gelernt. Die Taskfunktion soll in der Datei *imu.c* implementiert sein. Nennen Sie den Task:

```
1 mpu9250_Get_Data_Task()
```

Schreiben Sie die Funktion die zur Initialisierung und zum wiederholten Auslesen von Daten aus der IMU benötigt wird. Gehen Sie dazu wie in dem folgenden Abschnitt beschrieben vor. Erstellen Sie eine Methode zur Initialisierung der I<sup>2</sup>C Instanz:

- globale I<sup>2</sup>C Instanz erstellen

```
1 XlicPs Iic ;
```

- in Funktion: Config Struct erstellen

```
1 XlicPs_Config *Config;
```

- Config-Struct mit XlicPs Config füllen, (Tipp: Informationen zu Übergabeparametern finden sich in der *imu.h*)

```
1 Config = XlicPs_LookupConfig(...);
```

- I<sup>2</sup>C initialisieren und Status abfragen, übergeben Sie der Methode die benötigten Parameter (Tipp: Basisadresse des I<sup>2</sup>C findet sich auch in dem Config-Struct)

```
1 XlicPs_CfgInitialize (...) ;
```

- Zur Sicherheit einen Self-Test machen und das Ergebnis abfragen (in die Methode gucken, um den Rückgabewert interpretieren zu können)

```
1 XlicPs_SelfTest (...) ;
```

- I<sup>2</sup>C Clockrate setzen, suchen Sie in der *imu.h* nach Übergabeparametern

```
1 XlicPs_SetSCLk (...) ;
```

Das Nutzen der I<sup>2</sup>C Schnittstelle erfolgt über zwei Methoden zum Senden und Empfangen von Daten aus den **Registern** des MPU9250 Sensors. Erstellen und implementieren Sie die beiden benötigten Methoden in der Datei *imu.c*. Basierend auf der zuvor gelesenen Theorie: Überlegen Sie sich welche Übergabe- und Rückgabeveriablen die Methoden benötigen. Nennen Sie die Methoden wie folgt:

- Senden:

```
1 mpu9250_Write_Reg(...)
```

- Empfangen:

```
1 mpu9250_Read_Reg(...)
```

Die benötigte Schnittstelle des I<sup>2</sup>C Treibers von Xilinx sind die Methoden

```
1 XlicPs_MasterSendPolled()
```



und

```
1 XlicPs_MasterRecvPolled()
```

Schauen Sie sich die Methoden an und identifizieren Sie die benötigten Übergabeparameter. Implementieren Sie auch hier eine Fehlerüberprüfung.

Nach der erfolgreichen Initialisierung der I<sup>2</sup>C Schnittstelle und den Methoden zum Senden und Empfangen von Daten ist es vonnöten die inertielle Messeinheit zu initialisieren. Nutzen Sie dafür die vor dem Präkolloquium ausgearbeiteten Register. Erstellen Sie für die Initialisierung eine eigene Methode in der Datei *imu.c*.

Nach dem Initialisieren der IMU sollen Sie eine Methode zum Auslesen der Accelerometer- und Gyroskopdaten erstellen. Lesen Sie die Daten der IMU an einem Stück aus und vermeiden Sie schnell aufeinander folgende Lesevorgänge. Es lassen sich alle Register mit den benötigten Rohwerten mit einem Lesevorgang auslesen. Nutzen Sie die serielle Schnittstelle, um die Daten anzuzeigen. Initialisieren Sie I<sup>2</sup>C und IMU bevor Sie versuchen auf die Daten des Sensor zuzugreifen. Falls dies nicht klappt, ist es sinnvoll den Status des Sensors auszulesen. Vergleichen Sie dazu den Inhalt des Statusregisters mit dem angegebenen Sollwert. Gehen Sie systematisch alle Fehlerquellen durch, die die Kommunikation mit dem Sensor unterbinden könnten. Einen Fehler in der Hardware können Sie ausschließen. Nutzen Sie den Lauterbach Logik Analysator (Untermenü 'Probe' -> 'Timing') um die I<sup>2</sup>C Übertragung zu inspizieren. Verifizieren Sie die Rohwerte durch den Übungsleiter. Außerdem müssen die Accelerometerdaten skaliert werden. Für die Winkelberechnung ist es nötig, dass das Accelerometer die Erdbeschleunigung auf jeder der drei Achsen gleich misst (von Werk aus nicht der Fall). Mappen Sie die Daten der einzelnen Achsen des Accelerometers auf den gleichen Wertebereich (finden Sie dazu zunächst die Maximal- und Minimalwerte für jede Achse bezüglich der Erdbeschleunigung heraus). Die Gyroskopdaten müssen mit einem passenden Offset kalibriert werden.

Im Folgenden sollen die gemessenen Werte in eine Winkelangabe transformiert werden. Berechnen Sie zwei Winkel: Der erste soll auf Basis der Accelerometerdaten (Stichwort: `atan2()`) berechnet werden, der zweite auf Basis der Gyrodaten (Stichwort: Winkelgeschwindigkeit -> Winkel). Überprüfen Sie die beiden Winkel auf ihre Plausibilität. Implementieren Sie nun den oben beschriebenen Komplementärfilter, der die beiden Winkel zu einem fusioniert (orientieren Sie sich an der Quelle).

Schauen Sie sich an, wie ein PID Regler implementiert wird und realisieren Sie ihn in diesem Projekt. Erstellen Sie dazu eine neue Datei namens *pid.c*. Integrieren Sie den programmierten Komplementärfilter und PID Regler als Task in der Datei *pid.c* und realisieren Sie die Kommunikation der Tasks über globale Variablen. Überlegen Sie sich dafür, welche Komponenten des Reglers welchen Werten im Code entsprechen (Parallelanteil, Integralanteil, Ableitungsanteil). Die Parametrisierung des Reglers wird in Aufgabe

7 geschehen, Sie können die Ausgabe des Reglers aber schon wie im Motortreiber-Code vorgesehen auf  $\pm 100$  begrenzen.

Tipps:

- Achten Sie auf das Einfügen der Datei in dem richtigen Ordner.
- Schauen Sie sich die IMU an und interpretieren Sie die Achsausrichtung um herauszufinden welche Achsen für die Berechnung der Winkel aus Accelerometer- und Gyroskopdaten nötig sind.
- Was macht die Funktion `atan2()`? Zu welcher Library gehört sie? Es gibt eine Besonderheit die zur Verwendung der Library erfüllt sein muss, welche ist es?
- Wie erstellt man aus dem Gyroskopwert dem aktuellen Winkel? Führen Sie für die Sample Time eine Konstante ein, die später an die echte Ausführungsperiode angepasst werden kann

## 8.4. Post-Kolloquium

Bereiten Sie sich auf das Kolloquium vor, indem Sie sich die Arbeitsergebnisse und gefundenen Fehler gut dokumentieren. Verstehen Sie, was Sie programmiert haben.

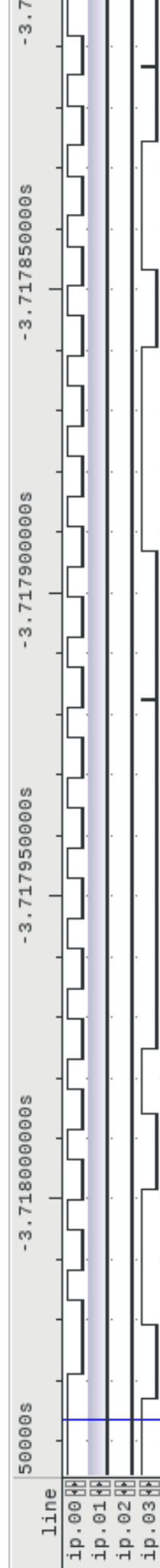


Abbildung 8.1.: I²C Datenstrom zum Auswerten

# 9 Aufgabe 5

## 9.1. Wissen

In dieser Aufgabe soll das Wissen in Bezug auf betriebssystemeigene Funktionen erweitert werden. Zur Erinnerung um im Zweifelsfall den richtigen Pfad zu finden: Der Zynq-7000 arbeitet mit Cortex A9 Cores. Wir nutzen die GNU Toolchain.

Alle Ergebnisse dieser Aufgabe werden im Post-Kolloquium besprochen.

## 9.2. Aufgabe

- Ziel soll es zunächst sein, die Methode

```
1 UCOSStartup()
```

zu untersuchen.

- Warum wird diese Methode noch vor dem Code aus der Main aufgerufen?
- Welchen Sinn hat diese Methode?
- Was macht die Methode:

```
1 CPU_Init()
```

- Welche Aufgabe hat die Funktion:

```
1 Mem_Init()
```

Muss diese Funktion aufgerufen werden?

- Betrachten Sie die Funktion

```
1 OSInit()
```

- Was ist in diesem Kontext ein *Hook*?
- Warum werden die Methoden

```
1 OS_InitMisc()  
2 OS_InitRdyList()
```

```
3 OS_InitTCBList()
```

benötigt? Was würde passieren, würden sie nicht ausgeführt werden.

- Inwiefern wird in dieser Methode Multitasking vorbereitet?
- Was ist ein *idle*-Task und wofür wird er benötigt? Wo wird er erstellt? Welche Priorität sollte er haben?
- Was ist ein *Startup*-Task? Was passiert dort?
- Erklären Sie den Sinn der Methode

```
1 OSStart()
```

- Was passiert in der Funktion

```
1 OSTaskCreateExt()
```

- Was macht die Funktion

```
1 OSTaskStackInit()
```

- Was macht folgende Methode:

```
1 OS_TCBInit()
```

- Wie funktioniert die Methode

```
1 OStimeDly()
```

- Untersuchen Sie die Methoden, warum wird eine *kritische Sektion* betreten?

```
1 OS_Sched() und OS_SchedNew()
```

- Nachdem der Task erstellt wurde muss das Betriebssystem auch mit dem neuen Task arbeiten.

```
1 OSStartHighRdy()
```

Wofür sind in dieser Funktion die Anweisungen in Zeile 202 bis 205 zuständig? Warum ist ein Ausrufezeichen in manchen Befehlen? Was macht das Zirkumflex hinter dem Befehl?

- Was passiert bei einem Kontextswitch? Schreiben Sie die Arbeitsschritte auf. Finden Sie heraus in welcher Methode der Kontextswitch ausgeführt wird. Welche wesentlichen Schritte werden abgearbeitet?

# 10 Aufgabe 6

## 10.1. Tracing

### 10.1.1. Funktionsdefinition

Unter Tracing bezeichnet man das Aufzeichnen des Programmablaufs, um diesen dann später zur Analyse von Fehlern zu nutzen. Die Analyse kann nach der Ausführung des Programms stattfinden. Auch Lese- und Schreibzugriffe auf Variablen können aufgezeichnet werden. Abstürze, bei denen die Ursache in einem Speicherüberlauf oder Nullpointer-Exceptions vermutet wird, können damit gelöst werden. Tracing kann auf verschiedene Arten realisiert werden, die jeweils eigene Vor- und Nachteile mit sich bringen.

#### Software-Trace

Das untersuchte Programm wird so verändert, dass es die benötigten Informationen selbst erzeugt. Dazu werden die gesammelten Daten in Variablen in den Zielgerät-RAM geschrieben und später vom Debugger ausgelesen. Vorteile dieser Variante sind, dass die Daten in beliebigem Umfang und beliebig genau bereitgestellt werden können. Gleichzeitig ist allerdings zu bedenken, dass die Hardware, auf der das Programm läuft, nun auch die Datensammlung bewerkstelligen muss. Die logische Konsequenz ist die Verringerung der Geschwindigkeit der Ausführung und ein erhöhter Speicherbedarf. Auf Systemen, auf denen kaum Leistung und Speicher zur Verfügung stehen, kann dies zu Problemen führen. Außerdem ergibt sich aus dieser Variante des Tracings ein hoher Einfluss auf das Zeitverhalten des Systems. Bestehende Fehler können, während der Ausführung mit Software Tracing, anders auftreten als ohne Software-Tracing (siehe Kapitel ??). Als Beispiel soll hier ein Programm dienen, welches durch einen externen Interrupt beeinflusst wird: Die Anwendung reagiert auf einen Interrupt und stürzt im betrachteten Fall ab. Dieser Absturz findet immer genau dann statt, wenn der Interrupt aktiviert wird, während sich das Programm in Methode XY befindet. Der Versuch diesen Fehler mit Hilfe von Software-Tracing zu lösen, verändert die Laufzeit des Programms so, dass der Interrupt nun zu einem anderen Zeitpunkt im Programm auftritt. Das Programm stürzt nun während des Debuggens nicht mehr ab. Jede dem Debuggen dienende Veränderung ändert das Zeitverhalten.

#### Offchip-Trace

Im Gegensatz zum Software-Tracing kommt diese Methode des Tracings nicht ohne externe Trace-Hardware aus (siehe Abbildung 10.2). Sollen Informationen zum Zustand des Systems zur Laufzeit aufgenommen werden, werden diese am Prozessor des Zielgerätes



Abbildung 10.1.: Lauterbach Trace Debugger PowerTrace-II

abgenommen. Die übliche Methode bei Mikroprozessoren ist das Auslesen des Adressbusses zum Speicher und einiger Steuersignale. Mithilfe dieser Daten kann der Programmlauf rekonstruiert werden. Bei modernen Chips sind CPU Kerne, Haupt- und Massenspeicher, Cache und Peripherie in einem Gehäuse integriert. Das macht es unmöglich den Speicherbus abzugreifen. Um diese Systeme trotzdem noch mit Trace-Debugging nutzen zu können, werden sogenannte Trace-Interfaces bereitgestellt. Auf ihnen wird in komprimierter Form der Programmfluss übertragen. Es handelt sich dabei meist um ein 4, 8 oder 16 Bit breiten Bus, über den mit Frequenzen bis 400 MHz Daten übertragen werden. Die bereitgestellten Informationen liegen so vor, wie sie auch in der CPU vorliegen würden, das heißt es werden auch Speicherzugriffe aufgezeichnet. Es muss sich nicht um etwaige fehlende Informationen über Lese-, aber vor allem Schreibzugriffe auf den Cache gekümmert werden.

### Onchip-Trace

Diese Methode des Tracings kommt ohne externen Trace-Speicher aus. Es gibt CPUs mit einem Trace-Speicher, der in das System integriert ist (siehe Abbildung 10.3). Auf diesem werden ähnlich der Methode des Offchip-Tracings die benötigten Daten gespeichert und können nach Beenden des Programms ausgelesen werden. Vorteil gegenüber dem Software-Tracing ist, dass keine Änderungen am Programm vorgenommen werden müssen. Im Vergleich zum Offchip-Tracing wird zwar kein externer Trace-Speicher benötigt, allerdings ist der Speicher des internen Trace-Speichers, aus Kostengründen und um Platz zu sparen, sehr klein gehalten. Um diesen Nachteil zumindest teilweise zu kompensieren, gibt es häufig die Möglichkeit das Programm zu stoppen und einen Interrupt auszulösen, wenn der interne Trace-Speicher voll ist. Dann können die Daten auf die Debugplattform übertragen und das Programm weiter ausgeführt werden. Der Nachteil dieser Variante ist



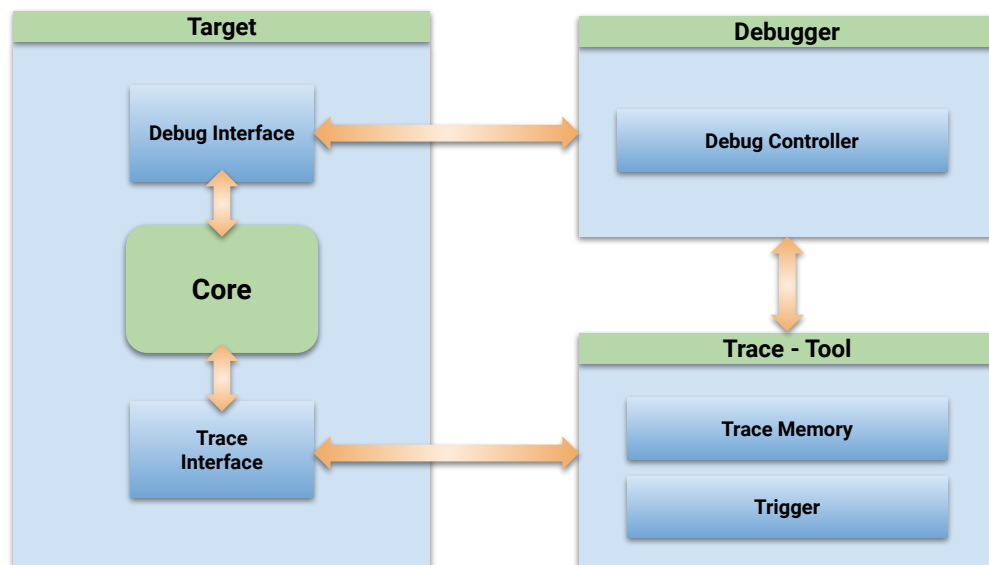


Abbildung 10.2.: Funktionsweise Offchip Tracing

das benötigte Starten und Stoppen und der damit verbundenen Eingriff in das Zeitverhalten des Systems.

### 10.1.2. Art der Anwendung, Nutzung des Werkzeugs

Die Anwendungsmöglichkeiten von Tracing sind vielfältig. Oft ist es nötig, die Ausführungszeit einer Methode zu kennen. Auch ist es nützlich die Register und Variablen ohne Unterbrechung des Programms auslesen zu können. Wenn ein Programm abstürzt, macht es Tracing möglich genau nachzuvollziehen, welche Funktionsaufrufe mit welchen Werten vor dem Absturz getätigt wurden. Programmkomponenten, die sich mit den vorherigen Debugwerkzeugen nur schwer oder gar nicht analysieren ließen, können nun auf ihre Auswirkungen auf den Programmfluss überprüft werden. Dazu gehört zum Beispiel erhöhte Interruptlast oder Unterbrechung durch höherpriorie Tasks. Fehleranalyse von Fehlern zur Laufzeit und die Analyse von Kommunikation über Busse wird einfacher. In dieser Aufgaben sollen sie Trace-Points nutzen um den Wechsel von einer Funktion zur anderen darzustellen. Diese Tracepoints können zum Beispiel ein Taskset visualisieren, indem sie zu Beginn und am Ende eines Tasks gesetzt werden.

### 10.1.3. Grenzen und Nachteile

Trotz dessen, dass die beiden in Kapitel 10.2 vorgestellten Trace-Methoden von den Nachteilen bezüglich des Heisenbergeffekts beim Software Tracing nicht betroffen sind, haben sie Nachteile. Dazu gehört der sehr hohe Kaufpreis solcher Systeme, der sich im niedrigen fünfstelligen Bereich bewegt. Außerdem ist die Unterstützung von Tracing unter den Entwicklungsboards wesentlich weniger weit verbreitet, als das bei JTAG-Debugging der Fall ist. Das Interface benötigt aufgrund der hohen Datenraten bei der Übertragung

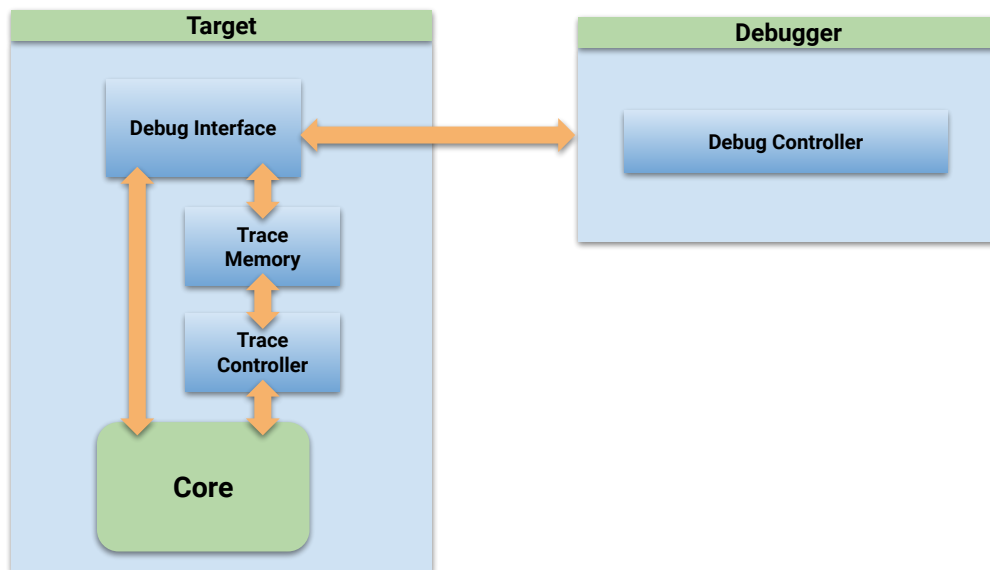


Abbildung 10.3.: Funktionsweise Onchip Tracing

viele Pins. Unterschiede zwischen den verschiedenen Trace-Methoden lassen sich in der Tabelle 10.1 finden.

Trace - Methode	Trace-Hardware	Trace - Größe	Programm-anpassung	Echtzeit-verletzung
Software Trace	keine	klein	ja	erheblich
OffChip Trace	ja	groß	keine	keine
OnChip Trace	keine	klein	keine	keine

Tabelle 10.1.: Vergleich von Software-, Onchip- und Offchip- Trace

Diese Aufgabe soll Ihnen Methoden vermitteln, mit denen Sie die Auslastung des Systems beurteilen und optimieren können. Grundstein für die Überlegungen dieser Aufgabe ist ein System mit mehreren Tasks. Es stellt sich nun die Frage, wann welcher Task CPU-Zeit in Anspruch nehmen darf. In der Aufgabe lernen Sie verschiedenen Scheduling Algorithmen kennen und analysieren sie auf die Auswirkungen auf das Task-Set. Alle benötigten Informationen zur Bearbeitung bekommen Sie aus dem Folien der Übung und

Vorlesung zur Lehrveranstaltung *Rechnerstrukturen 2*. In den Vorlesungsfolien sind aus Kapitel 5, Folien 29 bis 59 relevant. Kapitel 9 und 10 der Übung helfen beim Anwenden der Verfahren.

## 10.2. Aufgabenstellung

Bearbeiten Sie das Arbeitsblatt und notieren Sie sich Ergebnisse und Vorgehensweisen. Beantworten Sie außerdem folgende Fragen:

- Was ist der Unterschied zwischen Analyse und Simulation?
- Welche Arten der Taskaktivierung gibt es?
- Was sind *Arrival Curves*?
- Erklären Sie die Begriffe *Periode* und *Jitter*
- Was ist *Präemption* im Bezug auf Scheduling?
- Halten Sie die Eckdaten von TDMA und Round Robin fest
- Was ist RMS?

Kompilieren und Laden Sie nach dem Bearbeiten des Arbeitsblattes das Programm aus *Aufgabe 6* auf das Board. Untersuchen Sie mit Hilfe der Tracingtechniken des Boards das Taskset. In dieser Aufgabe wird ein Tasksetsimulator genutzt. Es lassen sich mehrere Tasks nach dem P-J-D Modell konfigurieren. Diese sind in der Datei *src/gt\_tasks.c* definiert. Die Tasks werden von dem Taskset-Simulator periodisch aktiviert. In der Datei findet sich die Methoden

```
1 GT_TaskStartHook
2 GT_TaskEndHook
```

Mit ihnen lassen sich Aktionen zu Beginn und am Ende der periodisch aktivierten Tasks auslösen. Schreiben Sie in dieser Methode eine Funktion, die es erlaubt zu unterscheiden welcher Task aktiviert wurde. Mit der Hilfe dieser Funktion können nun passende Tracepoints gesetzt werden um den Wechsel zwischen den Tasks darzustellen. Aufgrund der begrenzten Möglichkeiten des OnChip Tracing können nur vier Trace-Enable Punkte gesetzt werden. Alternativ lassen sich auch noch die Punkte Trace-On und Trace-Off nutzen. Setzen Sie die Trace Punkte und starten Sie das Programm. Stoppen Sie es wieder und wählen Sie oben in der Menüleiste *Trace* und dann *Chart* und den Unterpunkt *Symbols*. Versuchen Sie die Wechsel in und aus den Tasks mit Hilfe von Tracing darzustellen.

Task	Periode	Jitter	BCET	WCET	Priorität	SPP		SPNP	
						BCRT	WCRT	BCRT	WCRT
Task 1	2ms	0ms	0,5 ms	1,0 ms					
Task 2	1ms	0ms	0,1 ms	0,2 ms					
Task 3	10ms	0ms	0,2 ms	0,2 ms			4,0 ms		9,7 ms
Task 4	5ms	0ms	0,2 ms	1,0 ms					
Task 5	20ms	0ms	0,2 ms	0,5 ms			9,7 ms		8,3 ms

### Aufgabe 1:

Vergeben sie die Prioritäten der Tasks nach RMS! 0 ist die höchste Priorität.

### Aufgabe 2:

Berechnen sie die maximale Last auf dem Prozessorkern!

### Aufgabe 3:

Berechnen sie für das nun gegebene Taskset die BCRT sowohl für SPP als auch für SPNP Scheduling!

### Aufgabe 4:

Ermitteln sie für **Task 1** und **Task 2** die WCRT, sowohl für SPP als auch SPNP, **zeichnerisch!**

### Aufgabe 5:

Ermitteln sie für **Task 4** die WCRT, sowohl für SPP als auch für SPNP, **rechnerisch!**

### Aufgabe 6:

Angenommen die Aktivierung von **Task 1** hat einen **Jitter von 0,5ms**, wie wirkt sich dies bei **SPP** Scheduling auf die BCRT und WCRT von **Task 2** und **Task 4** aus?

Task	BCRT SPP	WCRT SPP
Task 2		
Task 4		

RMS: Rate Monotonic Scheduling	BCRT / WCRT: {Best / Worst} Case Response Time
SPP: Static Priority Preemptive	BCET / WCET: {Best / Worst} Case Execution Time
SPNP: Static Priority Non-Preemptive	

# 11 Aufgabe 7

## 11.1. Aufgabenteil 1

In dieser Aufgabe sollen die zuvor erstellten Programmodule zusammengesetzt werden. Es soll ein balancierender Roboter entstehen. Erstellen Sie auf Basis des in Aufgabe 6 genutzten Taskset-Simulators ein Taskset, dass ihre Tasks periodisch ausführt. Denken Sie daran ihre Init-Funktionen einzutragen. Nachfolgend werden die Anforderungen an die bisherigen Programme aufgelistet:

### 11.1.1. PID

Eingänge:

- Beschleunigungssensordaten
- Gyroskopdaten
- Sample Time

Funktion

- Winkel aus Accelerometer und Gyroskop berechnen
- Winkel miteinander Fusionieren
- PID Wert erstellen

Ausgänge

- PID Wert zwischen -1000 und 1000

### 11.1.2. IMU

Funktion

- IMU initialisieren
- Werte über I2C auslesen

Ausgänge

- Beschleunigungssensordaten
- Gyroskopdaten

Zusätzliche Informationen: I2C Soll-Geschwindigkeit : 100kHz

### 11.1.3. Motortreiber

Eingänge

- PID Wert

Funktion

- Ansteuern der Richtung der Motoren für vorwärts und rückwärts
- Regeln der Frequenz der Steps: max. Frequenz: min 1700us/step, max 250us /step

Tipps:

- zweiten Timer für zweites Rad anlegen
- PID Werte wie folgt wählen

```
1 #define KP 300.0
2 #define KD 0.0
3 #define KI 200.0
```

Quellcode 11.1: PID-Code Hilfe

- Achten Sie auf die richtige Sample-Time
- GyroOffset und AccOffset korrekt berücksichtigen

```
1 int16_t accMin[3] = {-7150, -7750, -9800};
2 int16_t accMax[3] = {9400, 9000, 7100};
```

Quellcode 11.2: PID-Code Hilfe

- Achten Sie auf die korrekte Bestimmung des Gyro-Offsets:

```
1 /* * * * * * clears Gyro offset and sets it new * * * * * */
2 int16_t gyroOffset [3];
3 unsigned int pid_Init(void *pdata){
4     uint8_t i = 0;
5     uint8_t t = 0;
6     for (i = 0; i < 3; i++) {
7         gyroOffset[i]=0;
8     }
9     for (t = 0; t <10; t++) {
10         for (i = 0; i < 3; i++) {
11             gyroOffset[i ]=( gyroData[i ]/10) ;
12         }
13         OSTimeDly(OS_TICKS_PER_SEC/10);
14     }
15     return 0;
16 }
```

## Quellcode 11.3: PID-Code Hilfe

- der Winkel aus dem Acc wird bei Verwendung der Achsen 1 und 2 (ersten beiden Beschleunigungswerte aus dem Registerset) entgegengesetzt zu dem des Gyros berechnet, daher:

```
1 accAngle = -accAngle;
```

## Quellcode 11.4: PID-Code Hilfe

- Nutzen Sie für den Komplementärfilter folgende Parameter:

```
1 currentAngle = 0.999*(prevAngle + gyroAnglePerStep) + 0.001*accAngle;
```

## Quellcode 11.5: PID-Code Hilfe

- Begrenzen Sie den integrierten Fehler des Winkels zum Sollwinkel auf +/- 750
- Kontrollieren Sie ihre PID Wert Berechnung

```
1 pidValue = (int16_t)(KP*(errorAngle) + KI*(errorSumAngle)*SAMPLE_TIME + KD/
    SAMPLE_TIME*(currentAngle - prevAngle));
```

## Quellcode 11.6: PID-Code Hilfe

- Begrenzen Sie den PID-Wert auf +/- 100
- Teilen Sie nicht durch 0 (Achtung bei Teilen mit PID-Wert)

## 11.2. Aufgabenteil 2

Sie finden sich nun in einem Szenario der Wirtschaft: Ihr Vorgesetzter schränkt ihre Ressourcen auf dem genutzten Steuergerät ein, weil diese anderweitig genutzt werden. Verändern Sie dazu in der Datei *gt\_tasks.c* das Struct *GT\_AllTasks* wie folgt und fügen Sie die Zeile

```
1 {50, 0, 0, {3, 5}},
```

## Quellcode 11.7: Ergänzung Task Timing Struct

hinzu. Außerdem sollen Sie in dem Struct *GT\_Tasks* die Zeile

```
1 { GT_TASK_INT, GT_ACT_INT, 3, 29, _GT_GetNextActivation, _GT_GetTaskCet, NULL
    , NULL, NULL, 20, GT_RUNABLE_NULL, GT_INTERNAL_NULL, (void *)&
    GT_AllTasks[3]},
```

---

### Quellcode 11.8: Ergänzung Task Struct

hinzufügen. Achten Sie darauf die Konstante `GT_NUM_OF_TASKS` anzupassen. Ihre Aufgabe ist es nun, die Funktionalität des Roboters wiederherzustellen. Untersuchen Sie dafür, wo im Programm CPU Zeit eingespart werden kann.

Um die in diesem Bereich genutzte CPU-Zeit zu reduzieren bietet es sich an die Kommunikation mit der inertialen Messeinheit zu verändern. Die Hardwareeinheit des Xynq-7000 macht es möglich nach jeder abgeschlossenen Kommunikation einen Interrupt auszulösen. Wir können die Kontrolle somit während des Senden an andere Tasks abgeben. Orientieren Sie sich für das Erstellen und Verknüpfen des Interrupts an dem des Timers. Suchen Sie nach geeigneten Funktionen vergleichbar zu denen, die bei der Einrichtung des Timer-Interrupts genutzt wurden. Grundsätzlich sollten Sie wie folgt vorgehen:

- Erstellen Sie einen Interrupt und suchen Sie die korrekte Interrupt-ID heraus, geben Sie als Interrupt-Handler *MasterInterruptHandler* an.
- Die I2C Instanz hat einen Statushandler der angegeben werden kann, er bietet sich an um die Semaphore zu pushen
- Die Funktionen *XIicPs\_MasterSendPolled* und *XIicPs\_MasterRcvPolled* haben passende Gegenstücke zur Nutzung mit Interrupts der I2C Hardware. Nach dem Aufrufen der Funktion muss auf die Fertigstellung gewartet werden. Dies wird über den Semaphore realisiert.



# 12 Aufgabe 8

## 12.1. Wissen

Hier Lehrtext zu der in Kapitel ?? beschriebenen benötigten Theorie einfügen. Als Grundlage dienen die beigefügten Quellen.

## 12.2. Aufgabenstellung

Hier Aufgabe zu den in Kapitel ?? beschriebenen Inhalten und auf Basis der Implementierungen aus dem HiWi Job ausformulieren.

# Akronyme

<b>DMA</b>	Direct Memory Access
<b>elf</b>	Executable and Linking Format
<b>GCC</b>	GNU Compiler Collection
<b>GDB</b>	GNU-Debugger
<b>HLL</b>	High Level Language
<b>JTAG</b>	Joint Test Action Group
<b>MMU</b>	Memory Management Unit

# A Anhang

## A.0.1. Parameter

- I2C Busgeschwindigkeit: 100000
- I2C Device ID: XPAR\_XIICPS\_1\_DEVICE\_ID
- MPU9250 I2C Adresse: 0x68
- Timer 0 Device ID: XPAR\_PS7\_TTC\_0\_DEVICE\_ID
- Timer 1 Device ID: XPAR\_PS7\_TTC\_1\_DEVICE\_ID
- Timer 0 Interrupt ID: XPAR\_XTTCPS\_0\_INTR
-

Task	Periode	Jitter	BCET	WCET	Priorität	SPP		SPNP	
						BCRT	WCRT	BCRT	WCRT
Task 1	2ms	0ms	0,5 ms	1,0 ms					
Task 2	1ms	0ms	0,1 ms	0,2 ms					
Task 3	10ms	0ms	0,2 ms	0,2 ms			4,0 ms		9,7 ms
Task 4	5ms	0ms	0,2 ms	1,0 ms					
Task 5	20ms	0ms	0,2 ms	0,5 ms			9,7 ms		8,3 ms

### Aufgabe 1:

Vergeben sie die Prioritäten der Tasks nach RMS! 0 ist die höchste Priorität.

### Aufgabe 2:

Berechnen sie die maximale Last auf dem Prozessorkern!

### Aufgabe 3:

Berechnen sie für das nun gegebene Taskset die BCRT sowohl für SPP als auch für SPNP Scheduling!

### Aufgabe 4:

Ermitteln sie für **Task 1** und **Task 2** die WCRT, sowohl für SPP als auch SPNP, **zeichnerisch!**

### Aufgabe 5:

Ermitteln sie für **Task 4** die WCRT, sowohl für SPP als auch für SPNP, **rechnerisch!**

### Aufgabe 6:

Angenommen die Aktivierung von **Task 1** hat einen **Jitter von 0,5ms**, wie wirkt sich dies bei **SPP** Scheduling auf die BCRT und WCRT von **Task 2** und **Task 4** aus?

Task	BCRT SPP	WCRT SPP
Task 2		
Task 4		


RMS: Rate Monotonic Scheduling	BCRT / WCRT: {Best / Worst} Case Response Time
SPP: Static Priority Preemptive	BCET / WCET: {Best / Worst} Case Execution Time
SPNP: Static Priority Non-Preemptive	

# Literaturverzeichnis

- [All] Allegro. *A4988, DMOS Microstepping Driver*. Verfügbar online unter [https://www.pololu.com/file/0J450/a4988\\_DMOS\\_microstepping\\_driver\\_with\\_translator.pdf](https://www.pololu.com/file/0J450/a4988_DMOS_microstepping_driver_with_translator.pdf); abgerufen am 22.4.2019. 30
- [arm] Infocenter von ARM. Verfügbar online unter <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka4141.html>; abgerufen am 22.4.2019. 25
- [con] University of Illinois System Website. Verfügbar online unter [https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3\\_Processes.html](https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html); abgerufen am 22.4.2019.
- [ech] Vorlesungsfolien der Uni Erlangen. Verfügbar online unter [http://www4.informatik.uni-erlangen.de/DE/Lehre/WS03/V\\_STP1/Skript/stp1-pa-ws03-kapitel4.pdf](http://www4.informatik.uni-erlangen.de/DE/Lehre/WS03/V_STP1/Skript/stp1-pa-ws03-kapitel4.pdf); abgerufen am 22.4.2019. 5
- [feSuK14] Fraunhofer-Institut für eingebettete Systeme und Kommunikationstechnik. Jahresbericht. 2014. Verfügbar online unter [https://www.esk.fraunhofer.de/content/dam/esk/dokumente/Jahresbericht\\_Fraunhofer\\_ESK\\_2013-2014.pdf](https://www.esk.fraunhofer.de/content/dam/esk/dokumente/Jahresbericht_Fraunhofer_ESK_2013-2014.pdf); abgerufen am 22.4.2019.
- [Gra] Philipp Graf. Verfügbar online unter <https://pdfs.semanticscholar.org/0aa7/b91e99d9749adbdca55286c31c0c990fe849.pdf>; abgerufen am 22.4.2019. 25
- [i2c] I2C Communication. Verfügbar online unter <https://www.totalphase.com/support/articles/200349156-I2C-Background>; abgerufen am 22.4.2019. 35
- [Inva] Invensense. *MPU9250 Datenblatt*. Verfügbar online unter <http://www.invensense.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf>; abgerufen am 22.4.2019. 35
- [Invb] Invensense. *MPU9250 Register Map*. Verfügbar online unter <http://www.invensense.com/wp-content/uploads/2017/11/RM-MPU-9250A-00-v1.6.pdf>; abgerufen am 22.4.2019. 35
- [jta] Corelis JTAG Interface and Boundary-Scan Educational Resources. Verfügbar online unter <https://www.corelis.com/education/tutorials/jtag-tutorial/what-is-jtag/>; abgerufen am 22.4.2019. 23

- [Koo] Phil Koopman. Better Embedded System SW: Blog to the Book: Better Embedded System Software. Verfügbar online unter <https://betterembsw.blogspot.com/2012/12/software-timing-loops.html>; abgerufen am 22.4.2019.
- [Lab02] Jean J Labrosse. *MicroC/OS-II: the real-time kernel*. Taylor & Francis US, 2002.
- [lau] Feature Overview Lauterbach Debugger. Verfügbar online unter <https://www.lauterbach.com/frames.html?home.html>; abgerufen am 10.10.2019. 3
- [Lau14] Lauterbach. Lauterbach iprobe user's guide. 2014. Verfügbar online unter [http://www2.lauterbach.com/pdf/iprobe\\_user.pdf](http://www2.lauterbach.com/pdf/iprobe_user.pdf); abgerufen am 22.4.2019. 35
- [mak] C-HowTo: Makefile. Verfügbar online unter <http://www.c-howto.de/tutorial/makefiles/>; abgerufen am 22.4.2019. 11
- [mbe] mbed.org, ARM, Timer and Interrupts. Verfügbar online unter [https://os.mbed.com/media/uploads/robt/mbed\\_course\\_notes\\_-\\_timers\\_and\\_interrupts.pdf](https://os.mbed.com/media/uploads/robt/mbed_course_notes_-_timers_and_interrupts.pdf); abgerufen am 22.4.2019.
- [mica] Micrium Documentation. Verfügbar online unter <https://doc.micrium.com/display/osiidoc>; abgerufen am 22.4.2019. 18
- [micb] Micrium Documentation zu Task Control Blocks. Verfügbar online unter <https://doc.micrium.com/pages/viewpage.action?pageId=16879778>; abgerufen am 22.4.2019. 18
- [micc] Micrium Documentation zu Message Queues. Verfügbar online unter <https://doc.micrium.com/display/osiidoc/Message+Queue+Management>; abgerufen am 22.4.2019. 18
- [micd] Micrium Documentation zu Kontextwechsel. Verfügbar online unter <https://doc.micrium.com/display/osiidoc/Context+Switching>; abgerufen am 22.4.2019.
- [mice] Micrium Documentation zu Message Queues. Verfügbar online unter <https://doc.micrium.com/display/osiidoc/Message+Queue+Management>; abgerufen am 23.4.2019.
- [Pie] Jan Pieter. Reading a IMU Without Kalman: The Complementary Filter. Verfügbar online unter <https://www.pieter-jan.com/node/11>; abgerufen am 22.4.2019. 35
- [RS14] Universität Mannheim Robert Schieler. Building and using a cross development tool chain. 2014. Verfügbar online unter <ftp://gcc.gnu.org/pub/gcc/>

- summit/2003/BuildingandUsingaCrossDevelopmentToolChain.pdf; abgerufen am 22.4.2019. 11
- [wik] Erstellung von Message Queues. Verfügbar online unter [https://wiki.eecs.yorku.ca/course\\_archive/2016-17/W/4352/\\_media/an1005\\_inter-process\\_communication\\_.pdf](https://wiki.eecs.yorku.ca/course_archive/2016-17/W/4352/_media/an1005_inter-process_communication_.pdf); abgerufen am 21.5.2019. 18
- [Xil18] Xilings. *Zynq-7000 SoC, Technical Reference Manual*, 2018. 30
- [zc7] Feature Overview Xilinx ZC706. Verfügbar online unter <https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html>; abgerufen am 10.10.2019. 3



Laurenz Borchers, Kai-Björn Gemlau  
Institut für Datentechnik und Kommunikationsnetze  
TU Braunschweig

