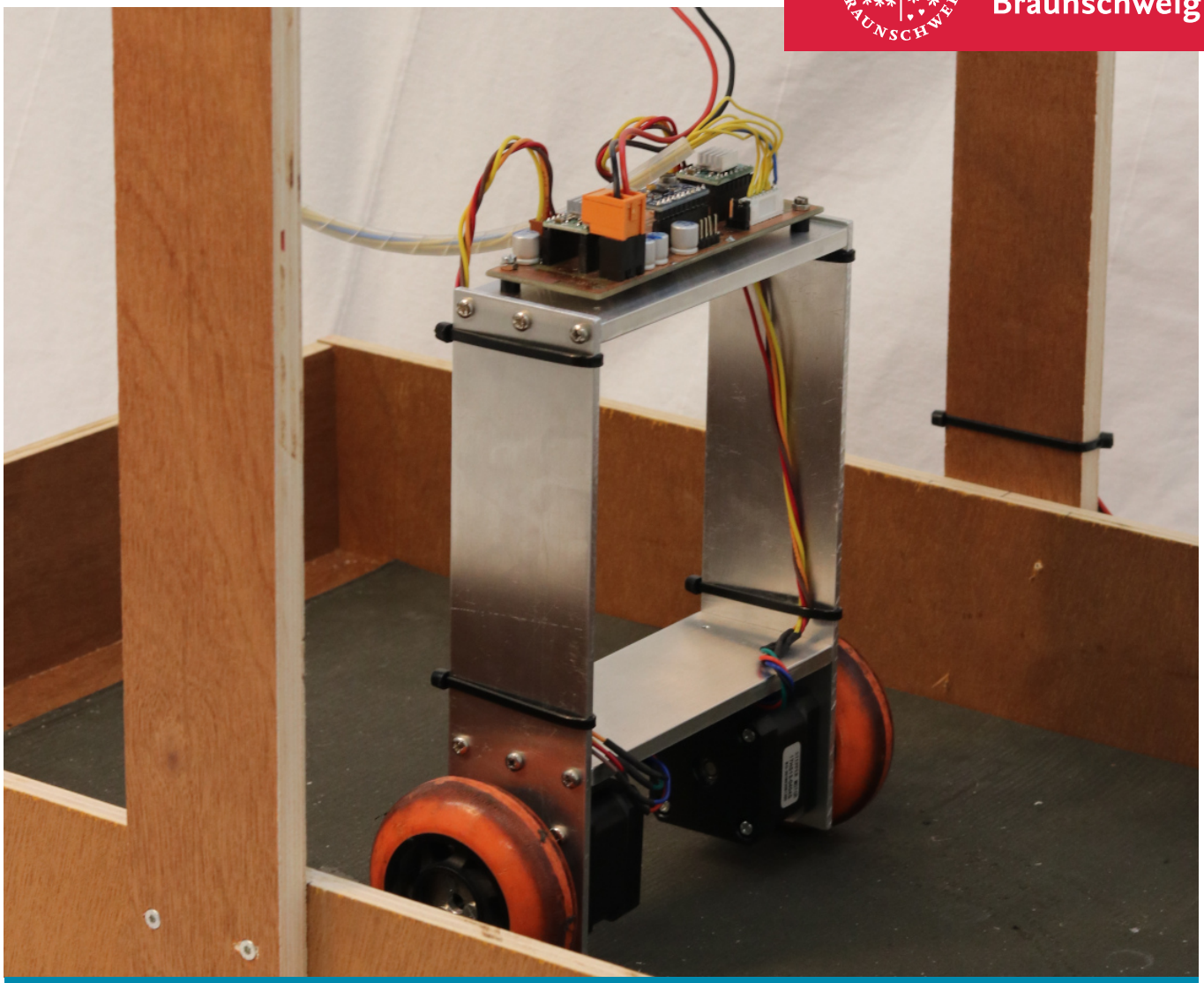
INSTITUTE OF COMPUTER AND NETWORK ENGINEERING

Technische Universität Braunschweig



Laborpraktikum: Software Debugging in eingebetteten Echtzeitsystemen

Skript

Laurenz Borchers, Kai-Björn Gemlau

12. April 2022

Inhaltsverzeichnis

1. Einleitung **1**
- 1.1. Motivation . . . 1
- 1.2. Lehrziel . . . 1
- 1.3. Anwendungsfall . . . 2

2. Hardware und Aufbau **3**
- 2.1. Hardware . . . 3

3. Grundlagenwissen **5**
- 3.1. Echtzeitsysteme . . . 5
- 3.2. Debugging . . . 5

4. Coding Guidelines **8**
- 4.1. Regeln . . . 8

5. Aufgabe 1 **11**
- 5.1. Wissen . . . 11
- 5.2. Pre-Kolloquium . . . 13
- 5.3. Aufgabenstellung . . . 14

6. Aufgabe 2 **18**
- 6.1. Wissen . . . 18
- 6.2. Aufgabenstellung . . . 19
- 6.3. Post-Kolloquium . . . 21

7. Aufgabe 3 **23**
- 7.1. Wissen . . . 23
- 7.2. Pre-Kolloquium . . . 30
- 7.3. Aufgabe . . . 31
- 7.4. Post-Kolloquium . . . 34

8. Aufgabe 4 **36**
- 8.1. Wissen . . . 36
- 8.2. Pre-Kolloquium . . . 37
- 8.3. Aufgabe . . . 38
- 8.4. Post-Kolloquium . . . 41

9. Aufgabe 5 **43**
9.1. Wissen 43
9.2. Aufgabe 45

10. Aufgabe 6 **47**
10.1. Tracing 47
10.2. Lauterbach-Wissen 51
10.3. Aufgabenteil 1 54
10.4. Aufgabenteil 2 56

11. Aufgabe 7 **58**
11.1. Aufgabenteil 1 58
11.2. Aufgabenteil 2 59

Akronyme **61**

A. Anhang **62**

Literaturverzeichnis **65**

1 Einleitung

1.1. Motivation

Im Programmieralltag wird man immer wieder damit konfrontiert, dass Software nicht einwandfrei funktioniert. Neben der Tatsache, dass es bei 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 entsprechende 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 inertiale 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].

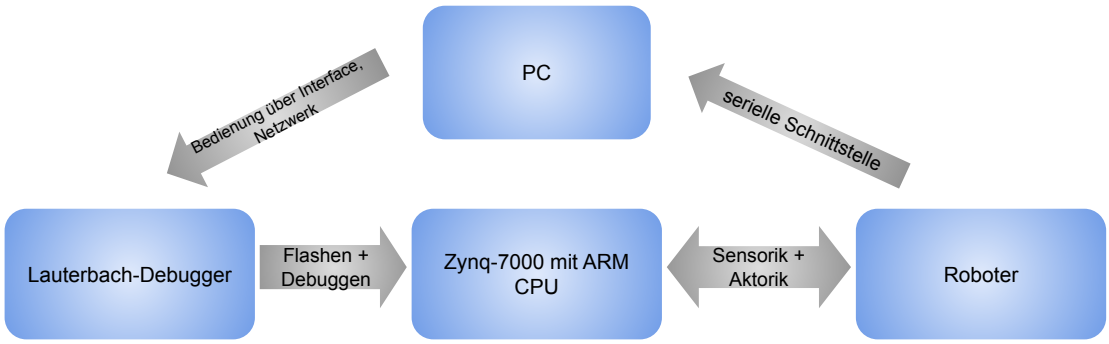


Abbildung 2.1.: Komponenten und Zusammenhänge des Versuchsaufbaus im Praktikum

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].

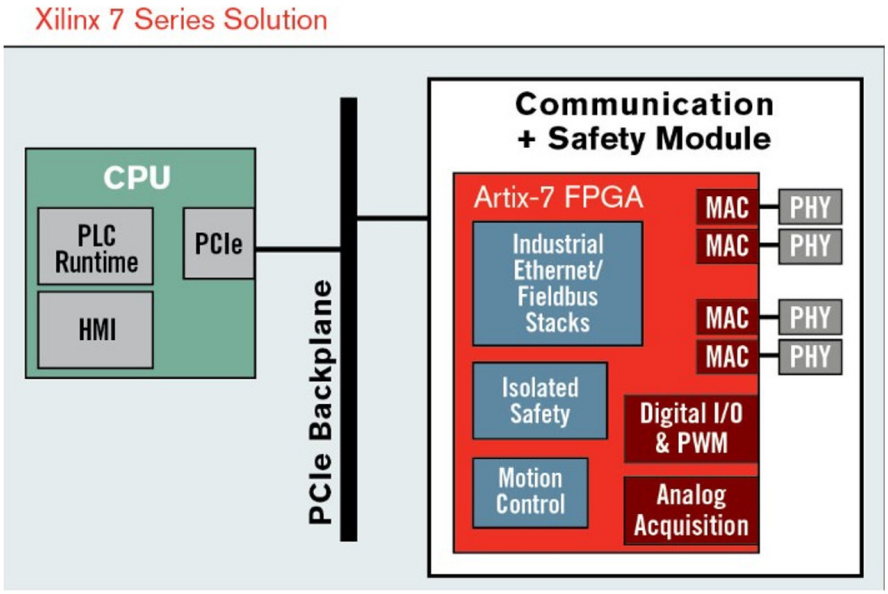


Abbildung 2.2.: Nutzung des Programmable Logic Controllers im Zynq7000 SoC

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 niederprioren 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.

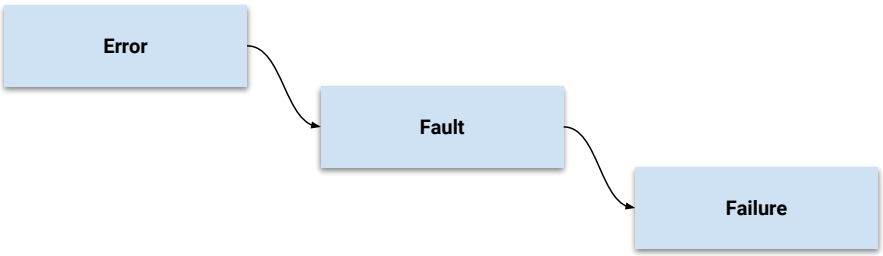


Abbildung 3.1.: Entwicklung eines Fehlers: Error->Fault->Failure

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 entscheident 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 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 XIicPs instance.
9 * @param MsgPtr is the pointer to the receive buffer.
```

```
* @param  ByteCount is the number of bytes to be received.
* @param  SlaveAddr is the address of the  slave  we are  receiving  from.
*
* @return None.
*
* @note   This receive  routine  is  for  interrupt-driven transfer  only.
*
*****************************************************************************/
void XIicPs_MasterRecv(XIicPs *InstancePtr, u8 *MsgPtr, s32 ByteCount,u16 SlaveAddr){
 ....
}
```

Quellcode 4.2: Beispielhaftes Doxygen Kommentar aus der Xilinx Libary

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.

```
/** Includes */
#include " .... h"

/** module input variables */
extern  int16_t  accData [3];

/** module output variables */
int32_t  pidValue=0;

/** module internal variables  */
 static   float  _pid_errorSumAngle=0;

/** module internal constatns */
const  int16_t  ACC_MIN[3] = {-10, -20, -30};

/** internal  function prototypes */
 static  int16_t  _pid_map(int16_t x,  int16_t  in_min, int16_t  in_max, int16_t  out_min, int16_t  out_max);
 ...
/** functions **/
void  pid_init(void){
 ...
}

 static  int16_t  _pid_map(int16_t x,  int16_t  in_min, int16_t  in_max, int16_t  out_min, int16_t  out_max){
 ...
}
```

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 Maschinensprache 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 des 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.

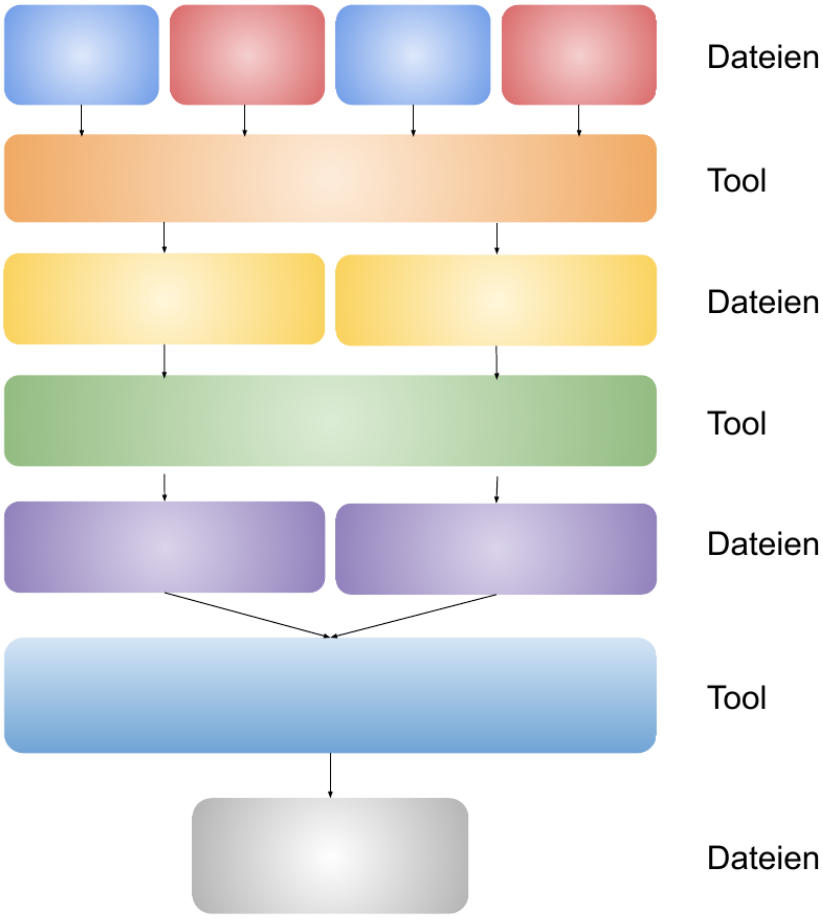


Abbildung 5.1.: Toolchain Diagramm zum Ausfüllen und zur Vorbereitung auf das Prekolloquium

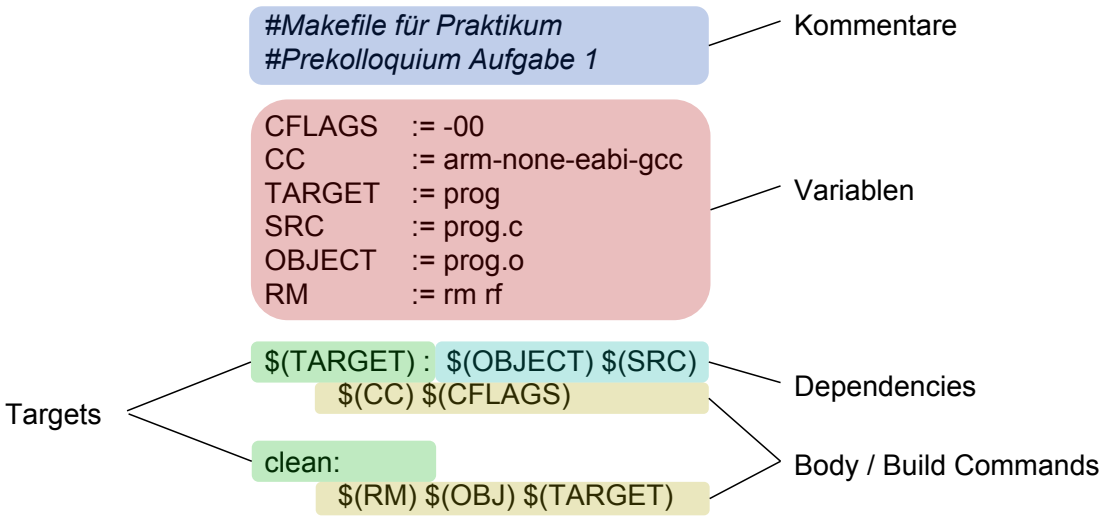


Abbildung 5.2.: Beispiel zum Aufbau einer Makefile

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 Schnittelle 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 sollen 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. Sie sollten erklären können welche Teile im Makefile welche Funktion erfüllen. In der folgenden Aufgabe werden Sie unterschiedliche Fehler im Code aber auch im gegeben Makefile finden müssen. Die Fehler im Makefile sollten Sie durch genaue Analyse des Makefiles bereits zum Teil finden können. Die doku.odt Datei enhält einige Ausschnitte des Makefiles die Sie genau beschreiben müssen. Machen Sie sich zudem mit den Linux Befehlen *find* und *grep* vertraut.

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

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

Dazu clonen Sie zunächst Ihren Fork mit dem Link aus dem Webinterface angezeigt in Ihren lokalen Ordner.

Zusätzlich sollten Sie einen neuen Branch in folgender Form anlegen:

```
git checkout -b praktikum
```

Es kommt vor, dass der Betreuer im Praktikumsverlauf Änderungen am Master vornehmen muss, dies führt häufig zu Konflikten. Durch den Branch werden diese vermieden.

Sie arbeiten in ihrem Fork auf dem angelegten Branch.

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.

Immer wenn eine neue Terminal-Session gestartet wird müssen zunächst einige Umgebungsvariablen gesetzt werden, damit die benötigten Tools gefunden werden. Dazu dienen hier die “setup-lm” Befehle, die die PATH-Variable um die gewünschten Tool-Verzeichnisse erweitern und bei Bedarf Lizenzserver zu setzen.

```
setup-lm lauterbach r_2020_09
setup-lm eclipse cdt-2020-06
setup-lm gcc gcc-arm-none-eabi-7-2018-q2
```

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

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

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

Achtung: Es kommt vor dass die vorherige Lauterbach-Session auf dem Debugger nicht ordnungsgemäß beendet wurde oder der Lauterbach von einem anderen Nutzer belegt war. Dann wirft der Befehl zunächst den Fehler ßelected device already in use by..."Dann bitte einmal prüfen ob wirklich der richtige Lauterbach angesprochen ist und nicht der einer andren Gruppe. Ein erneutes Ausführen dauert länger, sollte dann aber erfolgreich sein.

Nach dem Klonen des Gits ist die Entwicklungsumgebung einzurichten:

- Terminal in Verzeichnis der Wahl (aber nicht in dem Git Repositoryordner) öffnen
- 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:

```
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 dem 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:

```
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

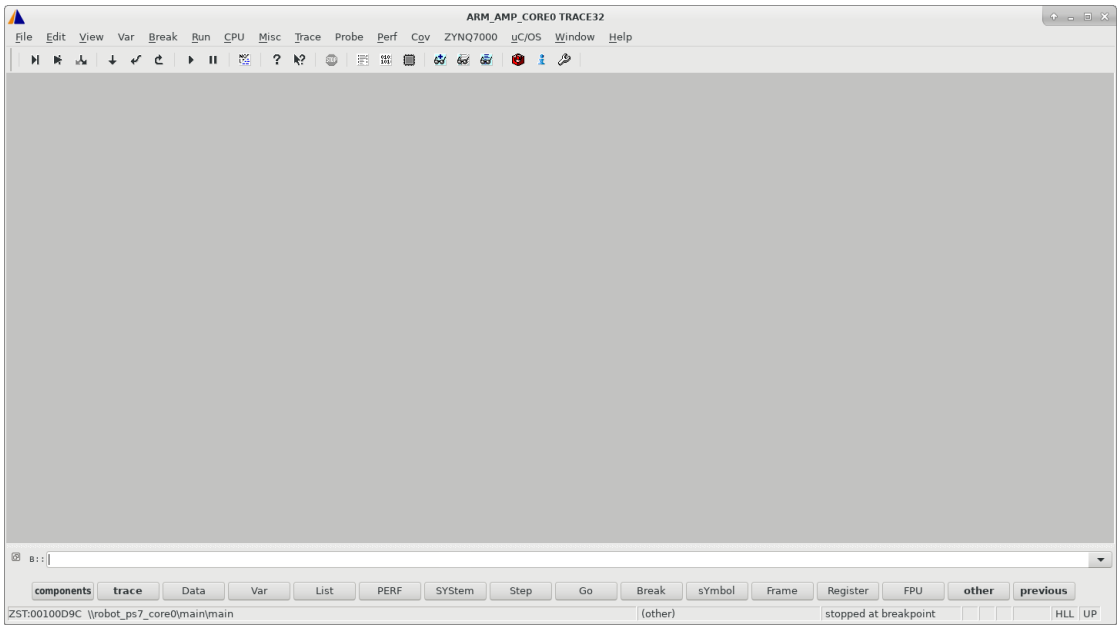


Abbildung 5.3.: Lauterbachumgebung zum Flashen des Boards in Aufgabe 1

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. Das Board sollte Ihnen jetzt jede Sekunde ein "I'LL BE BACK" ausgeben.

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

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 µC/OS-II wird ein Task mit der Methode

```
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

```
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:

- *µC/OS-II Micrium Documentation* zum Thema Task Management [mica, µC/OS-II Quick Reference, Task Management]
- *µC/OS-II Micrium Documentation* zum Thema Task Control Blocks [micb, µC/OS-II Quick Reference, Task Management]
- *µC/OS-II Micrium Documentation* zum Thema Message Queues [micc, µC/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*

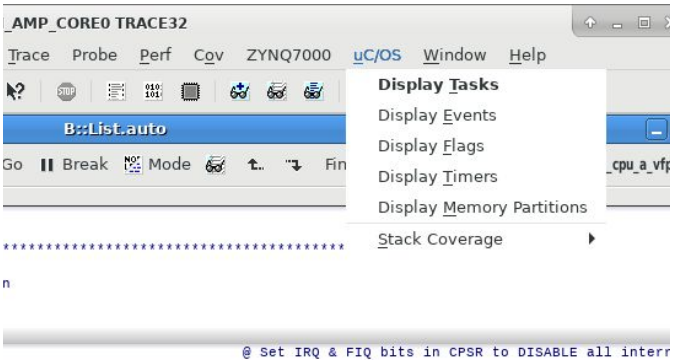


Abbildung 6.1.: Menü zum Inspizieren von Betriebssystemfunktionen

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 Frequnz 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.

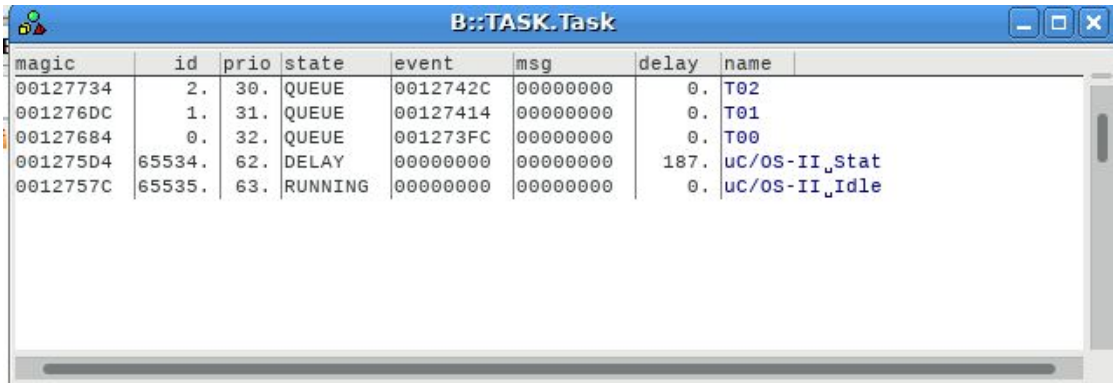


Abbildung 6.2.: Fenster zum Inspizieren der Stackausnutzung

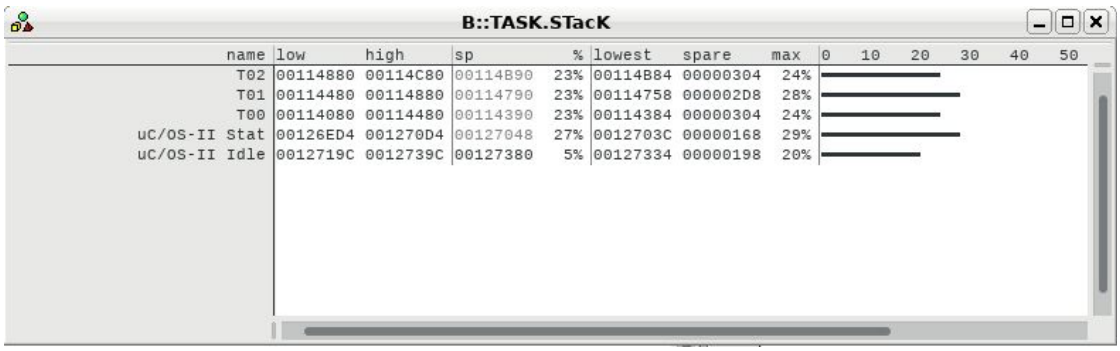


Abbildung 6.3.: Fenster zum Inspizieren von laufenden Tasks, ihren Prioritäten und IDs

- 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?

```
typedef struct os_tcb {

    OS_STK        *OSTCBStkPtr;

    void          *OSTCBExtPtr;

    OS_STK        *OSTCBStkBottom;
    INT32U         OSTCBStkSize;

    INT16U         OSTCBOpt;

    INT16U         OSTCBId;

    struct os_tcb *OSTCBNext;
    struct os_tcb *OSTCBPrev;

    OS_FLAGS       OSTCBFlagsRdy;

    INT8U          OSTCBStat;
    INT8U          OSTCBPrio;

} OS_TCB;
```

Quellcode 6.1: Ausschnitt aus dem Task Control Blocks in μC/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-

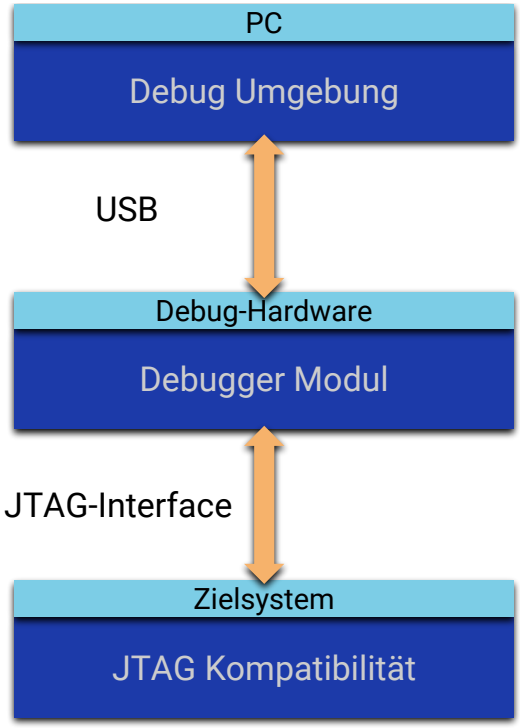


Abbildung 7.1.: Cross Debugging via Joint Test Action Group (JTAG)-Debugging

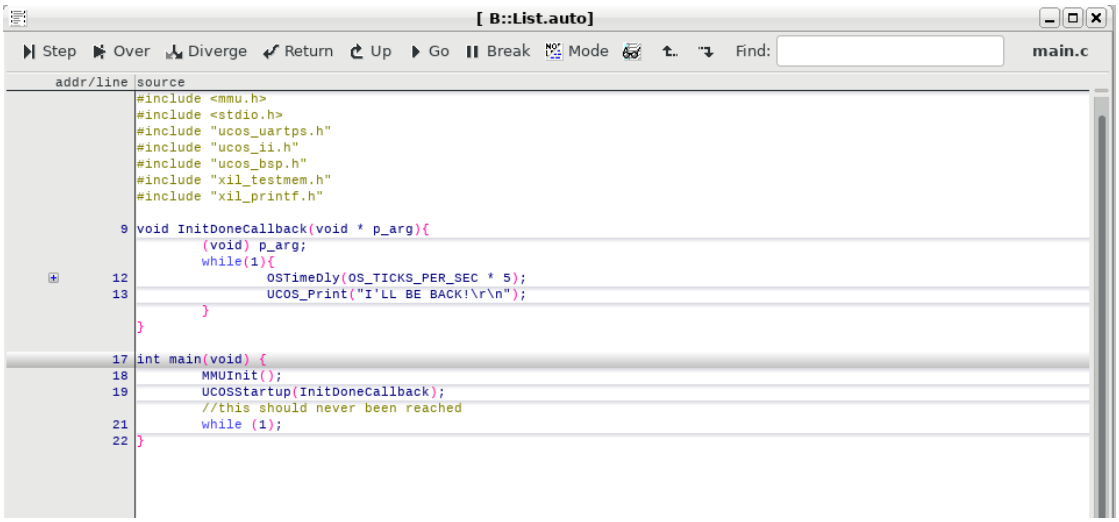


Abbildung 7.2.: Quelltextansicht in der Lauterbach Umgebung

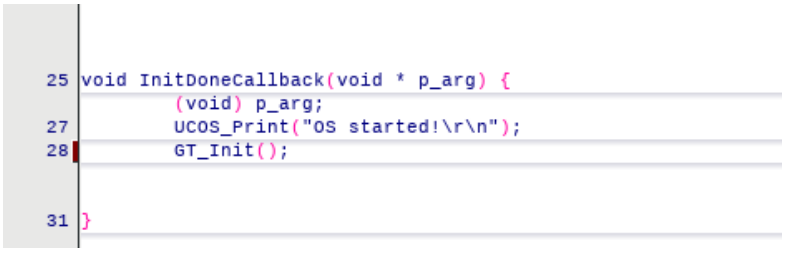


Abbildung 7.3.: Setzen eines Breakpoints in Zeile 28.

führende Unterroutine 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).

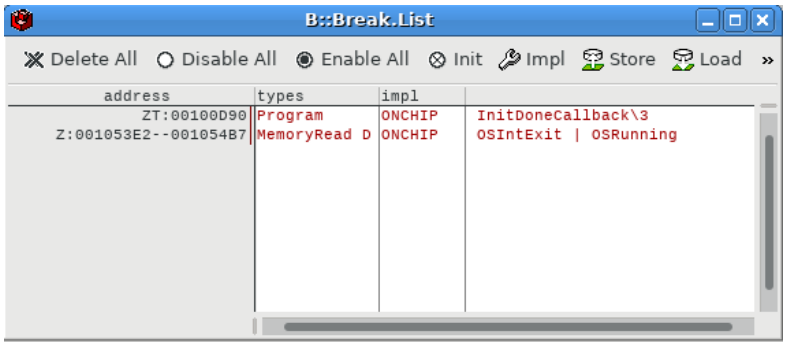


Abbildung 7.4.: Breakpoint-Übersichtsfenster in der Lauterbach Umgebung

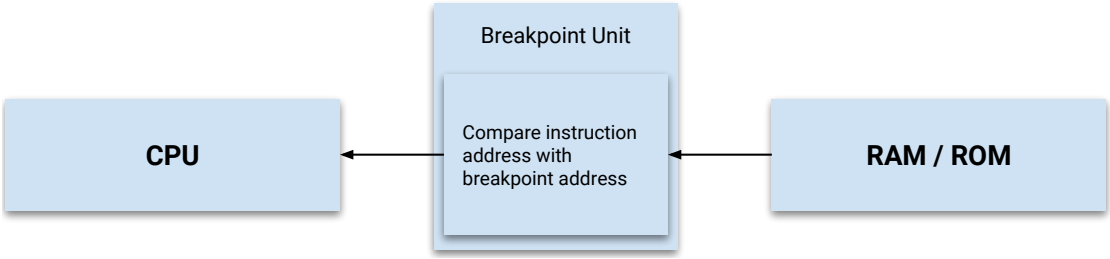


Abbildung 7.5.: Hardware Breakpoint Realisierung

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 spezifiert ist.

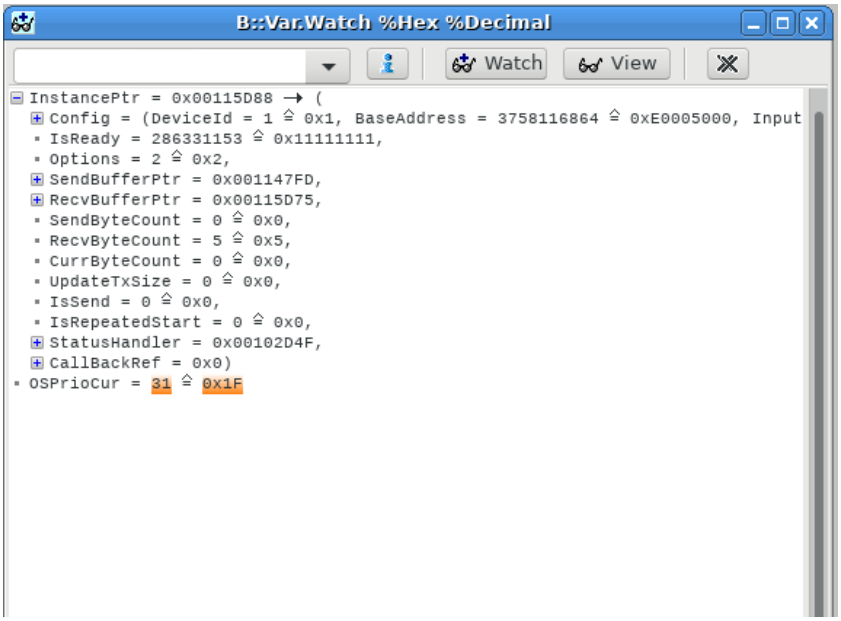


Abbildung 7.6.: Watch-Fenster Breakpoint Realisierung

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 Unterroutine 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

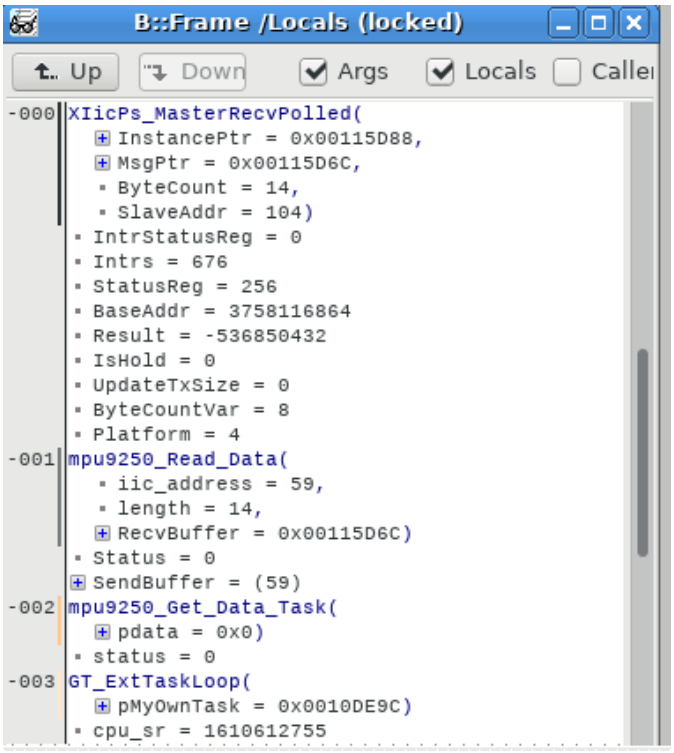


Abbildung 7.7.: Watch-Fenster Breakpoint Realisierung

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

```c
/*
** zur Veranschaulichung:
** alle Variablen global und im gleichen Speichersegment
*/

uint8_t numbers[4];
uint8_t a = 10;
uint8_t b = 15;
uint8_t result;

void main(void) {

  /*
  ** for-Schleife mit Fehler in Abbruchbedingung
  ** Schreiboperation auf numbers[4]
  ** -> fehlerhafte Daten in Speicherbereich der Variable a
  */
  for(uint8_t i = 0; i <= 4; i++){
    numbers[i] = i;
  }


  /* falsches Ergebnis durch fehlerhafte Daten in Variable a */
  result = a + b;

}
```

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 höherpriore 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 ansteuern 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. l

7.2. Pre-Kolloquium

Versuche 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 -1000 und 1000 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.

- Configs:
  - Ein Timer Tick entspricht einer Mikrosekunde
  - Nutzen Sie die Timer TTC0_0 und TTC0_1
  - Die GPIO-Pinnummern sind 54 und 55
- Simulieren Sie einen PID-Regler indem:
  - Sie einen globale Variable *pidValue* in *main.c* anlegen
  - Sie In der Methode *InitDoneCallback* eine Schleife erstellen, die die globale Variable in ihrem Wertebereich hoch zählt.
  - Tipp: Denken Sie daran, dass Sie die Variable nicht unendlich schnell hoch zählen.
- Initialisieren Sie einen Timer in der *ttc_timer.c*:
  - Erstellen Sie dafür eine init-Methode
  - Timer Control Struct definieren

```
typedef struct TimerStruct{
  u32 OutputHz;     /* Output frequency */
  XInterval Interval;    /* Interval value */
  u8 Prescaler;     /* Prescaler value */
  u16 Options;      /* Option settings */
} TmrCntrSetup;
```

  - Timer Instanz erstellen (global)

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

  - *TmrCntrSetup*-Struct erstellen (global)
  - Erstellen Sie eine Konfigurationsinstanz für den Timer

```
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

```
XTtcPs_LookupConfig(XPAR_PS7_TTC_0_DEVICE_ID);
```

- Den Timer initialisieren

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

- Die in dem TimerCntrSetup gewählten Optionen anwenden

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

- Den Prescaler auf Basis der gewünschten Frequenz automatisch herausfinden

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

- Prescaler setzen

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

- Match Value erstellen und setzen

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

- Intervalllänge erstellen und setzen

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

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

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

  - Erstellen Sie eine GPIO Instanz

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

- Implementieren Sie die Funktion

```
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 Richtung der Pins mit

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

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

```
XGpioPs_WritePin()
```

- Erstellen Sie eine Funktion *timer_Set_Interval_Length*, die die Interval-Länge sowie das Match-Value eines Timers setzt.
- Schreiben sie eine Task-Funktion *timer_Task*, die die Drehrichtung der Schrittmotoren sowie deren Geschwindigkeit in Abhängigkeit des pidValue setzt. Diese soll die Timer stoppen, die GPIOs korrekt setzen, die Intervall-Längen aus dem pidValue berechnen und dann die Timer wieder starten. Erstellen Sie in ihrer main.c einen Task, der zunächst die *timer_Init()* Funktion aufruft und dann alle 2ms die *timer_Task()* Funktion.
  - Da ihre Methode in Abhängigkeit zum PID Wert steht, und dieser in der *main.c* simuliert wird, bietet es sich an die globale Variable *pidValue* mit *extern* in ihre Datei einzubinden.
  - Überlegen Sie sich welche Konsequenzen nebenläufiger Schreib- oder Lesezugriff auf eine Variable haben kann und wie Sie diese Effekte verhindern können.
  - Tipp: Nutzen Sie Critical Sections beim Zugriff auf die globale pidValue Variable.

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?

1

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. Zusätzlich zum JTAG-Debugging wollen wir Ihnen einige Funktionen des Lauterbach-Debuggers näher bringen.

Damit der Roboter jederzeit seinen Winkel zur Horizontalen kennt, benötigt er eine echtzeitfähige Lagemessung. Der Treiber für die Ansteuerung der inertialen Messeinheit über I^2C 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 ausließt und in zwei globalen Variablen speichert. Maßgeblich für die Funktion des Moduls ist die korrekte Initialisierung des I^2C 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^2C 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^2C nötig. Dazu können Sie sich [i2c] ansehen. Es sollte bekannt sein, wie ein I^2C 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^2C. 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 realsiert [Pie].

8.1.2. Visualisierung von Daten

Der Lauterbach-Debbuger und die Trace32-Software bietet eine Vielzahl verschiedener Funktionen. Beim Arbeiten mit Sensoren, die mit hoher Frequenz mehrere Messwerte produzieren, kann es sehr hilfreich sein die Daten zu visualisieren.
Eine IMU ist ein solcher Sensor. Bei mehreren Achsen und wenigen zehn Hertz Messfrequenz ist die Ausgabe mit *printf* nahezu unbrauchbar, da der Entwickler kaum mit dem Lesen der Werte hinterher kommt. Ganz abgesehen von dem großen Einfluss von *printf* auf das Zeitverhalten.

Es bietet sich daher an die Daten graphisch in einem Koordinatensystem darzustellen. Der Lauterbach ermöglicht es auf Variablen zuzugreifen und diese zu visualisieren. Dazu speichert man in die Messwerte in seinem Programm über eine gewisse Zeit in einem Array ab. Anschließend hält man sein Programm durch den Debugger an.

Zum einen kann man nach dem Array in dem “Symbol.browse”-Fenster suchen, die Variable zur Watch-List hinzufügen und sich die Werte textlich anzeigen lassen.

Zum anderen kann man das Array plotten, indem man in die Lauterbach-Befehlszeile folgendes eingibt:

```
1 var.DRAW <NAME_OF_ARRAY>
```

Der Befehl ermöglicht es auch mehrere Arrays in ein Fenster zu plotten, um Zusammenhänge zwischen Daten besser zu verstehen:

```
1 var.DRAW <NAME_OF_ARRAY_1> <NAME_OF_ARRAY_2>
```

8.2. Pre-Kolloquium

- Wie liest man aus einem I^2C 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 ?
- Mit welcher Formel berechnet man den Winkel aus ACC und Gyroskop-Daten? Wo ist der Unterschied zwischen beiden? Wo spielt die Sampling-Zeit mit hinein?
- Werten Sie den gegebenen I^2C 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

In dieser Aufgabe sollen Sie folgenden Ablauf schrittweise implementieren:

- I^2C initialisieren
- IMU initialisieren
- IMU-Task starten
- Periodisch IMU-Daten abfragen
- Winkel aus Accelerometer- und Gyro-Daten berechnen
- Berechneten Winkel an PID Filter propagieren

IMU-Task erstellen

Die *imu.c* soll zwei Methoden nach außen bereitstellten. Diese lauten:

```
1 int mpu9250_Imu_Init(void *pdata);          // Initialize
2 int mpu9250_CalculateAngle(void *pdata);    //Task function
```

Erstellen Sie in der *main.c* einen neuen 5-Millisekunden-Task und starten Sie ihn so, wie in Aufgabe 2 gelernt. Dieser soll das IMU Modul zunächst initialisieren und dann in der while(1) Schleife die Berechnung des Winkels periodisch aufrufen. Füllen Sie in der weiteren Aufgabe diese beiden Methoden mit allem was zur Initialisierung und zur Berechnung benötigt wird aus.

I²C initialisieren

Erstellen Sie in der *imu.c* eine Methode zur Initialisierung der I²C Instanz:

```
static uint8_t mpu9250_Iic_Init();
```

- Zunächst die globale Variable für die I²C Instanz erstellen

```
static XIicPs Iic;
```

- Zur I²C Initialisierung: Config Struct erstellen

```
XIicPs_Config *Config;
```

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

```
Config = XIicPs_LookupConfig(...);
```

- I²C initialisieren und Status abfragen, übergeben Sie der Methode die benötigten Parameter (Tipp: Basisaddresse des I²C findet sich auch in dem Config-Struct)

```
XIicPs_CfgInitialize (...) ;
```

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

```
XIicPs_SelfTest (...) ;
```

- I²C Clockrate setzen, suchen Sie in der *imu.h* nach Übergabeparametern. Achtung: Die Clockrate soll 100kHz betragen.

```
XIicPs_SetSClk (...) ;
```

Das Nutzen der I²C 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*.

```
static uint8_t mpu9250_Write_Reg(uint8_t iic_address, uint8_t data)
static int8_t mpu9250_Read_Data(uint8_t iic_address, uint8_t length, u8 RecvBuffer [])
```

Die benötigte Schnittstelle des I²C Treibers von Xilinx sind die Methoden

```
XIicPs_MasterSendPolled()
```

und

```
1 XIicPs_MasterRecvPolled()
```

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

IMU initialisieren

Nach der erfolgreichen Initialisierung der I^2C Schnittstelle ist es notwendig die inertiale Messeinheit zu initialisieren. Nutzen Sie dafür die zuvor implementierten Send- und Recv Methoden, um die Register der IMU wie gewünscht zu konfigurieren. Die benötigten Register haben Sie im Präkolloquium ausgearbeitet. Zu überprüfen, ob die Initialisierung erfolgreich war, ist nicht zwingend notwendig, hilft im Zweifel aber Fehler zu finden. Hier kann der Status der IMU einmal abgefragt werden.

IMU-Daten abfragen

Nach dem Initialisieren der IMU sollen Sie nun 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.

Falls beim Auslesen Fehler auftreten, 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.

Wenden Sie das Wissen aus 8.1.2 an, um die Daten von Accelerometer und Gyroskop mit der Trace32-Software zu plotten. Schalten Sie den Lowpass Filter vom Accelerometer an und aus und vergleichen Sie die Qualität der Messwerte.

Nutzen Sie den Lauterbach Logik Analysator (Untermenü ‘Probe‘->‘Timing‘) um die I^2C Übertragung zu inspizieren. Verifizieren Sie die Rohwerte durch den Übungsleiter.

Winkel berechnen

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). Finden Sie zunächst die Maximal- und Minimalwerte für jede Achse bezüglich der Erdbeschleunigung heraus. Dies können Sie tun, indem Sie den Roboter langsam um jede Achse drehen dabei jede Messung mit den _imu_accMaxData und _imu_accMinData Arrays vergleichen und diese ggf. aktualisieren. Notieren Sie sich diese Messwerte und nutzen Sie sie im Folgenden als Konstanten für die Grenzen des ACC Wertebereiches. Mappen Sie anschließend die Daten der einzelnen Achsen des Accelerometers auf diesen Wertebereich. Die Gyroskopdaten müssen mit einem passenden Offset kalibiert 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. Zusätzlich benötigen Sie noch die Winkeländerung pro Zeitschritt.

Benutzen Sie die oben genannten Winkel und implementieren Sie einen Komplementärfilter, der die beiden Winkel zu einem fusioniert. Dieser sollte wie in etwa so aussehen:

```
1 currentAngle = 0.98*(previousAngle + gyroAnglePerTimeStep) + 0.02*accAngle;
```

In Aufgabe 7 wird Ihnen ein PID-Regler vorgegeben werden, damit die Motoren des Roboters adäquat auf den aktuellen Winkel reagieren. Die Kommunikation mit dem PID-Regler-Task läuft über eine globale Variable.

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.

9.1.1. Bedienung des Lauterbach-Debuggers

Die Trace32-Software enthält viele Komfort-Funktionen, die das Debugging aber auch die Bedienung der Software erleichtern sollen.

Window

Die vielen Fenster können schnell unübersichtlich werden und überfordern Einsteiger häufig. Daher können die Fenster automatisch angeordnet werden:
Window->Cascade oder *Window->Tile*

Anpassungen die während des Betriebs an den Fenstern vorgenommen wurden sind, können in verschiedenen Konfigurationen gespeichert und geladen werden:
Window->Store Windows to... oder *Window->Load Windows from...*

Weitere Fenster können über die Lauterbach-Befehlszeile geöffnet werden, indem die Caption des Fensters eingegeben wird. Viele der unter der Befehlszeile gelisteten Befehle sind auch Fenster und sind je nach Anwendungszweck unterschiedlich nützlich.

So kann sich der Befehlsverlauf mit

```
1 History
```

angezeigt werden lassen und eine Übersicht aller Tasks findet sich mit

```
1 Task.task
```

Diese ist insbesondere dann hilfreich, wenn man sich mit Kontext-Wechsel beschäftigt. Dies setzt aber voraus, dass die Tasks benannt wurden sind. Dazu benutzt man:

```
1 OSTaskNameSet(TASK_PRIORITY, TASK_NAME, &ERROR_CODE)
```

Break- und Watchpoints

Wie schon bei den Fenster-Einstellungen kann Trace32 auch Breakpoints permanent speichern, sodass eine Debug-Session unterbrochen und am nächsten Arbeitstag fortgesetzt werden kann. Das Speichern erreicht man mit:

```
STORE <filename>.cmm BREAK
```

Zum Laden benutzt man:

```
do <filename>.cmm
```

Um Watchpoints auf eine Variable zu setzen gibt es den Befehl

```
Var.Break.Set  <variable_name>; /<access>  /VarCONDition <condition>
```

- *<variable_name>* ist mit der Variable zu ersetzen, die man überwachen möchte
- *<access>* ist der Zugriff: *ReadWrite*, *Read*, *Write*
- *<condition>* ist eine Bedingung in C-Style; zum Beispiel: (a == 3). Kann aber auch weggelassen werden, falls bei jedem Zugriff getriggered werden soll.

Damit lassen sich auch schwer erreichbare Stellen debuggen. So kann man in den Kontext-Wechsel eines bestimmten Tasks springen, indem man auf *OSPrioHighRdy* triggered und als Bedingung die Priorität des aktuellen Tasks angibt:

```
Var.Break.Set  OSPrioHighRdy; /Write /VarCONDition (OSPrioCur==<priority>)
```

Auch interessant ist es am Ende einer Funktion anzuhalten, zum Beispiel dann wenn der Rücksprung nicht mehr funktioniert.

```
break.set  Symbol.end(<function_name>)
```

Ergebnisse speichern

Die Software besitzt auch Funktionen, die die Dokumentation der Ergebnisse erleichtert. So können Screenshots des übergeordneten Parent-Window gemacht werden: *Window->Screenshot to file...*

Genauso wie Screenshots eines einzelner, untergeordneter Fensters gemacht werden können. Dazu klickt man in die obere, linke Ecke des Fensters auf sein Icon und wählt *Window Screenshot to file...*

In diesem Menü kann der Fenster-Inhalt auch mit *To Clipboard* in Textform gespeichert werden.

9.2. Aufgabe

- Ziel soll es zunächst sein, die Methode

```
UCOSStartup()
```

  zu untersuchen.

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

- Was macht die Methode:

```
CPU_Init()
```

- Welche Aufgabe hat die Funktion:

```
Mem_Init()
```

  Muss diese Funktion aufgerufen werden?

- Betrachten Sie die Funktion

```
OSInit()
```

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

```
OS_InitMisc()
OS_InitRdyList()
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

```
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.
(Hardware-)Tracing lässt sich aber auch einsetzen, um den zeitlichen Ablauf des Programms zu analysieren und so zum Beispiel Statistiken über die Worst-Case Execution Time (WCET) anzustellen.

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.
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.

Abbildung 10.1.: Lauterbach Trace Debugger PowerTrace-II

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 abgenommen. Die übliche Methode bei Mikroprozessoren ist das Auslesen des Adressbusses zum Speicher und einiger Steuersignale. Mithilfe dieser Daten kann der Programmablauf 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, dass 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,

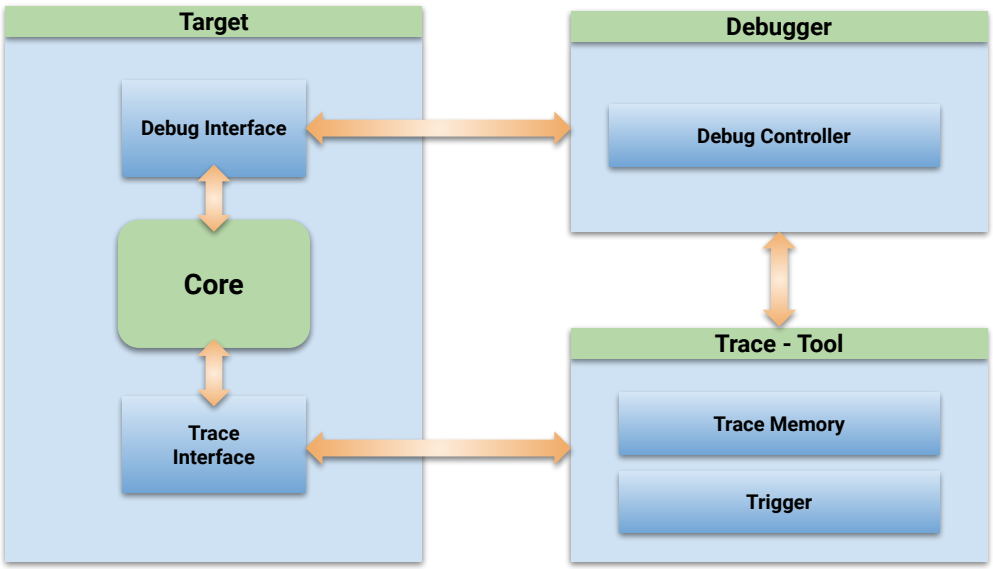


Abbildung 10.2.: Funktionsweise Offchip Tracing

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 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 Werte 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 höherpriore 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 SoftwareTracing nicht betroffen sind, haben sie Nachteile. Dazu gehört der sehr hohe Kaufpreis solcher Systeme, der sich im niedrigen fünfstelligem Bereich bewegt. Außerdem ist die Unterstützung von Tracing unter

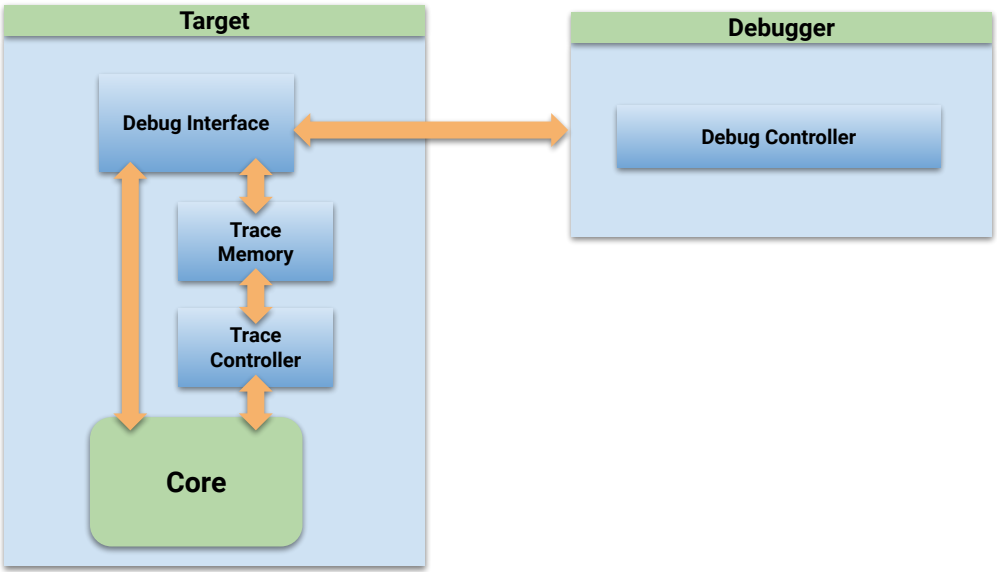


Abbildung 10.3.: Funktionsweise Onchip Tracing

den Entwicklungsboards wesentlich weniger weit verbreitet, als das bei JTAG-Debugging der Fall ist. Das Interface benötigt aufgrund der hohen Datenraten bei der Übertragung 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 verschieden 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. Lauterbach-Wissen

10.2.1. Tracing Points

Das Aufzeichnen des gesamten Programmflusses ist oftmals nicht notwendig und meistens eher störend beim Fehler finden, da die wichtigen Details in der Menge an Informationen untergehen. Daher ist es möglich das Tracing erst bei Bedarf zu aktivieren. Um das Handling zu erleichtern, ist die Syntax bei Lauterbach zwischen Break-Points und Tracing-Points sehr ähnlich (vgl. 9.1.1):

```
break.set  <function_name> /program /TraceEnable
```

TraceEnable aktiviert das Tracing für einen kurzen Moment beim eintreten der Bedingung oder des Events. Es erzeugt so zu sagen ein Snapshot. Dies ermöglicht es auch den Zugriff auf eine Variable zu tracen und den Rest des Systems auszublenden:

```
var.break.set  <variable_name> /<access> /TraceEnable
```

Leider stehen hardware-bedingt nur 4 TraceEnable-Points zur Verfügung. Daher greift man auf Trace-On/-Off-Points zurück. Beim Erreichen eines Trace-On-Points wird das Tracing aktiviert und wieder beendet beim Erreichen eines Trace-Off-Points.

```
break.set  <function_name_1> /program /TraceON
break.set  <function_name_2> /program /TraceOFF
```

Das Setzen von Tracepoints ist auch über die GUI möglich z.B. indem man im Sourcecode ein Rechtsklick macht und *Breakpoints->TraceEnable/-On/-Off* auswählt.

Ebenso wie bei den Breakpoints ist die Verwendung des *symbol*-Befehls möglich [tra, S. 286]

10.2.2. Darstellung

Die Darstellung der aufgezeichneten Daten übernimmt die Trace32 Software und kann angezeigt werden mit (Abbildung 10.5):

```
trace.chart
```

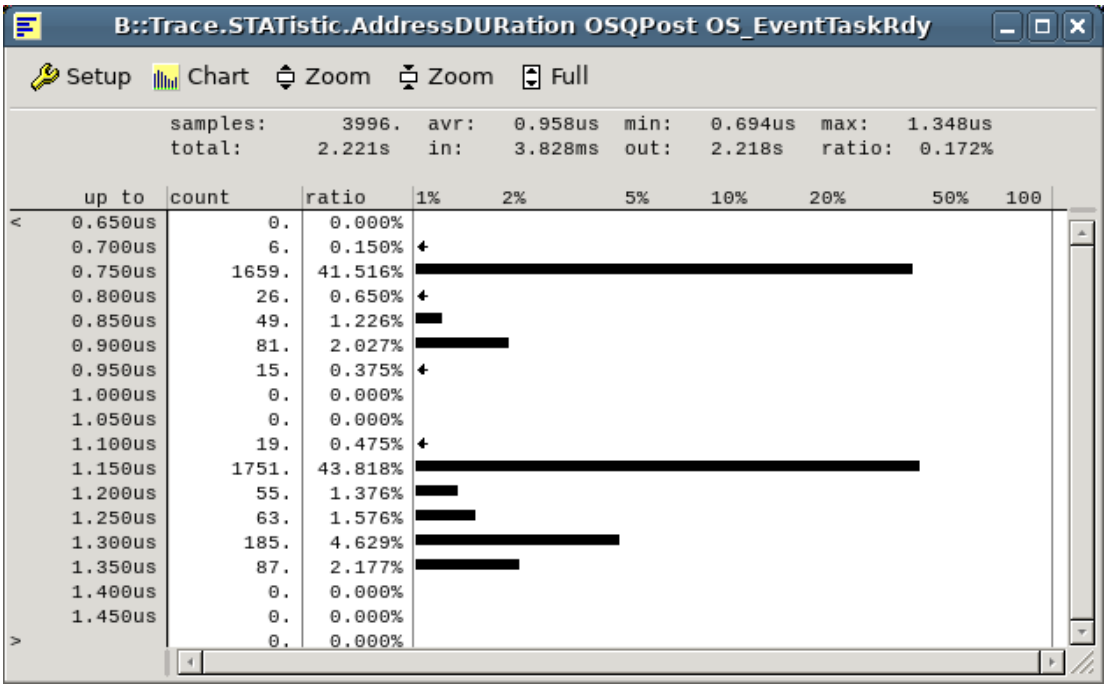


Abbildung 10.4.: Verteilung dargestellt
Das Diagramm zeigt die Dauer zwischen *OSQPost* und *OS_EventTaskRdy*.

Alternativ kann über die GUI gearbeitet werden mit *Trace->Chart->Symbols*.

Je nach Dauer und Menge der Daten kann die Darstellung etwas dauern.
Zudem kann Lauterbach die Verteilung bestimmter Werte in einer Statistik darstellen. Der folgende Befehl erzeugt ein Diagramm, dass die Verteilung der Dauer zwischen den beiden angegebenen Methoden darstellt (Abbildung 10.4).

```
1 Trace.STATistic.AddressDURation <function_name_1> <function_name_2>
```

Falls man erneut Tracen möchte, sollte man die gespeicherten Daten zurücksetzen mit:

```
1 Trace.Init
```

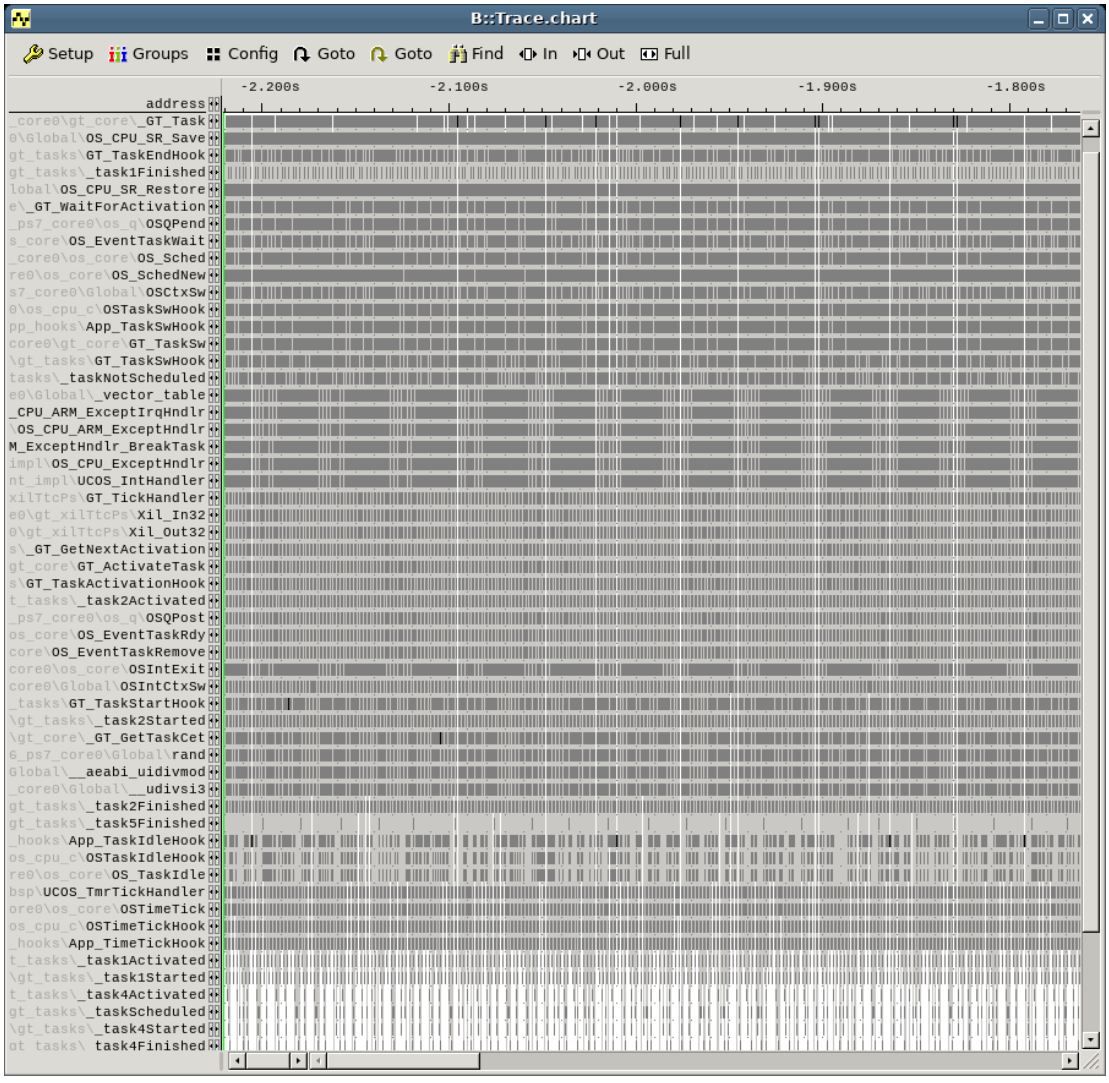


Abbildung 10.5.: Trace im Chart dargestellt

Ein langer und großer Trace, der zwar viele Daten enthält. Aber die wesentlichen Punkte sind nicht sofort ersichtlich.

10.3. Aufgabenteil 1

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?

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 Scheudling!

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, für SPP **rechnerisch** und für SPNP **zeichnerisch**!

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	

10.4. Aufgabenteil 2

Kompilieren Sie nun die *Aufgabe 6* und laden diese auf das Board. In dieser Aufgabe wird ein Tasksetsimulator genutzt, indem sich mehrere Tasks nach dem P-J-D Modell konfigurieren lassen. Tragen Sie zunächst die korrekten Prioritäten der Tasks in der Datei `src/gt_tasks.c` ein.

Untersuchen Sie zunächst mit Hilfe der Tracingtechniken des Lauterbachs die Kalibrierung des Tasksetsimulators. Der Simulator nutzt eine Schleife in der Funktion

```
1 __burn_wcet(CET,GT_CPU_OS_TASK_OFFSET,GT_CPU_CYCLE_SCALE);
```

um die $BCET \leq CET \leq WCET$ zu simulieren. Das heißt der Task hat effektiv keine Funktion sondern verbraucht nur die Rechenzeit CET. Die CET wird dabei für jede Aktivierung zufällig zwischen der $BCET$ und $WCET$ gewählt. Um die Ausführungszeiten des Simulators auf den Prozessor anzupassen müssen bestimmte Parameter korrekt gesetzt werden, da ansonsten die CET nicht den Sollwerten entsprechen. Ihre Aufgabe ist es zunächst diese Parameter so zu konfigurieren, dass die Ergebnisse den Sollwerten entsprechen. Die Parameter unterteilen sich in einen Scale-Wert und einen Offset. Der Scale-Parameter garantiert die korrekte Ausführungszeit langer Delays. Der Offset muss korrekt gesetzt werden, damit kurze Delays genau genug für eine Simulation des Taskssets sind. Zu Beginn der Main-Funktion wird eine Methode zum Kalibrieren der Parameter aufgerufen. Der Kommentar zu der Methode erklärt die Funtionsweise. Messen Sie die Ausführungszeiten der von dem Taskset genutzen Funktion in der Methode `GT_calibrate` indem Sie Tracepunkte nutzen. Die CET wird dabei exponentiell von 0,1ms auf 51,2ms erhöht. Die Zeiten zwischen den Trace-Events können sie in der trace.list Darstellung ablesen. Nutzen Sie zunächst weiter das Onchip-Tracing (*zc706_onchip_trace.cmm*), da bei der Kalibrierung ja nur wenige Tracedaten generiert werden.

Erstellen Sie sich eine Tabelle mit Soll- und Ist-Werten für die einzelnen Schleifen-Durchläufe in der Kalibrierungsfunktion. Nun können Sie das Verhältnis zwischen Soll- und Ist-Werten ausrechnen und in einem Diagram darstellen lassen. Stellen Sie die Parameter so ein, dass $0,96 \leq IST/SOLL \leq 1,0$ gilt.

Tipp: Passen Sie ihr .cmm Skript für Aufgabe 6 an, um wiederkehrende Arbeitsschritte wie das Setzen von Break-/ Tracepunkten zu automatisieren!

Ist die Kalibrierung geglückt sollen Sie das Taskset aus der theorethischen Aufgabe in den Simulator übertragen und visualisieren. Der Taskssetsimulator erlaubt es mehrere Tasks nach dem Periode-Jitter Modell zu konfigurieren. Diese sind in der Datei *APP/Aufgabe6/src/gt_tasks.c* bereits mit den $BCET$ und $WCET$ Parametern aus dem Aufgabenblatt definiert.

Nach der Bearbeitung des Arbeitsblattes und der Kalibrierung des Simulators sollen Sie nun untersuchen, wie sich das vorher berechnete Zeitverhalten auf einem realen System verhält. Nutzen Sie im Folgenden OffChip-Tracing, indem Sie die *zc706_offchip_trace.cmm* ausführen.

Vergleichen Sie die *zc706_offchip_trace.cmm* und *zc706_onchip_trace.cmm*. Wo liegen die Unterschiede? Tipp: Tools wie *meld* oder *vimdiff* erleichtern die Arbeit.

In der Datei finden sich auch die Methoden

```
1 GT_TaskActivationHook
2 GT_TaskStartHook
3 GT_TaskEndHook
4 GT_TaskSwHook
```

Mit ihnen lassen sich Aktionen auslösen, wenn ein Task bereit ist, gestartet, gescheduled oder beendet wird.

- Erweitern Sie diese Methoden so, dass sie unterscheiden können, welcher Task aktiviert bzw. beendet wurde.
- Benutzen Sie zu erst Trace-Enable Points, um das Scheduling eines einzelnen Tasks aufzuzeichnen (Activated, Scheduled, Not-Scheduled, Finished).
- Benutzen Sie anschließend Trace-On/-Off Points, um das Verhalten aller Tasks zu untersuchen.
- Erstellen Sie eine Statistik über die Verteilung der WCET und der WCRT jedes Tasks.
- Vergessen Sie nicht ihre Ergebnisse zu dokumentieren z.B. mit Screenshots oder als Text-Export.

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:

- Aktuellen Winkel der IMU
- Sample Time

Funktion

- PID Wert erstellen

Ausgänge

- PID Wert zwischen -1000 und 1000

11.1.2. IMU

Funktion

- IMU initialisieren
- Werte über I2C auslesen

Ausgänge

- IMU Winkel

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 800us/step, max 100us /step

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

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

Quellcode 11.1: Ergänzung Task Timing Struct

hinzu. Außerdem sollen Sie in dem Struct *GT_Tasks* die Zeile

```
{ 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 11.2: 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.

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.

```
UCOS_IntVectSet(...);
UCOS_IntSrcEn (...);
```

Die Interrupts werden nur dann ausgelöst, wenn der Timer entsprechend konfiguriert wird. Tipp: Nutzen Sie *XTTCPS_IXR_INTERVAL_MASK*

```
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.

```
u32 StatusEvent;
StatusEvent = XTtcPs_GetInterruptStatus (...) ;
XTtcPs_ClearInterruptStatus (...) ;
```

Akronyme

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

Literaturverzeichnis

[All] Allegro. *A4988, DMOS Microstepping Driver*. Verfügbar online unter 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; 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; 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; 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. 37

[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. 37

[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. 37

[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; abgerufen am 22.4.2019. 37

[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; 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/osiiidoc/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. 37

[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

[tra] General Function Reference. Verfügbar online unter https://www2.lauterbach.com/pdf/general_func.pdf; abgerufen am 25.05.2020. 51

[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; 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