Projektarbeit: Raspberrey PI

Modul: Grundlagen der technischen Informatik

Team 3

Lukas Bai
Kenneth Joss

Grundlagen der technischen Informatik - Projektarbeit Raspberry Pi

Team 3:
Lukas Bai
Kenneth Joss

1. Inhaltsverzeichnis

2. Vorbereitung

2.1. Aufgabenstellung

Die Aufgabenstellung umfasst folgende Punkte:

- Erweitern Sie den PI um eine LED-Komponente, d.h. schliessen Sie eine LED an einem GPIO-Port des PI an.
- Implementieren Sie eine Assembler-Routine (Bare-Metal-Ansatz), welche die LED ansteuert und damit ein SOS-Signal regelmässig wiederkehrend erzeugt.
- Ausgabe: blinken beliebiger, voreinstellbarer Morse-Signale
- Es dürfen KEINE Assembler-Macros eingesetzt werden (auch nicht selbsterstellte), sondern entsprechender Code soll in Sub-Routinen programmiert werden. Einzige Ausnahme: Das Macro IMM32.
- Zur besseren Gestaltung und Übersicht ist das Auslagern in Include-Dateien erlaubt, allenfalls sogar sehr sinnvoll.

Detaillierte Aufgabenstellung auf Moodle: https://moodle.ffhs.ch/mod/book/view.php?id=4172501

2.2. Absicht und Ziele

Moderne Computer und Rechner arbeiten weitgehend nach den Prinzipien und Strukturen einst definiert bei einer Gruppe von Pionieren. Seit den 1980er Jahren ist die Entwicklung rasant vorwärtsgeschritten und die funktionalen Kernelemente sind mehr und mehr verborgen unter einer dicken Schicht (Layer) von Treibern, Applikationen und User-Interfaces. Damit wird der Blick auf das Wesentliche drastisch eingeschränkt.

Im Unterricht GTI wollen wir den Fokus genau auf die Kernelemente der Computer-Technik richten und daher ist es notwendig, den ungetrübten Blick auf die CPU zu erlangen. Mit dem Raspberry PI besteht eine Möglichkeit, dies in der Praxis umzusetzen und wir wollen im Rahmen einer Projektarbeit die theoretischen Konzepte aus der Literatur am praktischen, "lebendigen" Objekt nachvollziehen.

2.3. Lieferobjekte

Sourcecode, Skripts und Dokumentation, um die Anwendung zu generieren, installieren und zu starten. Demo der Anwendung mit einem Raspberry PI und angeschlossenem GIO-Modul mit LEDs.

2.4. Methodisches vorgehen (Planung und Arbeitsfortschritt)

- Projekt definieren
- Toolchain installieren und Template Projekt Kompilieren
- Template Projekt auf Raspberry PI booten
- Benötigte Details dokumentieren und testen (Programm-Flow, aufrufen von Sub Routinen, ansteuern der LED, Timing ...)
- Programm Ablauf als Programmablaufplan zeichnen
- Programm umsetzen und testen
- Dokumentation erstellen und ergänzen

2.5. Roadmap: Milestones und Delivery

- M1: Abgabe initiales Dokument bis PVA-2
- M2: Erste Boot bare Version des Template Projekts bis [TBD]
- M3: Austesten und dokumentieren aller benötigten Funktionen bis [TBD]
- M4: Funktionierendes Programm bis [TBD]
- M5: Präsentation und Bericht (definitives Dokument) und ausgetestetes Programm bis PVA-5

2.6. Tool Entscheid

Entscheid:

GNU ARM Toolchain

Begründung:

- Plattform unabhängig: Sie kann unter Windows / Linux / OSX eingesetzt werden. Im Team verwenden wir Windows & Linux Rechner.
- Open Source & Community: Mit dem offenen Quellcode und der dahinterstehenden Community, ist es wahrscheinlicher Probleme schnell beheben zu können oder nötige Hilfestellungen im Internet (z.B. Stack Overflow) zu finden.
- Installation: Kann unter Linux einfach installiert werden (via APT). Auch für Windows kann mit dem WSL2 (Windows Subsystem for Linux 2) die Toolchain mit einem einzigen Kommando installiert werden («sudo apt install gcc-aarch64-linux-gnu»).
- Es kann auch auf dem Raspberry PI entwickelt verwendet werden

2.7. Fragen

Was ist «Bare Metal Programmierung»?

Programmierung direkt an der Hardware ohne Betriebssystem und ohne höhere Programmiersprache und ohne Funktionsbibliotheken. Das heisst der Code wird direkt in Assembler geschrieben. Es gibt keine Syscalls und das Produkt (Programm) ist bootfähig und wird vom BIOS beim Systemstart ausgeführt.

Wie kann dies mit dem ARMv8 gemacht werden?

Wir schreiben das Programm in Assembler auf einem Beliebigem Computer mit Linux, Windows oder MacOS, erstellen das Programm und speichern es auf einer bootfähigen SD-Karte. Oder wir senden das Programm über die Serielle Schnittstelle an den Raspberry PI.
(https://blog.nicolasmesa.co/posts/2019/08/booting-your-own-kernel-on-raspberry-pi-via-uart/)

Mit der «GNU ARM Toolchain» haben wir die Möglichkeit unseren eigenen Boot fähigen Kernel zu Programmieren. Wie es in der Aufgabenstellung verlangt wird können wir das Programm (Kernel) Assembler schreiben.

Was geht nicht?

Ein Kernel in Assembler zu schreiben ist sehr zeitaufwändig. Das Produkt kann deshalb in der Funktionalität nicht zu aufwändig sein. Kompromisse müssen eingegangen werden.

Das Debugging ist sehr erschwert, weil direktes Feedback nicht möglich ist. Es ist zum Beispiel nicht möglich einfach Text auf der Konsole auszugeben.

In einem Emulator zu arbeiten (wie «*qemu*») ist zwar möglich, aber nur limitiert sinnvoll, weil nicht alle Hardware unterstützt wird. Das Ansteuern der LEDs kann nicht oder nur mit grossem Aufwand emuliert werden.

Wie sieht der Prozess zur Erstellung eines Assembler-Programms und eines Boot-Images beim ARMv8 aus?

Der Assembler code wird mit einem Text Editor geschrieben. Danach wird der Source Code kompiliert und verlinkt. Siehe «Welche Tools werden, dazu benötigt (vgl. «Tool Chain») und in welcher Reihenfolge?». Dieser Prozess wird durch ein GNU – Makefile automatisiert.

Als tempelte Projekt dienen uns diese Dateien: https://moodle.ffhs.ch/mod/folder/view.php?id=4172510

Welche Tools werden dazu benötigt (vgl. «Tool Chain») und in welcher Reihenfolge?

1. «aarch64-linux-gnu-as»
Assembler zum Erstellen von *.o (Object) Dateien aus dem Sourcecode. Sourcecode ist als Text in Dateien mit der Dateiendung *.s gespeichert.

2. «aarch64-linux-gnu-ld»
Der Linker verlinkt die *.o (Object) Dateien zum fertigen Programm. Er erstellt eine Datei mit der Endung elf. Elf steht für «Executable and Linking Format» (https://www.man7.org/linux/man-pages/man5/elf.5.html).

Der Linker benötigt zusätzlich eine «ID – Datei».
Dies ist ein sogenanntes Linker Skript (https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_chapter/ld_3.html) und beinhaltet Anweisungen für den Linker, um ein Kernel Image zu erstellen, dass auf dem Raspberry PI ausgeführt werden kann.

3. «aarch64-linux-gnu-objcopy»
Dieses Tool erstellt aus der *.elf Datei das eigentliche Kernel Image. Es wird mit dem Parameter «-O binary» aufgerufen, dieser weist «objcopy» an ein sogenannter «raw binary» zu erstellen. (https://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html_node/binutils_5.html)

3. Umsetzung

3.1. Grundidee

Eine mit dem Raspberry PI verbundene LED soll durch Assembler Code zum Blinken gebracht werden. Die LED soll einen Text, welcher im Code definiert wird im Morsecode wiedergeben. Der zu blinkende Text soll ohne Anpassungen am Mechanismus geändert werden können.

3.2. Vorbereitung

Damit das Projekt «Bare Metal Coding» in Assembler angegangen werden kann, muss zuerst eine Entwicklungsumgebung geschaffen werden und einige Kompilierungstools installiert werden. Die Tools wurden bereits in Kapitel 2 definiert. Wir haben uns entschieden, die Toolchain in beiden Fällen (Windows und Linux) via CLI zu installieren. Hierzu haben wir unter Windows «WSL» aktiviert und konnten anschliessend die Installation analog zu Linux durchführen. Die grundsätzliche Installation der Toolchain erfolgt mit dem CLI Befehl: «sudo apt install git make gcc-aarch64-linux-gnu».

Um zusammen am Projekt arbeiten zu können, haben wir uns entschieden das Projekt per «Git» zu verwalten und haben es in ein GitHub Repository hochgeladen, wo auch die genauen Schritte inklusive Befehlen für die Vorbereitung dokumentiert sind.

Das anschliessen der LED an den Raspberry Pi haben wir mittels einer Anleitung aus dem Internet gemacht (https://www.raspberrypi.com/news/how-to-use-an-led-with-raspberry-pi/). Hierbei wird das Setup inklusive des Widerstands gut erklärt und beschrieben.

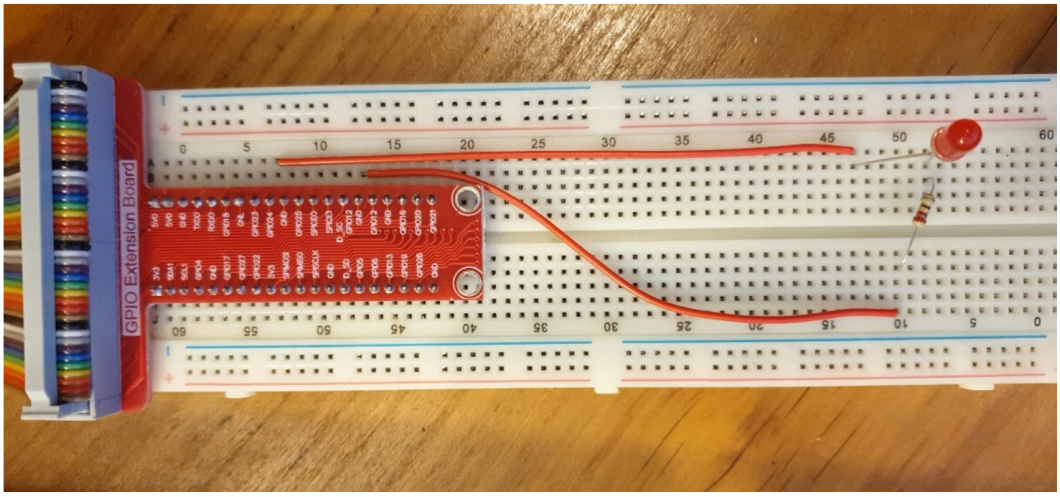

Abbildung 1: Unser Setup auf dem GPIO Extension Board

Um die Funktionalität des Aufbaus und der LED zu testen, haben wir zuerst mittels «dd» eine Raspios auf die SD-Karte geschrieben und mit einem simplen Python Script den Aufbau getestet. Dieser Schritt hat und auch gleich schon eine «boot» Partition auf er SD-Karte erstellt, welche wir zu einem späteren Schritt benötigen.

3.3. Der Code

Wir haben zuerst mit einer einfachen Funktion begonnen und das LED ganz simple Blinken lassen, indem wir auf dem Entsprechenden GPIO Pin den Stromfluss ein- und ausgestellt haben. Sobald dies Funktionierte, haben wir uns den Punkten aus der Aufgabenstellung gewidmet und die zuvor entwickelte Funktion stetig erweitert:

1. Assembler Code, um die LED zum Leuchten zu bringen
2. Assembler Code, um die LED blinken zu lassen mittels Hardware Timer
3. Bestehenden Code mit Stackpointer, Function-Call und variablem Text erweitern
4. Code optimieren und verbessern

Wir haben die RAM Adresse in einem externen «linker» Script (sys.ld) definiert. Das Programm startet, vom Bootloader aufgerufen, an der Adresse 0x80000. Der Stack beginnt ab 0x80000 und wächst gegen 0x0.

src > sys.ld

```
1  /* https://www.math.utah.edu/docs/info/ld_3.html */
2
3  MEMORY
4  {
5      ram : ORIGIN = 0x80000, LENGTH = 0x10000
6  }
7
8  SECTIONS
9  {
10     .text   : {*(.text*)} > ram
11     .bss    : {*(.bss*)} > ram
12 }
13
```

Abbildung 2: sys.ld Script

Den Stackpointer haben wir mit 0x80000 initialisiert, dieser wächst ebenfalls gegen 0x0 wobei sich der PC zwischen 0x80000 und 0x90000 befindet.

```
26
27  .L_START:   /* We're on the main core! */
28      mov     sp, #0x80000            /* init stack pointer */
```

Abbildung 3: Stackpointer init

Die Funktion des Programms wird wie folgt aufgerufen.

```
417  .L43: /* continue loop */
418      ldr     x0, [sp, 24]            /* load into register: load back sp + 24bit into x0 (text pointer)*/
419      ldrb    w0, [x0]                /* load into register (byte): get char from x0 address (w0 will be argument for blink(char c) */
420      bl      blink                   /* branch and link: call blink(w0) */
421      ldr     x0, [sp, 24]            /* and again load back text address pointer */
422      add     x0, x0, 1               /* increment text address pointer by one */
```

Abbildung 4: Funktionsaufruf

Auf Zeile 419 wird das Funktionsargument gespeichert und danach (Zeile 420) wird «blink» mittels «bl» (branch and link) aufgerufen. Zum Schluss des Aufrufes werden die Register aus dem Stack geladen und deren Kontext wiederhergestellt.

Die soeben aufgerufene Funktion speichert x29 (Framepointer) und x30 (Procedure link) im Stack und x29 bekommt (auf Zeile 154) die neue Stackpointer Adresse.

```
147  blink:
148  .LFB7:
149      stp     x29, x30, [sp, -32]!  /* store pair (x29 und x30) into sp -32, sp = sp -32
150                                       https://stackoverflow.com/questions/64638627/expla
151                                       1. suptract 32 from address stored at sp (stack po
152                                       2. store x29 and x30 into memory at address stored
153                                    */
154      mov     x29, sp               /* store stack pointer into register 29 */
155      strb    w0, [sp, 31]          /* store argument (char) into stack at 31 */
156      ldrb    w0, [sp, 31]          /* .. load back into register w0 (why?) */
157      cmp     w0, 97                /* is it 97 ('a') ? */
158      bne     .L11                  /* else try next at .L11 */
```

Abbildung 5: Funktion

Am ende der Funktion, wird mittels «ret» (return) an den vorherigen PC zurückgekehrt.

```
400  .L12: /* end of function (everithing meets here) */
401      mov     w0, 2
402      bl      space
403      ldp     x29, x30, [sp], 32    /* free stack and restore registers... */
404      ret                           /* ... and return to old pc */
405
```

Abbildung 6: Return

Auf Zeile 403 wird der vorherige Stackframe und Procedure link geladen und auf Zeile 404 wird auf die vorherige Adresse gesprungen.

Nach einigen versuchen und viel Codier Arbeit, konnten wir im Code einen Text/String definieren, welcher anschliessend in Morsezeichen auf der LED geblinkt wird. Mit dieser Erweiterung decken wir aus der Aufgabenstellung die Zusätzliche Aufgabe «*Ausgabe eines beliebigen Textes (z.B. «Hello world»), wobei die Morsesequenz im RAM abgespeichert ist.*» ab.

Der fertige Code (inkl. genauer Code-Kommentare) ist auf GitHub einsehbar:

https://github.com/bailuk/raspberry

Der Code ist mit Kommentaren versehen, weshalb hier auf eine genauere Erklärung verzichtet wird.

3.4. Code übersetzten und ausführen

Der geschriebene Code in Assembler muss nun auf die SD-Karte geschrieben werden. Dazu haben wir den Command «make» auf unserem Linux System. Mit Hilfe dieses Tools wird der geschriebene Code kompiliert. Es werden nun zwei neue Dateien erstellt, die «config.txt» und die «kernel8.img». Diese beiden Dateien nehmen wir und überschreiben die gleichnamigen Dateien auf unserer SD-Karte in dessen «boot» - Partition. Nach dem Kopiervorgang kann die SD-Karte in unseren Raspberry Pi 4 eingesteckt werden und dieser kann gestartet werden. Sofern der Code korrekt ist und beim Kompilieren keine Fehler aufgetreten sind, sollte die LED am Raspberry nun im zuvor festgelegten Morsecode blinken.

4. Schlusswort

Das Programmieren in Assembler ist uns zu Beginn sehr schwergefallen. Obwohl wir beide über gewisse Programmierkenntnisse verfügen, war der Einstieg in eine solch Systemnahe Sprache schwierig. Wir taten uns zu beginn schwer, die GPIO Base Adresse zu finden, was uns etwas Zeit gekostet hat. Das «Headless» Setup, ohne Debugging Möglichkeiten, hat gerade zu beginn für etwas Frust gesorgt. Ohne Fehlermeldung und ohne hinweise war das Suchen und Finden des, meist kleinen, Fehler im Code nahezu unmöglich. Aus diesem Grund haben wir uns für eine «Schritt für Schritt» Vorgehensweise (wie im Kapitel 3 erwähnt) entschieden. Dadurch konnten wir falls nötig immer wieder zu einem funktionierenden Code zurückkehren und uns nochmals von vorne dem nächsten Ziel widmen.

Das Projekt hat uns geholfen einige der in den Vorlesungen behandelten Themen zu vertiefen. Wir haben die Funktionsweise der GPIO Pins kennengelernt, bis hin zum Programmieren in Assembler. Dies war für uns beide neu. Nach einiger Zeit ist uns das Programmieren leichter gefallen und wir hatten das nötige Grundwissen zur Anwendung dieser Programmiersprache aufgebaut.

Mit der gewählten Vorgehensweise waren wir beide zufrieden und würden deshalb auch nochmals die gleiche Methode wählen. Verbessern würden wir die Informationsbeschaffung zu beginn. Uns war der Auftrag zu beginn des Projektes nicht klar und wir tappten noch etwas im Dunkeln. Zudem wäre Ersatz Hardware optimal, da uns ein Raspberry beim Testen kaputt gegangen ist.

5. Anhang

5.1. Code

Repository mit Source-Code und Dokumentation: https://github.com/bailuk/raspberry

Assembler Code: https://github.com/bailuk/raspberry/blob/main/src/led-blink.s

5.2. Flowchart

Als separates PDF: https://github.com/bailuk/raspberry/blob/main/doc/activity.pdf

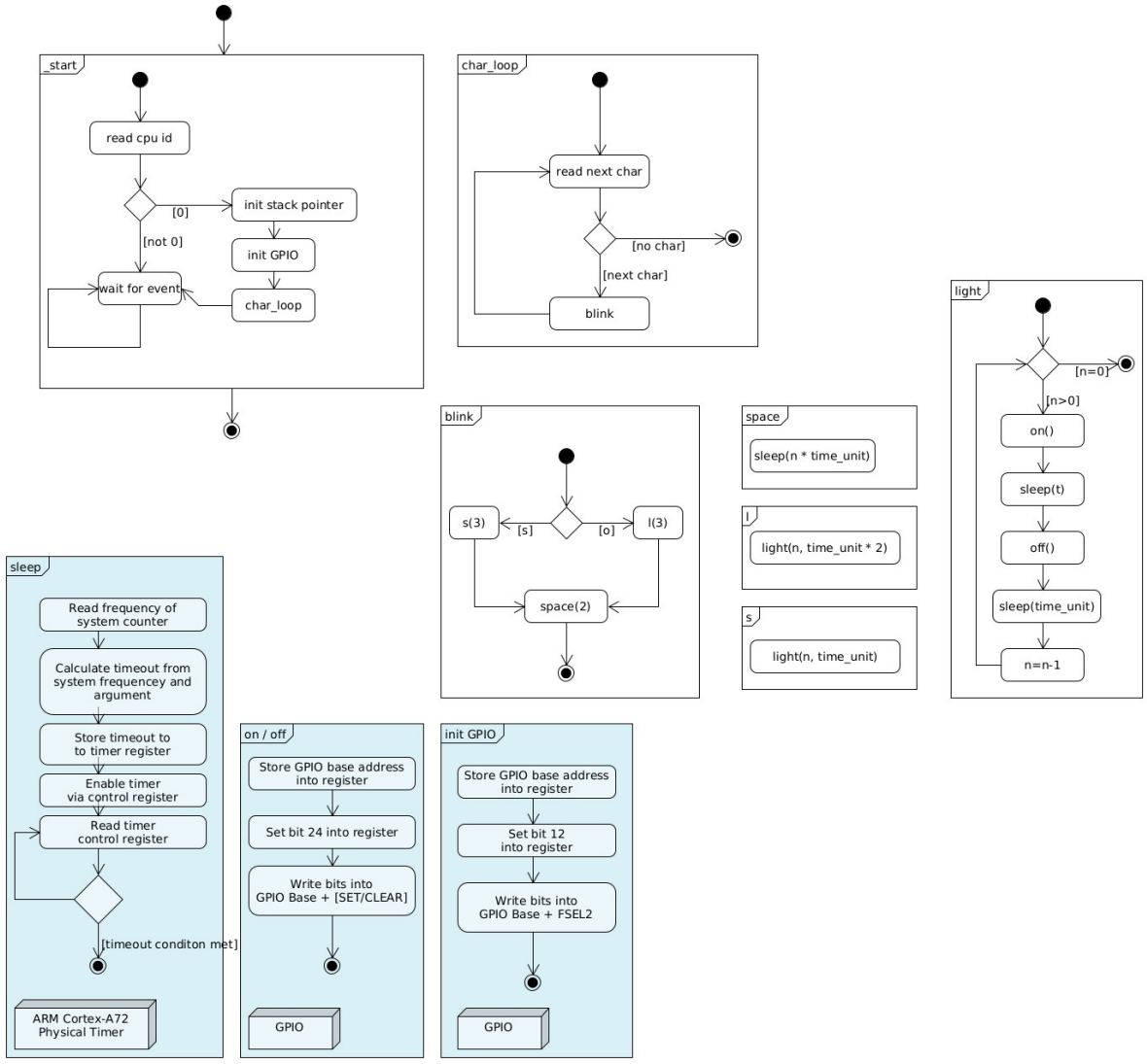

Abbildung 7: Flowchart (UML Activity Diagram)

5.3. Präsentation

https://github.com/bailuk/raspberry/blob/main/doc/GTI_Projektarbeit_Praesentation_Team3.pptx