Projektarbeit GTI

Erzeugung eines visuellen und akustischen Morse-Signals mit dem PI 3B+ in Assembler

Eveline Hutter – Franziska Walker – Christian Eiselt

Inhaltsverzeichnis

[1. Aufgabenstellung 3](#_Toc24449974)

[2. Beschreibung des verwendeten Rechners 3](#_Toc24449975)

[2.1. Raspberry Pi 3](#_Toc24449976)

[2.2. Die CPU: der ARMv8 3](#_Toc24449977)

[2.2.1. Die General Register der CPU 3](#_Toc24449978)

[2.3. General Purpose Input/Output (GPIO) 4](#_Toc24449979)

[3. Beschreibung der verwendeten Elektronikbauteile 4](#_Toc24449980)

[3.1. Schaltung 4](#_Toc24449981)

[3.2. Die Steckplatine 5](#_Toc24449982)

[3.3. LED 6](#_Toc24449983)

[3.4. Piezo-Lautsprecher 6](#_Toc24449984)

[3.5. Widerstand 6](#_Toc24449985)

[3.6. Verbindungskabel 6](#_Toc24449986)

[4. Dokumentation und Erklärung der Programmier-Konzepte 6](#_Toc24449987)

[4.1. Bare Metal Programmierung 6](#_Toc24449988)

[4.2. Die Erstellung eines Assembler-Programms und eines Boot-Images beim ARMv8 7](#_Toc24449989)

[4.3. Assembly Code 7](#_Toc24449990)

[4.4. Angabe von Zahlensystemen im Programmcode 7](#_Toc24449991)

[5. Dokumentation und Erklärung der verwendeten Assembler-Befehle 7](#_Toc24449992)

[6. Dokumentation und Erklärung der Lösungsansätze 8](#_Toc24449993)

[6.1. Adressierung 8](#_Toc24449994)

[6.2. Taktunabhängigkeit 9](#_Toc24449995)

[6.3. Initialisierung der GPIOs 10](#_Toc24449996)

[6.4. Generierung des Morse-Signals 10](#_Toc24449997)

[6.4.1. Exkurs: Morse-Alphabet 11](#_Toc24449998)

[6.4.2. Praktische Anwendung der Morse-Code-Richtlinien 12](#_Toc24449999)

[6.5. Besonderheiten des Audiosignals 14](#_Toc24450000)

[6.5.1. Ursprünglicher Lösungsansatz: PWM 15](#_Toc24450001)

[6.5.2. Verwendeter Lösungsansatz: Frequenzgenerierung innerhalb «dit» und «dah» 15](#_Toc24450002)

[6.5.3. Exkurs: Frequenzbewertung / dB(A) 16](#_Toc24450003)

[7. Reflexion der Arbeit 16](#_Toc24450004)

[7.1. Eveline Hutter 16](#_Toc24450005)

[7.2. Franziska Walker 17](#_Toc24450006)

[8. Literaturverzeichnis 18](#_Toc24450007)

[9. Anhang 20](#_Toc24450008)

[9.1. Quellcode: kernel.asm 20](#_Toc24450009)

Abbildungsverzeichnis

[Abbildung 1: Quellcode-Abschnitt der verwendeten General Purpose Register 4](#_Toc24450010)

[Abbildung 2: Schaltungsschema für die vorliegende Arbeit (LED: GPIO17, Lautsprecher: GPIO18) 5](#_Toc24450011)

[Abbildung 3: Steckplatine 5](#_Toc24450012)

[Abbildung 4: LED 6](#_Toc24450013)

[Abbildung 5: Piezo-Lautsprecher 6](#_Toc24450014)

[Abbildung 6: 220 Ohm Widerstand 6](#_Toc24450015)

[Abbildung 7: Verbindungskabel 6](#_Toc24450016)

[Abbildung 8: Quellcode-Abschnitt der Variablen für die Adressierung 9](#_Toc24450017)

[Abbildung 9: Flowchart des Makros «Timer» 9](https://ffhs-my.sharepoint.com/personal/christian_eiselt_students_ffhs_ch/Documents/2019%20HS/19HS.GTI/Projektarbeit/Projektarbeit%20GTI.docx#_Toc24450018)

[Abbildung 10: Quellcode-Abschnitt des Makros «Timer» 10](https://ffhs-my.sharepoint.com/personal/christian_eiselt_students_ffhs_ch/Documents/2019%20HS/19HS.GTI/Projektarbeit/Projektarbeit%20GTI.docx#_Toc24450019)

[Abbildung 11: Quellcode-Abschnitt der GPIO-Initialisierung 10](#_Toc24450020)

[Abbildung 12: Flowchart SOS-Morsecode 11](https://ffhs-my.sharepoint.com/personal/christian_eiselt_students_ffhs_ch/Documents/2019%20HS/19HS.GTI/Projektarbeit/Projektarbeit%20GTI.docx#_Toc24450021)

[Abbildung 13: Quellcode-Abschnitt der Festlegung des Morse-Signals 11](#_Toc24450022)

[Abbildung 14: Quellcode-Abschnitt des Branches für die Zeiteinheit Dit 12](#_Toc24450023)

[Abbildung 15: Flowchart des Branches für ein Dit 13](#_Toc24450024)

[Abbildung 16: Quellcode-Abschnitt des Branches für ein Dit 13](#_Toc24450025)

[Abbildung 17: Quellcode-Abschnitt der Branches für Buchstaben und Zahlen (Bsp. Buchstabe «C») 14](#_Toc24450026)

[Abbildung 18: Quellcode-Abschnitt des Branches für ein neues Wort 14](#_Toc24450027)

Tabellenverzeichnis

[Tabelle 1: Dokumentation und Erklärung der verwendeten Assembler-Befehle 7](#_Toc24450028)

[Tabelle 2: Berechnung der nötigen Zeiteinheiten für das Wort PARIS 12](#_Toc24450029)

# Aufgabenstellung

In der vorliegenden Arbeit werden Lösungen beschrieben, wie man mit Bare Metal Programmierung auf dem Rasberry Pi spezifische Funktionalitäten bauen kann. Als erste Arbeit wird verlangt, dass ein SOS Signal, welches unabhängig von der Takt-Frequenz ist, in Morse Code ausgegeben wird. Der Raspberry Pi muss dazu um eine LED-Komponente an einem GPIO-Port erweitert werden. Mittels Assembler-Routine soll die LED angesteuert werden und damit ein SOS-Signal regelmässig wiederkehrend erzeugt werden. Dabei soll das Morse-Signal automatisch erzeugt werden, sobald der Raspberry Pi an eine Stromversorgung angeschlossen wird. Zusätzlich soll der Code so angelegt sein, dass auch beliebige andere Morse-Signale geblinkt werden können. Als Zusatzaufgabe soll das Morse-Signal akustisch ausgegeben werden.

# Beschreibung des verwendeten Rechners

Die Projektaufgabe wird mit dem Einplatinen-Computer «Raspberry Pi 3 B+» umgesetzt, der mittels der Maschinensprache «Assembler» programmiert wird. Die visuelle und akustische Signalisation erfolgt mittels LED und Lautsprecher.

## Raspberry Pi

Der von der britischen Raspberry Pi Foundation im Jahr 2012 entwickelte Computer sollte jungen Menschen auf der ganzen Welt das Programmieren sowie die Hardware eines Rechners näher bringen. Nicht zuletzt aufgrund des sehr günstigen Verkaufspreises (Storm, 2018) von 5 – 60 Euro (je nach Modell) wurden bis Ende 2018 über 20 Millionen Exemplare verkauft – 5 Millionen allein im Jahr 2018 (Raspberry Pi Foundation, 2018).

## Die CPU: der ARMv8

Der verwendete Computer verfügt über einen Cortex-A53 Prozessor mit ARMv8-A Architektur, welcher vom Unternehmen ARM Limited entwickelt wurde. ARM Limited stellt die Prozessoren jedoch nicht selber her, sondern vergibt vielmehr Lizenzen an externe Unternehmen (z.B. Intel). Der Prozessor im vorliegenden Raspberry Pi wurde vom Unternehmen Broadcom Inc. als BCM2837 hergestellt.

ARM stand früher für Advanced RISC Machines Ltd, da die ARM Prozessorarchitektur auf dem RISC-Prinzip (Reduced Instruction Set Computer) basierte (ARM Limited, 2019b). Der ARMv8 ist weltweit einer der am weitesten verbreiteten Prozessoren. Aufgrund seines geringen Stromverbrauchs bei gleichzeitiger hoher Leistungsstärke findet er z.B. in vielen Smartphones Verwendung (Android Wiki, 2016).

Der ARMv8 Prozessor hat als erster nicht mehr eine 32-Bit-, sondern eine 64-Bit-Architektur. Hier kommt auch ein neuer Befehlssatz vor: das A64 Instruction Set (ARM Limited, 2017, 2019e). Trotzdem verfügt der ARMv8 über eine volle Abwärtskompatibilität zu ARMv7, da er neben dem AArch64-Modus zur Verarbeitung von 64-bit-Software auch im AArch32-Modus ausgeführt werden kann (ARM Limited, 2019c).

### Die General Register der CPU

Die Programme müssen in die General Registers geladen werden, denn nur von dort kann die Programmierung starten. Die Register befinden sich direkt auf dem CPU-Chip und sind daher sehr schnell erreichbar. Sie sind nicht «Memory Mapped», dies bedeutet, sie haben keine 26 Bit Adressen wie der Rest des Speichers (Cockerell, o.J.).

Der verwendete Prozessor erlaubt Zugriff auf 16 Register, welche je 32 Bit lang sind (Arm Limited, 2019d). R0 – R14 sind «General Purpose Register», das heisst, sie können theoretisch für alles verwendet werden. Die letzten zwei dieser 15 Register sind jedoch für einen speziellen Zweck angelegt und sollten generell nicht verwendet werden. Der «Program Counter» befindet sich auf dem letzten Register R15. Er wird auf 0 gesetzt zu Beginn des Programms und auf den Bits 2 – 25 auf den Adressbuss gelegt (Cockerell, o.J.). Spezielle Register sind somit:

* R13 (Stack Pointer SP): zeigt auf die oberste Adresse des Stack
* R14 (Load Register LR): läuft eine Subroutine, wird der Program Counter (PC) hier gespeichert
* R15 (Program Counter PC): enthält die Memory Adresse der nächsten Instruktion.

Im vorliegenden Code werden die Register R0-R3 für verschiedene Zwecke genutzt, R5-R8 dienen dem Timer-Makro und R11-R12 werden verwendet, um Rücksprungadressen zu speichern (vgl. Abbildung 1 für den entsprechenden Kommentar zu Beginn des Quellcodes).

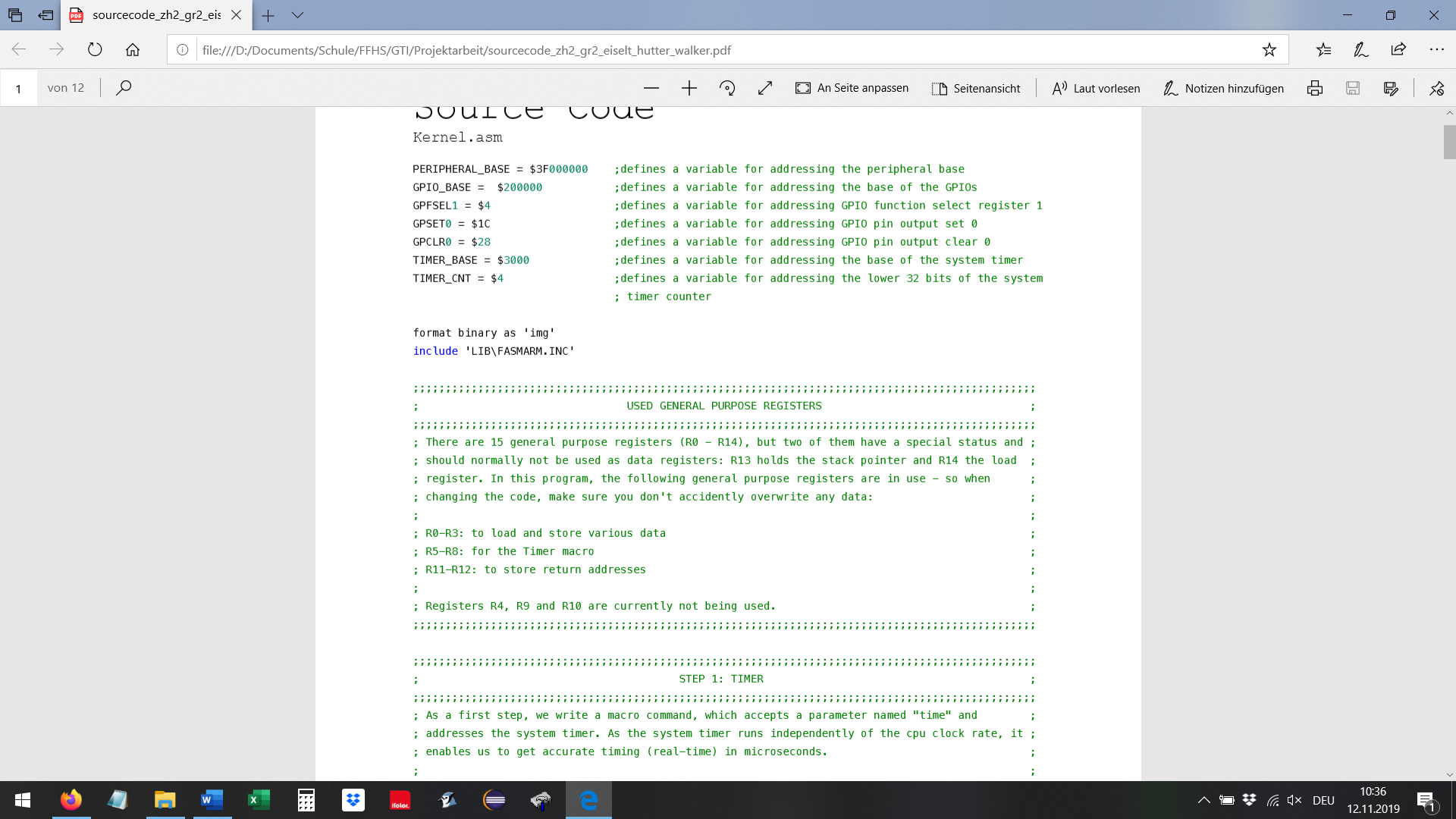


Abbildung 1: Quellcode-Abschnitt der verwendeten General Purpose Register

## General Purpose Input/Output (GPIO)

Der verwendete Raspberry Pi wird mit 40 GPIO-Pins («General Purpose Input/Output») geliefert, die frei verwendet werden können. Sie lassen sich sowohl zur Eingabe (z.B. für Sensoren) als auch zur Signalausgabe benutzen.

Das GPIO-Board enthält diverse 3V3-Pins (3.3 Volt), die für die Signalausgabe auf high (3V3 = z.B. LED an) oder low (0V = LED aus) gesetzt werden können (Raspberry Pi Foundation, o.J.). Für die Ansteuerung der LED und des Lautsprechers werden Signale digital an zwei ausgewählte GPIO-Pins gesendet.

# Beschreibung der verwendeten Elektronikbauteile

Für die visuelle und akustische Signalisation werden eine LED, ein Lautsprecher, eine Steckplatine und zwei Vorwiderstände sowie drei Verbindungskabel zur Verbindung mit dem Raspberry Pi benötigt. Im Folgenden werden diese Bauteile näher beschrieben und mit Bildern dokumentiert. Diese Abbildungen wurden mit der Software Fritzing, Version 0.9.3b erstellt.

## Schaltung

Die untenstehende Abbildung entspricht der verwendeten Schaltung:

* GPIO-Pin 17 wird mittels eines Verbindungskabels (rot) mit der Steckplatine verbunden. Auf der verbundenen Leitung befindet sich zur Stromverringerung ein 220 Ohm Vorwiderstand vor der LED (rot).
* Die Schaltung für das akustische Signal ist analog der für das visuelle. Statt Pin 17 wird der Lautsprecher mit Pin 18 verbunden (oranges Kabel).
* Der Minus-Pol beider Bauteile wird, eingesteckt auf die äussere rechte Leitungsbahn, durch ein Verbindungskabel (braun) mit dem Pin 6 (GND) verbunden, welcher dem Masse-Potential entspricht (Schnabel, 2019b).

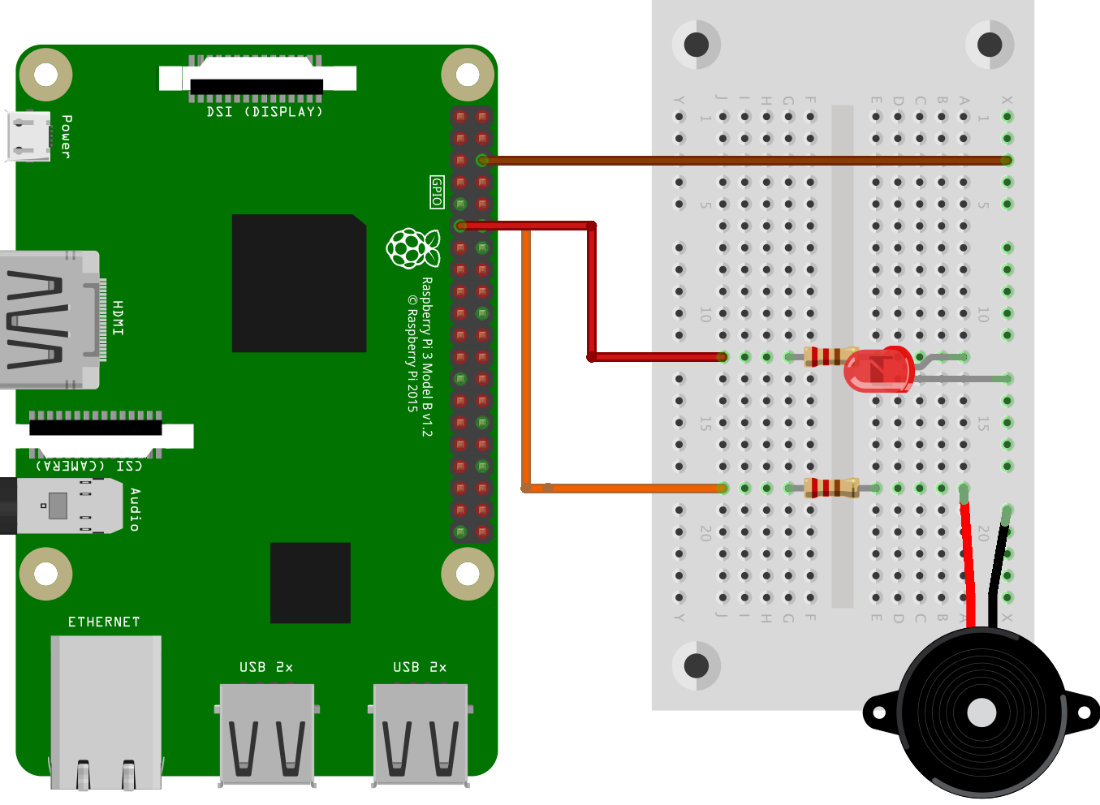


Abbildung 2: Schaltungsschema für die vorliegende Arbeit (LED: GPIO17, Lautsprecher: GPIO18)

## Die Steckplatine

Für die Verbindung der Bauteile wird eine Steckplatine verwendet. Die Verbindungen der Steckkontakte sind auf der folgenden Abbildung gekennzeichnet. Durch die Verwendung dieser Steckplatine müssen die Kontakte der Bauteile nicht mit den Verbindungskabeln oder gar dem Raspberry Pi verlötet werden.

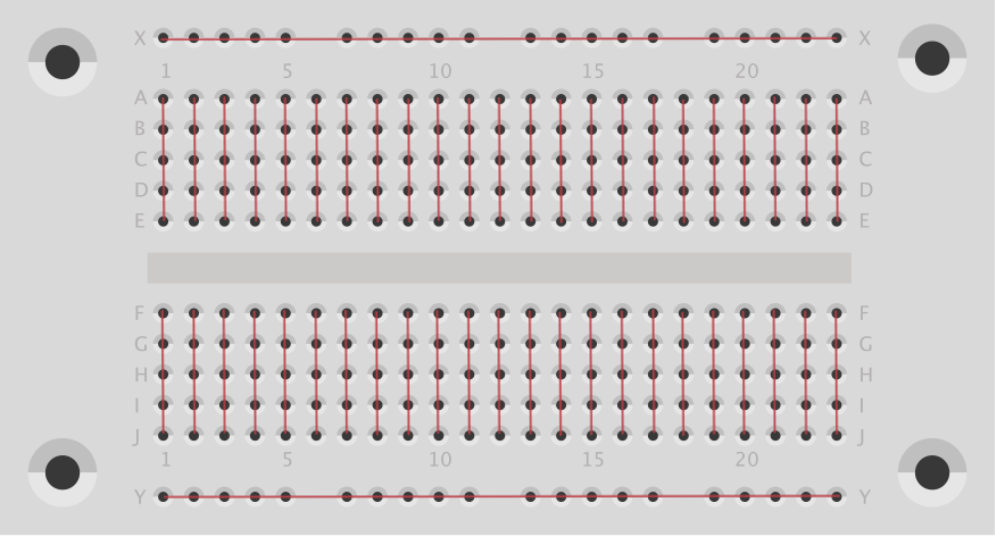


Abbildung : Steckplatine

## LED

Mit der LED («Light Emitting Diode») wird in dieser Schaltung ein visuelles Signal ausgegeben. LEDs können sehr schnell (bis in den MHz-Bereich) vom leuchtenden in den nichtleuchtenden Zustand schalten (Schnabel, 2019a).



Abbildung : LED

## Piezo-Lautsprecher

Der verwendete Piezo-Lautsprecher gibt das Signal akustisch in Form eines Pieptons bzw. der gewünschten Piepton-Sequenz aus.

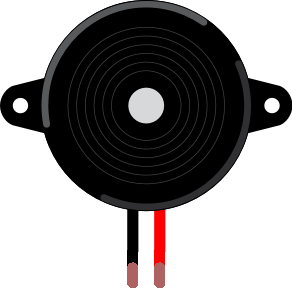


Abbildung : Piezo-Lautsprecher

## Widerstand

Für die Ansteuerung der LED und des Lautsprechers wird je ein Vorwiderstand mit 220 Ohm zwischengeschaltet, um den Strom zu begrenzen, der durch die LED fliesst (Schnabel, 2019a).



Abbildung : 220 Ohm Widerstand

## Verbindungskabel

Die Kabel werden verwendet, um die Plus- und Minus-Pole der Bauteile mit den entsprechenden Pins des Raspberry Pi zu verbinden.



Abbildung : Verbindungskabel

# Dokumentation und Erklärung der Programmier-Konzepte

## Bare Metal Programmierung

Bei der Bare Metal Programmierung wird die Programmierung direkt auf der Hardware, also ohne Einsatz eines Betriebssystems, durchgeführt. Die Befehle können von der CPU direkt interpretiert werden. Heute wird Bare Metal Programmierung insbesondere für zeitkritische Systeme benutzt. Diese Systeme haben oft nur einen Zweck, während beispielsweise moderne Computer mannigfaltigste Einsatzmöglichkeiten bieten (Wikipedia-Autoren, 2019).

## Die Erstellung eines Assembler-Programms und eines Boot-Images beim ARMv8

Um eine Boot-Image-Datei auf die SD Karte zu schreiben, braucht es ein Assembler Programm, welches die Befehle von einer .asm Datei auf eine .img Datei umwandelt. Zur Erstellung der .img Datei wird ein Compiler benötigt. Es gibt verschiedene Compiler, wobei für die vorliegende Arbeit der Compiler FASMARM, Version 1.43 verwendet wurde.

Die Karte muss im Format FAT32 sein. Eine Anleitung, diese zu formatieren, gibt DashCam Bros (2017). Die notwendigen Dateien, dass der Raspberry die SD-Karte erkennt und bootet sind: bootcode.bin, config.txt, kernel.asm, kernel.img, start.elf

## Assembly Code

Assembly Code ist das tiefste Level des vom Menschen lesbaren Codes. Die Assembly Sprache kann dazu verwendet werden, um eigene Kernels (.img datei) zu implementieren, welche beim Stromanschluss geladen werden. Assembly Code übersetzt die binäre Anleitung für den PC in kurze Wörter, die Mnemonics (Cockerell, o.J.).

## Angabe von Zahlensystemen im Programmcode

Bei der Eingabe von Zahlen muss nach dem #-Symbol ein Zusatz geschrieben werden, damit der Compiler weiss, welches Zahlensystem verwendet werden soll. Dies geschieht wie in den folgenden Beispielen beschrieben (in allen soll die Dezimalzahl 25 verwendet werden).

1. Dezimalzahlen entsprechen der Standardform und benötigen keinen Zusatz  #25
2. Hexadezimalzahlen benötigen den Zusatz «#0x»  #0x19
3. Manche Compiler ermöglichen wohl auch die Verwendung von Binärzahlen in der Form «#0b»  #0b11001. FASMARM gibt jedoch eine Fehlermeldung aus, wenn eine Zahl in dieser Form eingegeben wird.

# Dokumentation und Erklärung der verwendeten Assembler-Befehle

Tabelle 1: Dokumentation und Erklärung der verwendeten Assembler-Befehle

|  |  |  |  |
| --- | --- | --- | --- |
| Befehl | Name | Beispiel für die Verwendung im Quellcode | Beschreibung (in Anlehnung an Cockerell, o.J.) |
| include | include | include 'LIB\FASMARM.INC' | Integriert ein File in den Code. |
| mov | move | mov r6, PERIPHERAL\_BASE | Setzt den im zweiten Parameter angegebenen Zahlenwert (wird mit # angeführt) oder die angegebene Registeradresse bzw. Variable an die im ersten Parameter angegebene Registeradresse. |
| orr | logical or | orr r6, r6, TIMER\_BASE | Die Verknüpfung ist eine ODER Verknüpfung, welche die zweite und dritte Variable bitweise vergleicht und das Resultat in die erste Variable schreibt. Regeln sind: 0+0=0, 0+1=1, 1+0=1, 1+1=1. |
| ldr | load | ldr r7, [r6, TIMER\_CNT] | Lädt ein Register (zweite Variable) von einem spezifizierten Ort (aus dem Register oder RAM) und speichert es in ein Register (erste Variable). |
| sub | substraction | sub r5, r5, r7 | Speichert eine Subtraktion (zweite Variable – dritte Variable) in einem Register (erste Variable). |
| cmp | compare | cmp r5, r8 | Vergleicht zwei Register miteinander. Der Compare Operator wird gefolgt von verschiedenen anderen Befehlen, die auf das Ergebnis des Vergleiches aufbauen. |
| blt | branch if less than | blt .wait | Falls der vorausgehende Vergleich kleiner oder gleich ist, führt das Programm den definierten Loop nochmals aus. |
| lsl | logical shift left | lsl r1, #21 | Verschiebt die Bitfolge des 32-Bit-Registers nach links. Die erste Variable definiert das Register, die zweite die Anzahl Linksverschiebungen. |
| str | store | str r1, [r0, GPFSEL1] | Speichert einen Wert (zweite Variable) in ein Register (erste Variable). |
| bl [label] | branch to subroutine at [label] | bl letter\_s | Bringt den Prozess zu der benannten Subroutine. |
| b [label] | branch to [label] | b loop | Bringt den Prozess zu dem Label. |
| bx lr | branch indirect to load register | bx lr | Lädt das Instruction Set aus dem Load Register (R14) in den Program Counter (R15) und führt das Programm von dort aus weiter. |
| movs | move with update flags | movs r11 ,r14 | Führt den Move-Befehl inkl. Flags aus. |
| bne | branch if not equal | bne countdown\_dit | Falls der vorausgehende Vergleich nicht gleich ist, führt das Programm den definierten Loop nochmals aus. |
| bx [register] | branch indirect to [register] | bx r11 | Lädt das Instruction Set aus [register] in den Program Counter (R15) und führt das Programm von dort aus weiter. |

# Dokumentation und Erklärung der Lösungsansätze

In den folgenden Abschnitten wird unsere Herangehensweise an die vorgegebenen Aufgaben (s. Kapitel 1) genau beschrieben: zuerst wird kurz auf die Adressierung eingegangen, während anschliessend das Erreichen der Taktunabhängigkeit, das Ansteuern der GPIOs, die Generierung des visuellen Signals und schliesslich die Besonderheiten des Audiosignals erklärt werden. Die Flowcharts für die Programmabschnitte des Makros «Timer», der SOS-Morsesignal-Schleife sowie der Subroutine für ein Dit befinden sich jeweils bei den entsprechenden Erläuterungen.

## Adressierung

Um eine bessere Lesbarkeit des Codes zu erreichen, werden als Erstes Variablen für die verwendeten Hardware-Adressen eingeführt. Damit können im weiteren Verlauf des Codes die Namen der Variablen angegeben werden, wodurch leicht ersichtlich ist, welche Komponente des Prozessors angesprochen wird. Konkret werden die folgenden Variablen erstellt.

* PERIPHERAL\_BASE = die Basis der Peripherie
* GPIO\_BASE = die Basis der GPIOs
* GPFSEL1 = das GPIO function select register 1, in welchem die beiden von uns verwendeten GPIOs (GPIO17 und GPIO18) als Outputs definiert werden können
* GPSET0 = das GPIO pin output set 0, in welchem die Signalausgabe der beiden von uns verwendeten GPIOs gestartet werden kann
* GPCLR0 = das GPIO pin output clear 0, in welchem die Signalausgabe der beiden von uns verwendeten GPIOs beendet werden kann
* TIMER\_BASE = die Basis des System Timers
* TIMER\_CNT = die unteren 32 Bits des System Timer Counters (s. Kapitel 6.2)

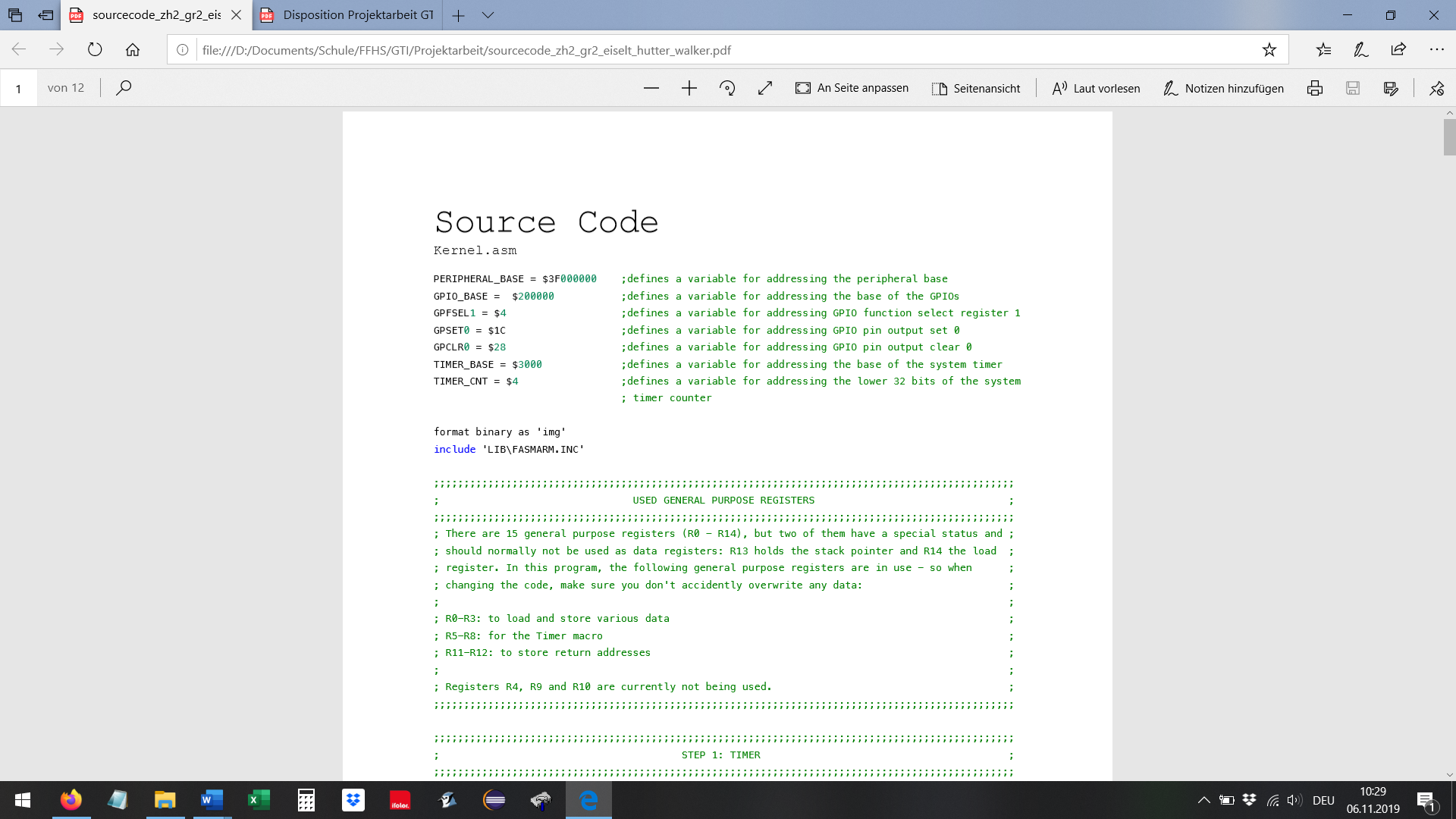


Abbildung 8: Quellcode-Abschnitt der Variablen für die Adressierung

## Taktunabhängigkeit

Im Beispiel-Code aus der Moodle-Vorlage (FFHS, 2019a[[1]](#footnote-1)) wird die Wartezeit über einen Loop, der von einer hohen Zahl rückwärts zählt, definiert. Durch diesen Code wird die Wartezeit taktabhängig. Das bietet Nachteile, da die Taktfrequenz variieren kann und daher schlecht geeignet ist, um genaue Zeiten zu definieren.

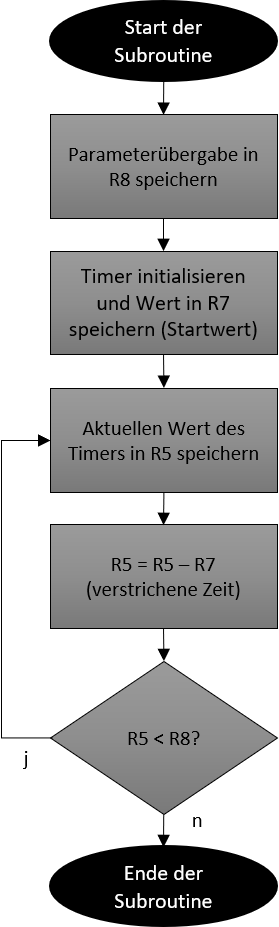


Abbildung 9: Flowchart des Makros «Timer»

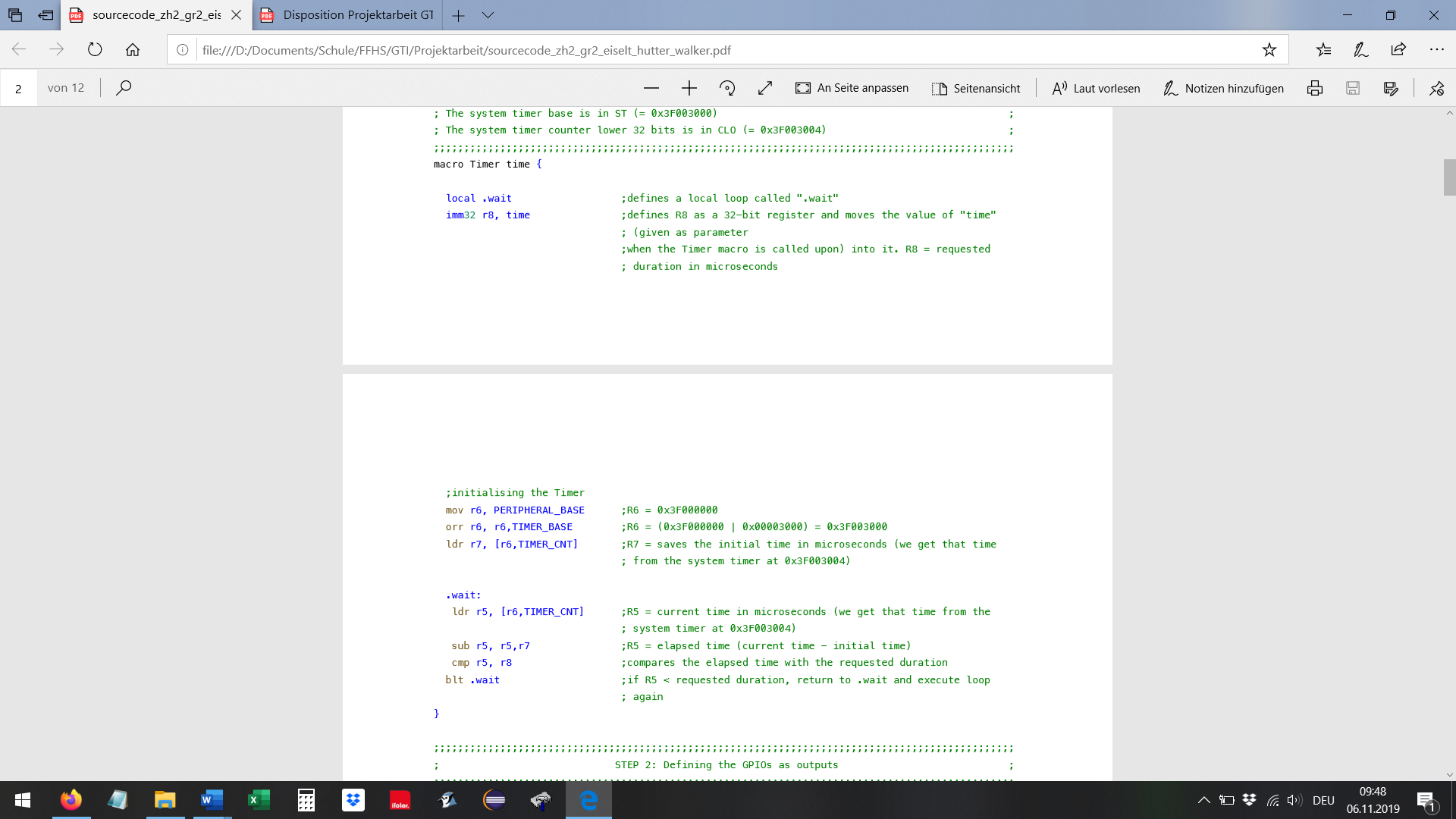
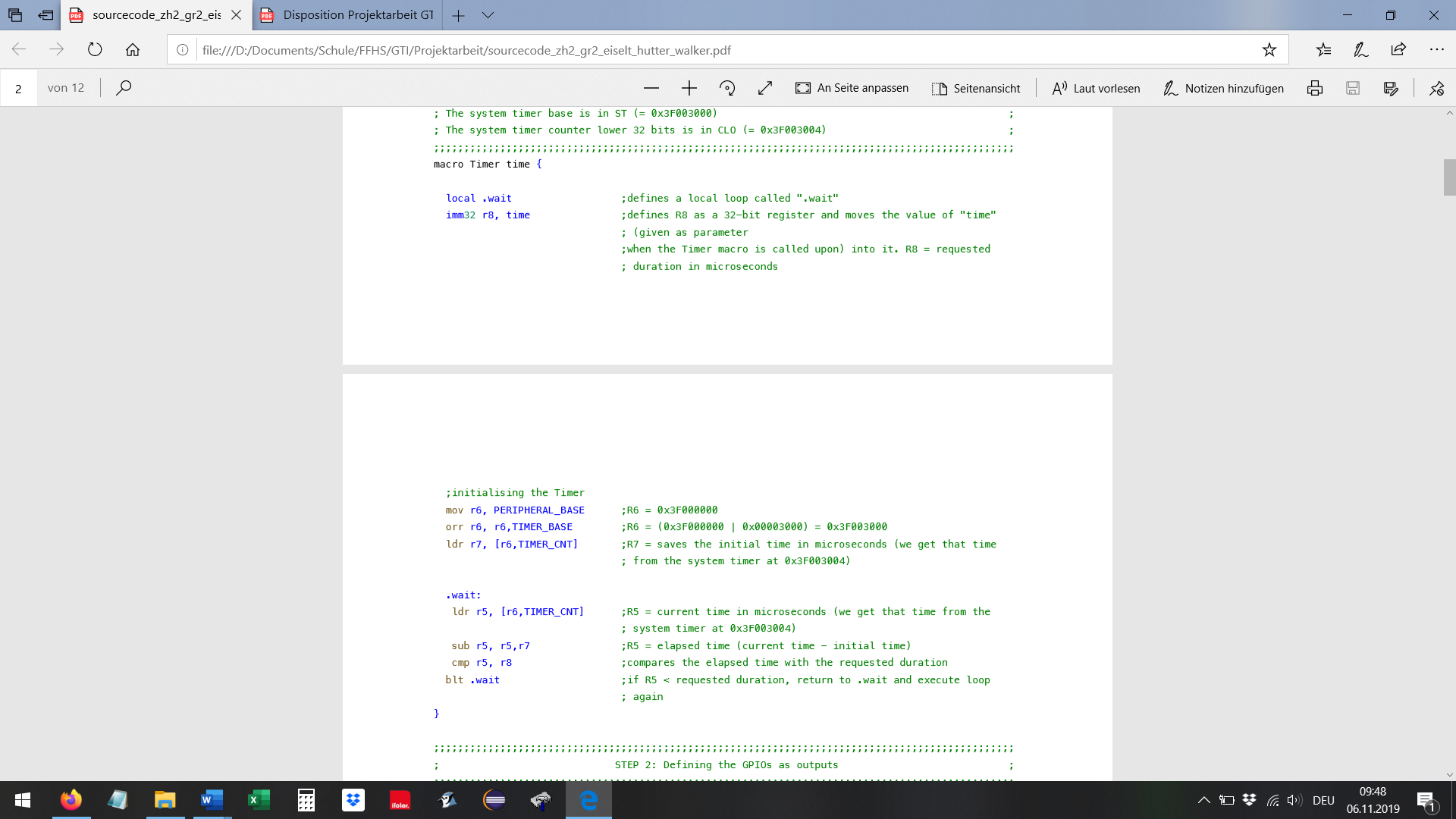
Um die Vorgabe der Taktunabhängigkeit zu erreichen, orientieren wir uns stattdessen am System Timer, welcher unabhängig von der Tatkfrequenz läuft (Broadcom Corporation, 2018). Dessen Basis finden wir bei 0x3F003000 und den Offset des Counters bei 0x4 (hier sind die unteren 32 Bits abgespeichert, wodurch wir hier einen Zähler in Mikrosekunden erhalten). Wie oben beschrieben, definieren wir die beiden zu Beginn als «TIMER\_BASE» resp. «TIMER\_CNT».

In Anlehnung an Philipp (2017) erstellen wir gleich zu Beginn des Codes ein Makro namens «Timer», welcher einen Parameter namens «time» entgegennimmt. Der Wert, der im Programm dem Makro als Parameter übergeben wird, wird in R8 gespeichert und der Startwert des Timers in R7. Anschliessend folgt eine Schleife, welche jeweils den aktuellen Wert des Timers in R5 speichert und anschliessend R5 mit R7 vergleicht – die Differenz der beiden entspricht der verstrichenen Zeit in Mikrosekunden. Sobald die verstrichene Zeit der Vorgabe des Parameters (in R8) entspricht, ist die Ausführung des Makros beendet.

Der Befehlsaufruf des Makros wird dabei absichtlich in Mikrosekunden belassen und nicht in Millisekunden abgeändert. Ursprünglich hatten wir den Timer in Millisekunden definiert, was jedoch zur Folge hatte, dass in einem späteren Codeabschnitt die Audiofrequenz nicht so präzise bestimmt werden konnte (s. auch Kapitel 6.4 und 6.5). Wir haben uns deshalb dazu entschieden, die etwas geringere Benutzerfreundlichkeit beim Aufruf von «Timer» in Kauf zu nehmen, da wir den Vorteil der präzisen Audiofrequenz-Generierung als wichtiger beurteilen.

## Initialisierung der GPIOs

Abbildung 10: Quellcode-Abschnitt des Makros «Timer»



Damit überhaupt je ein Signal über die GPIOs ausgegeben werden kann, müssen wir sie zuerst als Outputs definieren. Jeder GPIO Pin hat drei Bits in einem der «function select registers». Um einen GPIO als Output zu definieren, müssen die drei Bits die Abfolge 001 haben (Broadcom Corporation, 2018).

Wir wollen die LED auf GPIO17 ansteuern und den Lautsprecher auf GPIO18. Da diese beiden Pins im gleichen function select register (nämlich im GPFSEL1) definiert werden, können wir beide gleichzeitig als Outputs setzen. Für GPIO17 benötigen wir die Bits 23-21 und für GPIO18 die Bits 26-24. In den Bits 26-21 muss also die Abfolge (00)1001 stehen. Die Binärzahl 1001 entspricht der Dezimalzahl 9, weshalb nun im R1 #9 zwischengespeichert und anschliessend mit einem logical shift left um 21 Stellen nach links verschoben werden kann. Wenn der Wert in R1 nun ins GPFSEL1 geschrieben wird, sind die beiden Pins als Output definiert.

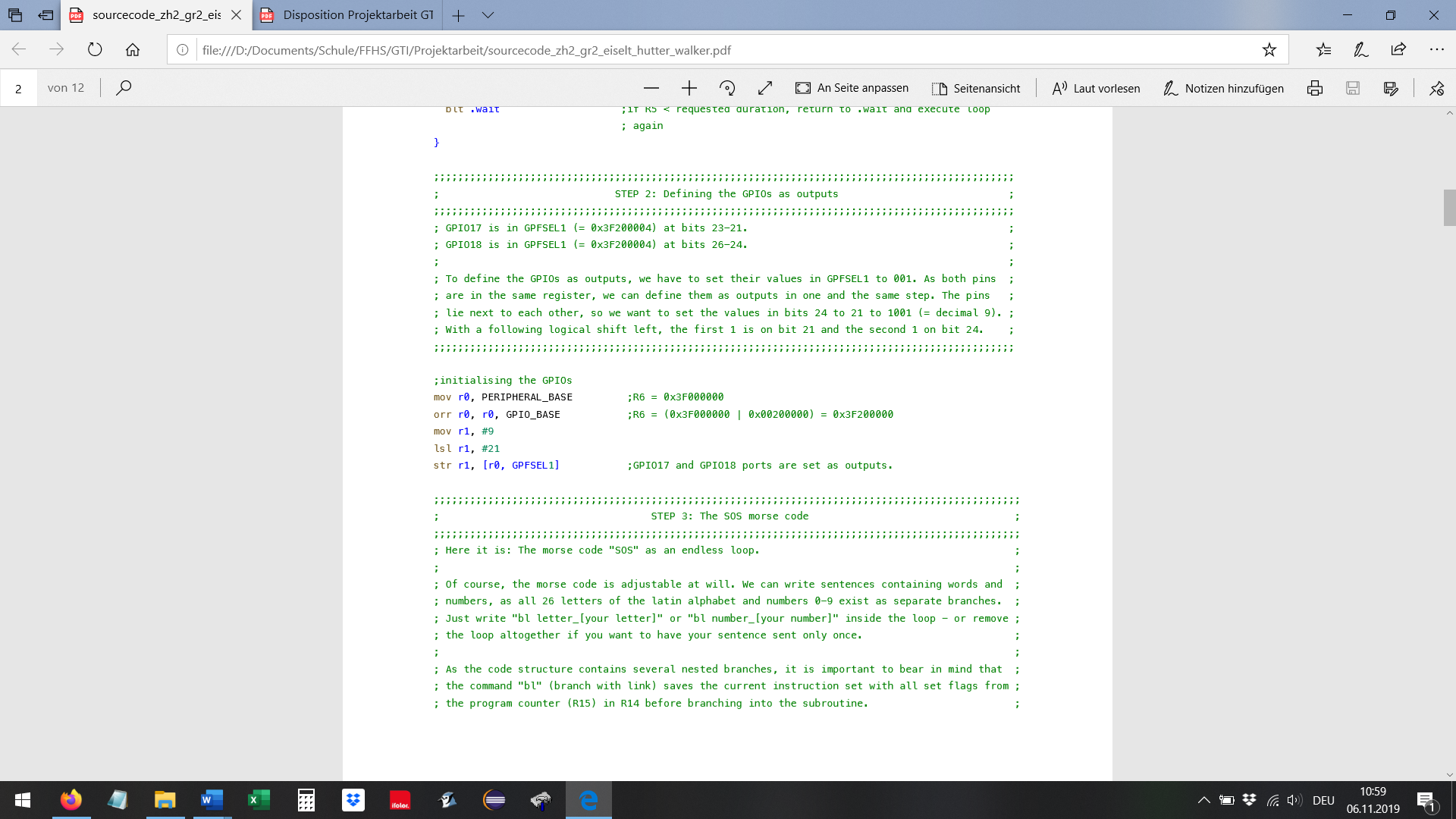


Abbildung 11: Quellcode-Abschnitt der GPIO-Initialisierung

## Generierung des Morse-Signals

Der Code-Abschnitt, welcher ein beliebiges Morse-Signal (nicht nur SOS in einer Endlosschleife) festlegt, soll natürlich möglichst benutzerfreundlich gehalten werden. Wir erreichen dies, indem wir für jeden Buchstaben des lateinischen Alphabets (A-Z) und jede Ziffer (0-9) einen eigenen Branch errichten, welcher «letter\_[Buchstabe]» bzw. «number\_[Ziffer]» heisst. Um einen Abstand zwischen zwei Wörtern zu erfassen, kann der Branch «new\_word» aufgerufen werden. Das eigentliche Programm – die vorgegebene Endlosschleife SOS – kann damit also auf nur sechs Zeilen Programmcode abgehandelt werden.

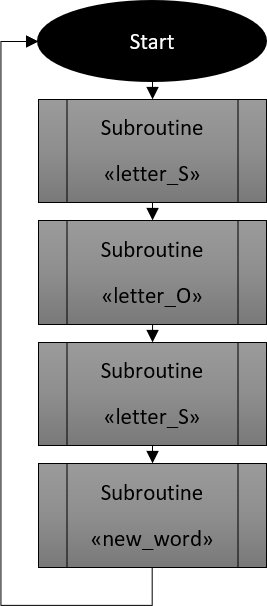


Abbildung : Flowchart SOS-Morsecode

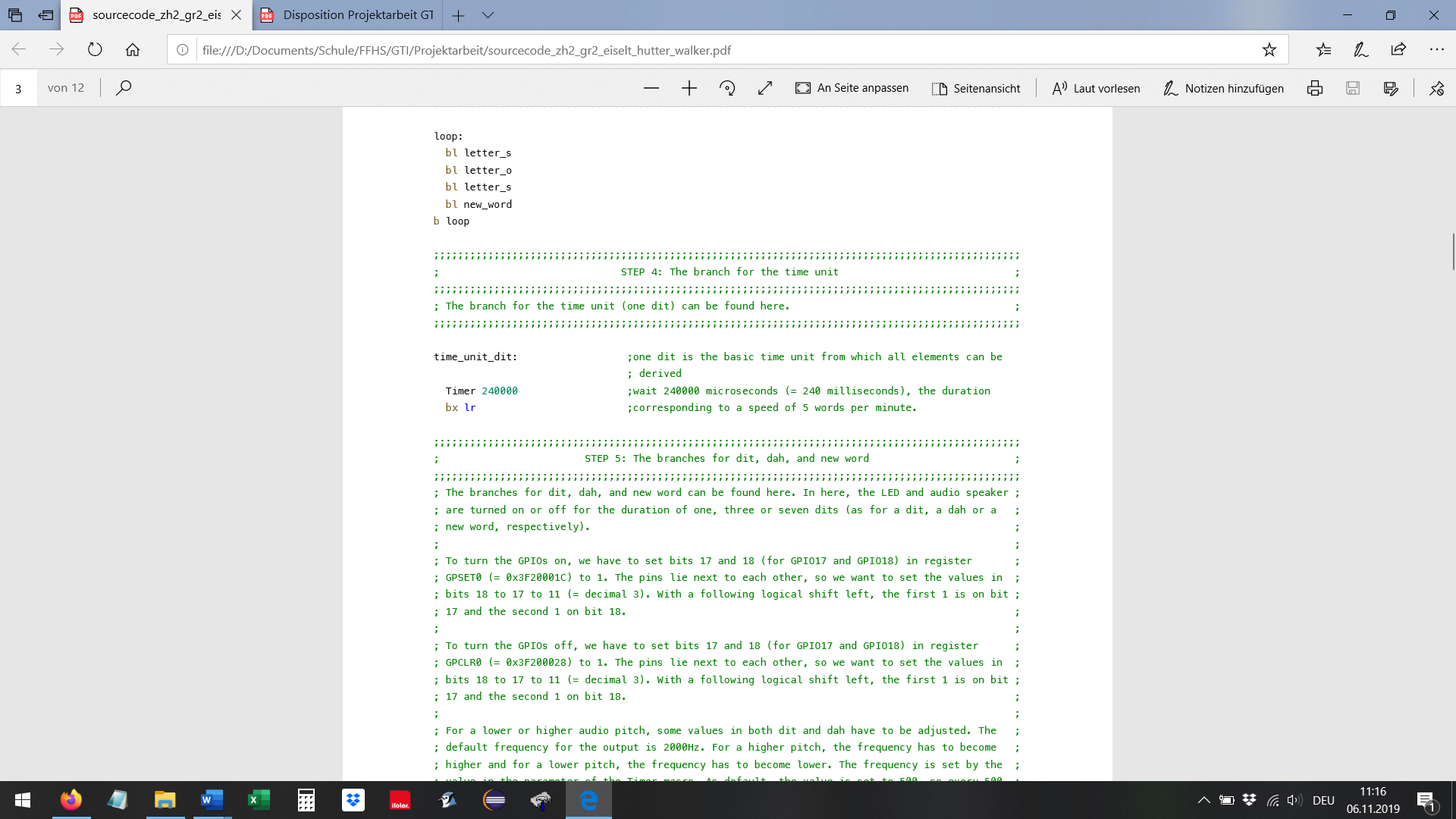


Abbildung 13: Quellcode-Abschnitt der Festlegung des Morse-Signals

Mit der vorliegenden Implementierung der einzelnen Branches könnten bei Bedarf auch weitere Zeichen wie Umlaute oder Sonderzeichen innert kürzester Zeit definiert werden. Um den entsprechenden Code nachvollziehbar zu machen, folgt im nächsten Abschnitt ein Exkurs zum Morse-Alphabet.

### Exkurs: Morse-Alphabet

Beim Morse-Alphabet werden Buchstaben, Ziffern und Sonderzeichen in eine festgelegte Abfolge von kurzen und langen Signalen übersetzt (z.B. A = kurz-lang oder S = kurz-kurz-kurz). Die Signalüber-tragung kann auf unterschiedliche Weise erfolgen, z.B. als Funksignal, Tastsignal oder visuelles Signal.

Jegliche Signallänge sowie jegliche Pause hängt gemäss dem Sektor für Funkkommunikation der internationalen Fernmeldeunion (International Telecommunication Union – Radiocommunication Sector) von der Länge eines kurzen Signals ab (ITU-R, 2009). Dieses wird oft auch «Dit» genannt (s. Phillips, 2018).

* Ein Dit entspricht der Länge eines kurzen Signals.
* Die Länge einer Pause zwischen zwei Signalen innerhalb eines Buchstabens entspricht ebenfalls einem Dit.
* Die Länge eines langen Signals entspricht drei Dits und wird «Dah» genannt.
* Die Länge einer Pause zwischen zwei Buchstaben innerhalb eines Wortes entspricht ebenfalls drei Dits.
* Die Länge einer Pause zwischen zwei Wörtern entspricht sieben Dits.

Wird ein Buchstabe als Abfolge von Dits und Dahs geschrieben, wird manchmal ein Dit mit «Di» für ein kurzes Signal, welches innerhalb, jedoch nicht am Ende des Buchstabens erscheint, verwendet und «Dit» für ein kurzes Signal am Ende des Buchstabens. Der Buchstabe A kann also als «di dah» und der Buchstabe S als «di di dit» geschrieben werden (Phillips, 2018). Diese Schreibweise wird in der vorliegenden Dokumentation ebenfalls verwendet.

Wie lange ein Dit tatsächlich dauert, kann mit der Einheit «words per minute» (WPM) berechnet werden. Das Standardwort für die Berechnung lautet PARIS, welches aus 50 Zeiteinheiten (Dits) besteht (s. untenstehende Tabelle zur Verdeutlichung).

Wir wollen 5 WPM erreichen. Dies ist eine ziemlich langsame Sendegeschwindigkeit, welche es uns als Morse-Laien gut ermöglicht, die Buchstaben zu identifizieren. Wenn wir also in einer Minute fünfmal das Wort PARIS senden (= 5 WPM), haben wir inkl. Pausen insgesamt 250 Dits. 60 Sekunden dividiert durch 250 Dits ergibt 0.24 Sekunden (= 240 Millisekunden / 240'000 Mikrosekunden) pro Dit.

Tabelle 2: Berechnung der nötigen Zeiteinheiten für das Wort PARIS

|  |  |  |  |
| --- | --- | --- | --- |
| Buchstabe | Morse-Signal | Zeiteinheiten (Dits) | Zeiteinheiten (Dits) Total |
| P | di dah dah dit | 1 (1) 3 (1) 3 (1) 1 [3] | 14 Dits |
| A | di dah | 1 (1) 3 [3] | 8 Dits |
| R | di dah dit | 1 (1) 3 (1) 1 [3] | 10 Dits |
| I | di dit | 1 (1) 1 [3] | 6 Dits |
| S | di di dit | 1 (1) 1 (1) 1 {7} | 12 Dits |
| **Total** |  |  | **50 Dits** |
| ( ) = Pause innerhalb eines Buchstabens (zwischen zwei Signalen)  [ ] = Pause innerhalb eines Wortes (zwischen zwei Buchstaben)  { } = Pause zwischen zwei Buchstaben | | | |

### Praktische Anwendung der Morse-Code-Richtlinien

Wie im Exkurs zum Morse-Alphabet beschrieben, ist also vor allem die Definition der Zeiteinheit Dit zentral. Im vorliegenden Code werden wir dieser Anforderung gerecht, indem wir ihr einen eigenen Branch zuweisen, welcher das Makro «Timer» entsprechend lange aufruft. Da wir oben eine gewünschte Dauer von 240 Millisekunden pro Dit definiert haben, der Timer jedoch mit Mikrosekunden arbeitet, muss also ein Parameter-Wert von 240000 übergeben werden.

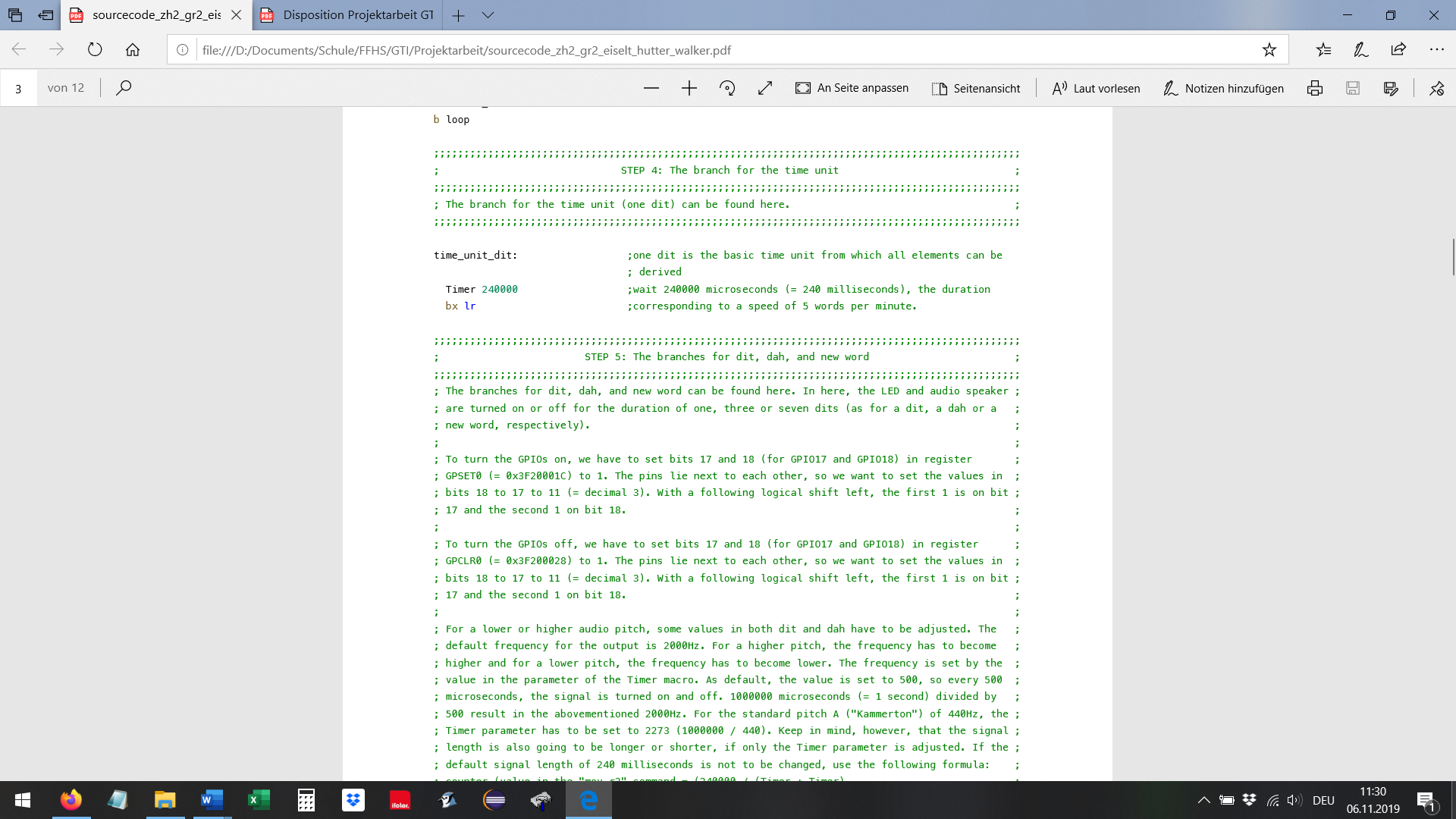


Abbildung 14: Quellcode-Abschnitt des Branches für die Zeiteinheit Dit

Die «time\_unit\_dit» kann nun dafür gebraucht werden, um alle restlichen Branches des Codes zu definieren, denn diese greifen alle wieder auf diesen zentralen Branch zurück. Damit dies aber überhaupt funktionieren kann, muss bei jedem dieser Branches zuerst die Rücksprungadresse zwischengespeichert werden. Wenn nämlich in die Subroutine «time\_unit\_dit» abgezweigt wird, wird der aktuelle Programmzähler aus R15 in R14 gespeichert (Cockerell, o.J.). Der letzte Befehl «bx lr» in «time\_unit\_dit» ermöglicht es dem Programm, zurück zum Stamm zu finden, von wo aus der Branch aufgerufen wurde. Innerhalb aller Branches, welche einen anderen Branch mit «bl» aufrufen, wird also die eigentliche Rücksprungadresse R14 überschrieben. Deshalb ist es unabdingbar, vor der Abzweigung in irgendwelche Subroutinen die Rücksprungadresse zwischenzuspeichern. Wir verwenden dafür die Register R12 (für alle Buchstaben und Zahlen sowie für ein neues Wort) und R11 (für Dit und Dah, da diese sowohl selber eine Subroutine aufrufen, als auch von einem Stamm aufgerufen werden – also nochmals eine Ebene weiter innen verschachtelt liegen).

Sowohl der Branch «dit» als auch der Branch «dah» werden so implementiert, dass anschliessend zuerst die Basis für Ein- und Ausschalten der GPIOs geschaffen wird. Um GPIO17 und GPIO18 einzuschalten, müssen die Bits 17 und 18 im oben definierten GPSET0 auf 1 gesetzt werden und um sie auszuschalten, müssen die Bits 17 und 18 in GPCLR0 auf 1 gesetzt werden. Wir wollen also beide Male die Abfolge 11 auf den Bits 18-17. Die Binärzahl 11 entspricht der Dezimalzahl 3, weshalb wir in R1 #3 zwischenspeichern und anschliessend mit einem logical shift left um 17 Stellen nach links verschieben. Nun kann R1 je nach Anforderung mit GPSET0 oder GPCLR0 verknüpft werden, um die GPIOs ein- oder auszuschalten.

Es folgt eine Schleife, welche innerhalb von 240 (Dit) bzw. 720 (Dah) Durchgängen die GPIOs für jeweils 500 Mikrosekunden ein- und wieder ausschaltet. Hier liegt auch der einzige Unterschied zwischen den Branches «dit» und «dah», ansonsten sind sie identisch. Der Grund, weshalb die GPIOs innerhalb der definierten Zeit von 240 Millisekunden so oft ein- und ausgeschaltet werden – und nicht einfach 240 Millisekunden lang ein, gefolgt von 240 Millisekunden lang aus –, liegt in der Anforderung für die Audiogenerierung. Darauf wird in Kapitel 6.5 noch genauer eingegangen.

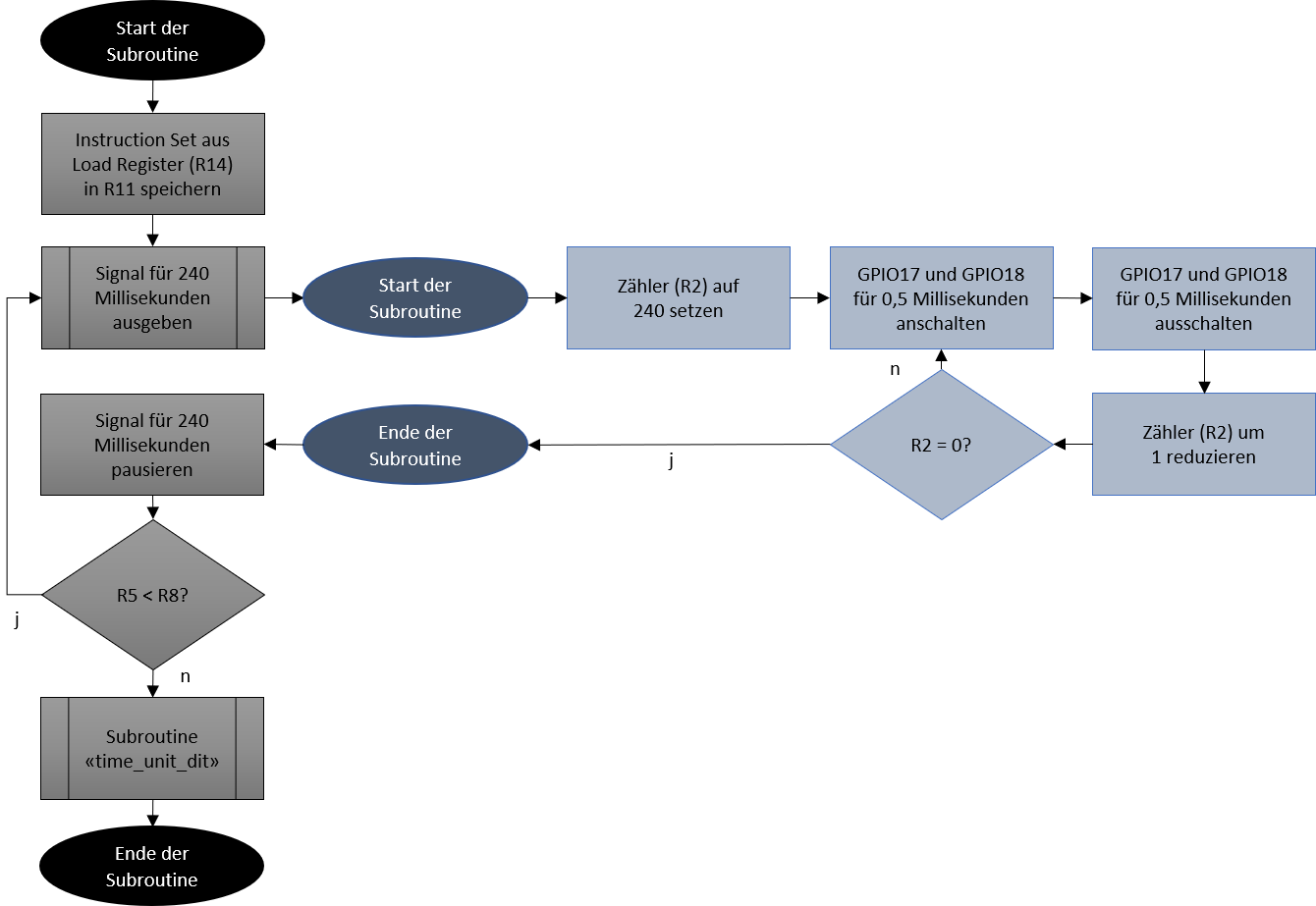


Abbildung 15: Flowchart des Branches für ein Dit

Nach Verlassen der Schleife wird am Schluss jedes Dit oder Dah nochmals eine «time\_unit\_dit» eingeschoben, in welcher mit ausgeschaltetem GPIO gewartet wird. Dies dient der oben erwähnten Pause innerhalb eines Buchstabens zwischen zwei Signalen.

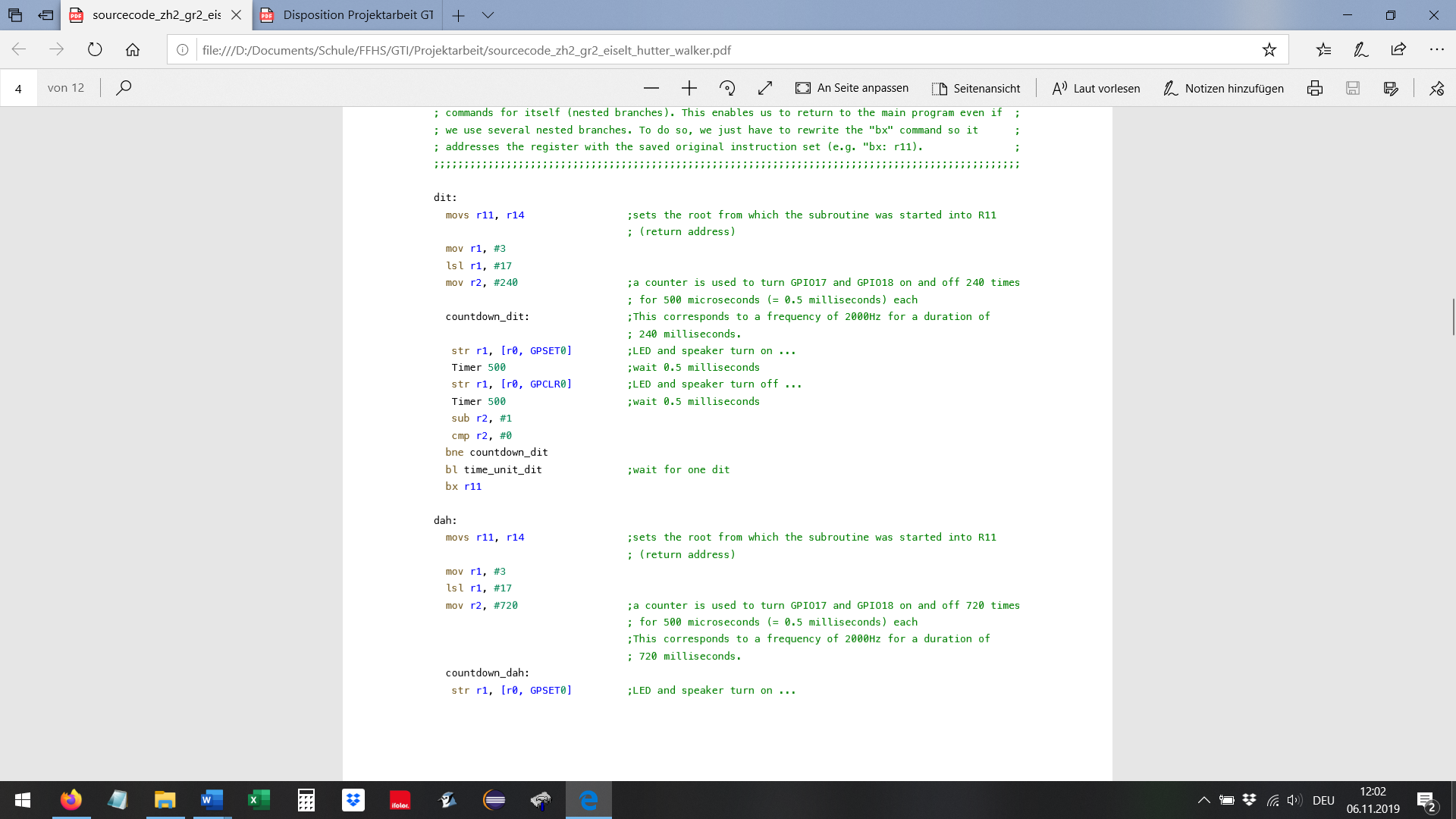


Abbildung 16: Quellcode-Abschnitt des Branches für ein Dit

Mit diesen Erklärungen kann nun auch die Implementierung der Branches für die Buchstaben und Zahlen nachvollzogen werden. Für jedes Zeichen werden einfach so viele «dit» oder «dah» aufgerufen, wie seiner Signalfolge im Alphabet entsprechen. Am Schluss jedes Zeichens wird zusätzlich noch eine Wartezeit von zwei Dits angehängt, damit die Vorgabe von drei Dits zwischen zwei Zeichen eingehalten wird (ein Dit Wartezeit ist ja bereits in «dit» und «dah» enthalten).

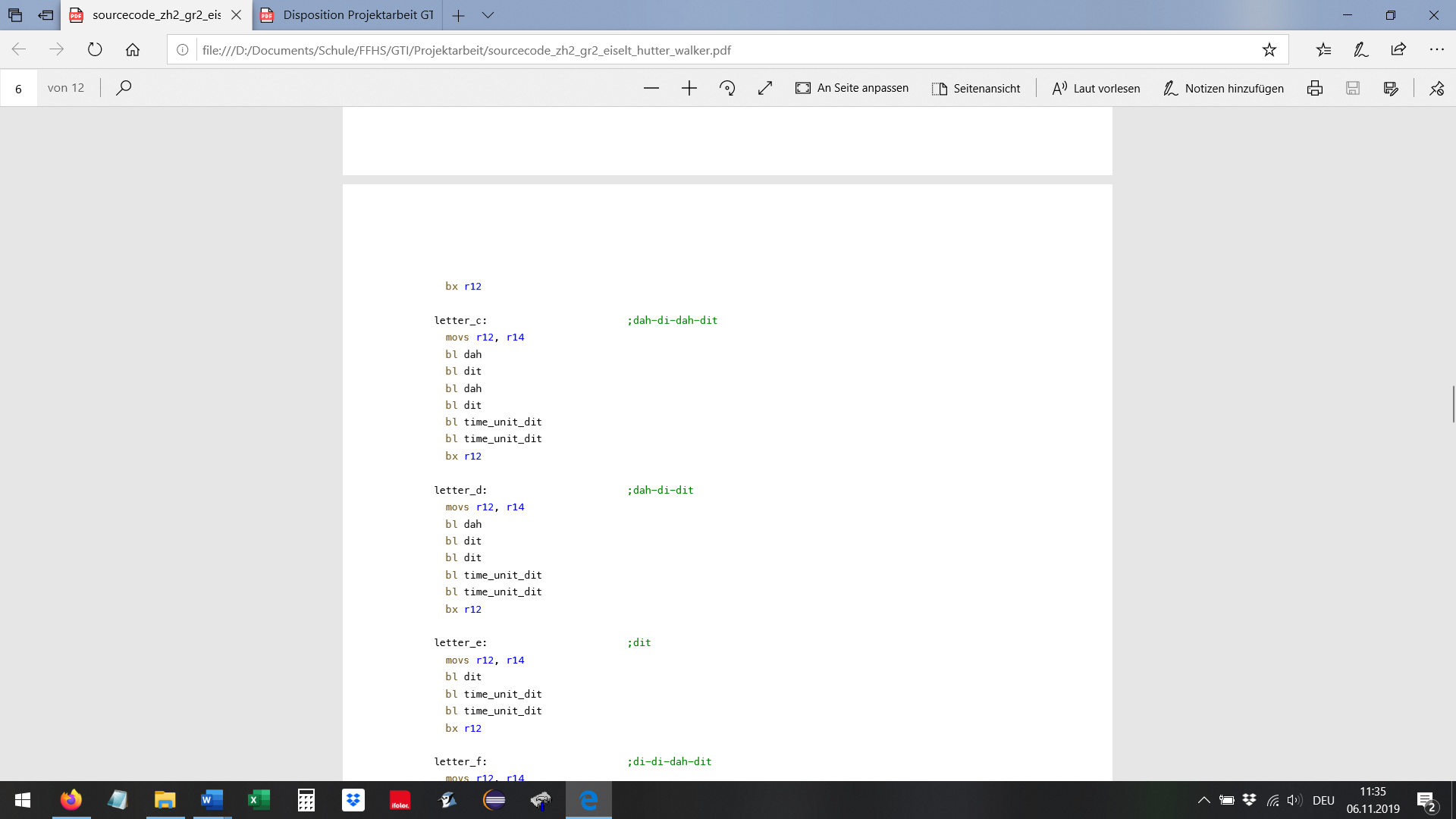


Abbildung 17: Quellcode-Abschnitt der Branches für Buchstaben und Zahlen (Bsp. Buchstabe «C»)

Ein neues Wort muss gemäss Definition nach einer Wartezeit von sieben Dits erfolgen. Wie soeben aufgezeigt, enthält jeder Buchstabe bereits eine Wartezeit von drei Dits, sodass im Branch für ein neues Wort nur noch viermal die «time\_unit\_dit» aufgerufen werden muss. Dies geschieht mittels einer Schleife – wobei es natürlich genauso gut möglich wäre, auf die Schleife zu verzichten und nach dem Speichern der Rücksprungadresse einfach viermal «bl time\_unit\_dit» einzufügen.

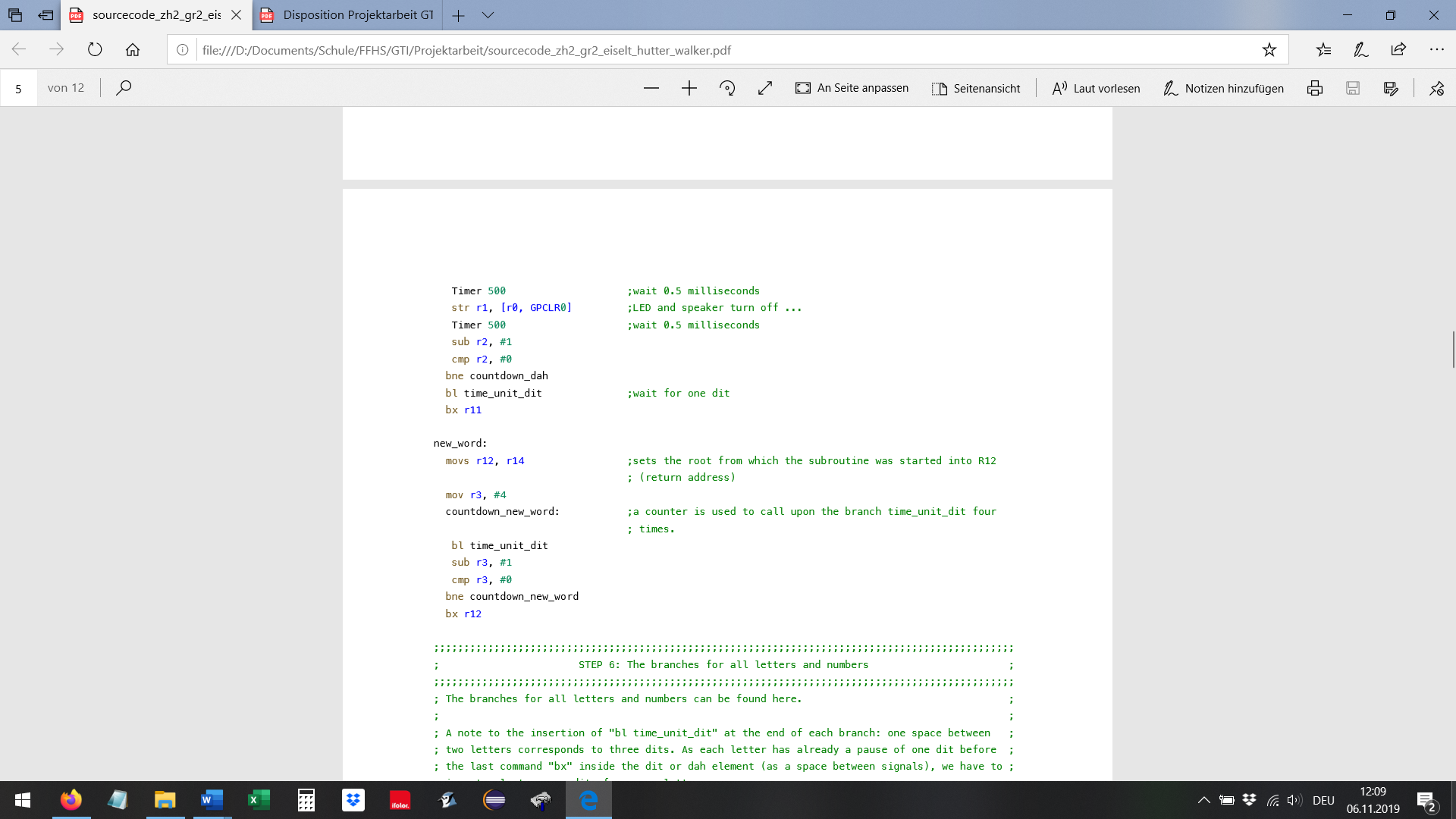


Abbildung 18: Quellcode-Abschnitt des Branches für ein neues Wort

## Besonderheiten des Audiosignals

Wenn man an einen der beiden GPIOs einen Lautsprecher anstatt ein LED anschliesst und das Programm wie in den LED-Tutorials (z.B. FFHS, 2019a[[2]](#footnote-2)) beschrieben laufen lässt, hört man aus dem Lautsprecher nur ein leises Klicken beim Ein- und Ausschalten. Dies ist auf die zu geringe Frequenz zurückzuführen: damit ein Ton generiert werden kann, muss die Sendefrequenz um ein Vielfaches erhöht werden.

### Ursprünglicher Lösungsansatz: PWM

Zuerst wollten wir GPIO18 als Output mit PWM (pulse width modulation) ansteuern. Dazu muss man die «alternate function 5» des GPIO18 nutzen. Das PWM würde es ermöglichen, ein Signal mit einer benutzerdefinierten Frequenz auszugeben, wann immer GPIO18 eingeschaltet wird.

Dieses Vorgehen hat beim Ausprobieren in der Konsole bestens funktioniert (s.a. Smart Power Electronics, 2018). Bei der Bare Metal Implementierung mit Assembler sind wir jedoch gescheitert, als es um die Festlegung der Frequenz ging. Die BCM-Dokumentation wirft diesbezüglich wieder einmal mehr Fragen auf, als sie beantwortet. Beispielsweise steht zuoberst auf S. 141 «PWM clock source and frequency is controlled by CPRMAN», wobei die Referenz «CPRMAN» an keiner anderen Stelle im gesamten Dokument zu finden ist (Broadcom, 2018, S. 141). Auch Online-Foren bieten keine endgültig befriedigenden Antworten. So definiert z.B. Mokutsuno (2017) die CLOCK\_BASE bei 0x3F101000 und schreibt dann als Kommentar «undocumented. Extrapolated from RPI\_V1 CLOCK\_BASE».

Wir vermuten, dass das generelle Vorgehen in etwa wie folgt aufgebaut sein sollte (in Anlehnung an Pyeatt, 2016):

1. GPFSEL1 (Bits 24-26) auf 010 statt 001 setzen = GPIO Pin 18 takes alternate function 5
2. PWM\_CTL (0x3F20C000) wie folgt definieren:
   1. PWEN1 (Bit 0) auf 1 setzen = Channel 1 enabled
   2. MODE1 (Bit 1) auf 0 setzen = PWM mode
   3. USEF1 (Bit 5) auf 0 setzen = Data register is transmitted
   4. MSEN1 (Bit 7) auf 1 setzen = M/S transmission is used (mark space mode anstatt balanced mode)
3. PWM\_RNG1 (Offset 0x10) für die Basisfrequenz definieren.
4. PWM\_DAT1 (Offset 0x14) für den Betriebszyklus auf einen Wert zwischen 0 und 1000 setzen.
5. CLK auf den folgenden Adressen ansteuern (Passwort erforderlich):
   1. CLK\_BASE (0x3F101000)
   2. CLK\_CTL (Offset 0xA0)
   3. CLK\_DIV (Offset 0x4)
   4. CLK\_PASSWD (0x5A000000)

Als sich jedoch auch nach mehrtägiger Recherche und Ausprobieren immer noch kein Erfolg einstellte, entschlossen wir uns dazu, für die Implementierung des Audiosignals einen anderen Lösungsansatz zu erarbeiten.

### Verwendeter Lösungsansatz: Frequenzgenerierung innerhalb «dit» und «dah»

Wie erwähnt, benötigt der Lautsprecher eine um ein Vielfaches höhere Sendefrequenz als die LED. Da es uns nicht gelang, die beiden Sendefrequenzen zu entkoppeln, wird im vorliegenden Code die LED nun mit derselben Frequenz gesteuert wie der Lautsprecher. Da das menschliche Auge nicht dazu imstande ist, eine so schnelle Abfolge von Ein- und Ausschalten der LED wahrzunehmen, erscheint es, als ob die LED durchgehend leuchten würde. Demnach wird die Lösung der vorgegebenen Aufgabe durch diese Vorgehensweise nicht beeinträchtigt.

Die Tonhöhe wird demnach innerhalb der Branches «dit» und «dah» festgelegt. Die Masseinheit für die Frequenz lautet Hertz (Hz), wobei ein Hz einem Signal pro Sekunde entspricht. Als Standardfrequenz wurde eine Frequenz von 2000Hz gewählt, d.h. es werden 2000 Signale pro Sekunde gesendet (zu den Überlegungen hierzu, s. Kapitel 6.5.3). Soll die Frequenz angepasst werden, muss zum einen die Parameterübergabe an den «Timer» innerhalb von «countdown\_dit» bzw. «countdown\_dah» geändert werden. Die Formel für die Berechnung der Frequenz lautet wie folgt:

Sendefrequenz = 1'000'000 / Timer

Eine Sekunde hat 1 Million Mikrosekunden. Aktuell werden in unserem Code dem Timer als Parameter 500 Mikrosekunden übergeben. In der oberen Formel können wir also für unsere Frequenz einsetzen: 1'000'000 / 500 = 2'000, womit wir die erwähnte Standardfrequenz 2000Hz erhalten.

Nun befindet sich die Frequenz jedoch innerhalb einer Schleife, welche die Zeitdauer des Signals festlegt. Wird beispielsweise die Frequenz durch Setzen des Timers auf 250 auf 4000Hz verdoppelt, wird die Signallänge auf die Hälfte verkürzt, denn die Schleife wird ja weiterhin 240 (Dit) bzw. 720 (Dah) mal durchlaufen. Eine Schleife dauert jetzt aber nicht mehr 1000 (500 + 500) Mikrosekunden, sondern nur noch 500 (250 + 250) Mikrosekunden. Somit passt also die Signallänge nicht mehr zu den Pausen, welche mit der «time\_unit\_dit» festgelegt werden. Um beim Beispiel mit 4000Hz zu bleiben: soll die Signallänge weiterhin 240 Millisekunden betragen, muss der Zähler der Schleife verdoppelt werden, also 480 (Dit) bzw. 1440 (Dah) betragen. Wenn die Standard-Signallänge von 240 Millisekunden also nicht verändert werden soll, können die folgenden beiden Formeln für die Berechnung des Zählers herangezogen werden:

Zähler in R2 vor «countdown\_dit» = 240'000 / (Timer + Timer)

Zähler in R2 vor «countdown\_dah» = 3 x Zähler in R2 vor «countdown\_dit»

So wurden auch die aktuellen Zählerwerte berechnet: 240'000 / (500 + 500) = 240 (Zähler in R2 vor «countdown\_dit») sowie 3 x 240 = 720 (Zähler in R2 vor «countdown\_dah»).

### Exkurs: Frequenzbewertung / dB(A)

Der Output unseres Miniatur-Lautsprechers ist ohne Verstärker nicht besonders laut. Um trotzdem ein gut hörbares Audiosignal zu generieren, haben wir die menschliche Frequenzbewertung einberechnet.

Das menschliche Gehör empfindet nicht alle Tonhöhen als gleich laut. Um sehr hohe oder sehr tiefe Töne wahrzunehmen, brauchen wir einen weitaus höheren Schalldruck als für Töne mittlerer Frequenzen (Akustika, 2019). Eine sogenannte Frequenzbewertungskurve zeigt auf, für welche Frequenzen das menschliche Ohr am empfindlichsten ist. Dabei wird meist das Mass dB(A) verwendet, um eine Lautstärke unter Berücksichtigung der frequenzabhängigen menschlichen Gehörempfindung anzugeben (BG BAU, o.J.). Beim Betrachten der Frequenzbewertungskurve dB(A) ist zu erkennen, dass die höchste Empfindlichkeit des menschlichen Gehörs zwischen 2000Hz und 4000Hz liegen muss.

Für unser Audiosignal haben wir aufgrund dieser Überlegungen eine Frequenz von 2000Hz gewählt. Diese Frequenz wird als lauter bewertet als tiefere Frequenzen, ist aber doch noch tief genug, dass die Tonhöhe immer noch als angenehm empfunden wird. Zudem ist der entsprechende Abschnitt im Code mit der gewählten Frequenz sehr einfach zu lesen, da die «Dit»-Schleife gleich auf 240 Durchgänge gesetzt werden kann, um einer Dauer von 240 Millisekunden zu entsprechen.

# Reflexion der Arbeit

## Eveline Hutter

Ich fand den Einstieg in die Projektarbeit extrem schwierig. Vor allem die Adressierung der Komponenten bereitete mir Mühe und ich fand trotz zahlreicher Tutorials (u.a. Immler, 2018; Kampert, 2019; Thiebaut, 2015) nicht eine einfache Grundlage, auf welche ich aufbauen konnte. Die beiden im Moodle (FFHS, 2019b)[[3]](#footnote-3) vorgeschlagenen ARM-Dokumentationen (ARM Limited 2014, 2019a) erschlagen einen Neuling komplett, sodass man überhaupt nicht weiss, was man damit anfangen soll. Dass die Dokumentation (Broadcom Corporation, 2018) lückenhaft – und bei der Adressierung sogar fehlerhaft – ist, tat sein Weiteres dazu. Erst als ich im Unterricht in PVA3 von einem Mitstudenten die Adressierung Schritt für Schritt erklärt bekam, konnte ich wirklich mit dem Programm starten.

Die Programmierung selbst machte nach dieser ersten grossen Hürde richtig Spass. Ich fand es interessant, verschiedene Vorgehensweisen auszuprobieren und den Code nach und nach zu verbessern, sodass die endgültige Version die höchst mögliche Benutzerfreundlichkeit aufweist, sollte jemand Aussenstehendes den Code für sich verwenden und evtl. abändern wollen. Da ich bislang noch keinerlei Programmiererfahrung hatte, war auch die Kommentierung eine neue Erfahrung für mich.

Der einzige Makel des Programms liegt meiner Meinung nach darin, dass der Code an mehreren Orten geändert werden muss, wenn man die Frequenz und/oder die Zeiteinheit für ein Dit verändern will: im Branch «dit» und im Branch «dah» - sowie im Branch «time\_unit\_dit», wenn die Zeiteinheit geändert werden soll). Dies kann leider mit der gewählten Herangehensweise der Frequenzgenerierung innerhalb der Branches «dit» und «dah» nicht benutzerfreundlicher gelöst werden. Wenn es uns gelungen wäre, die Frequenz mittels PWM zu setzen, müsste für eine Änderung der Basis-Zeiteinheit nur der Branch «time\_unit\_dit» geändert werden und für eine Änderung der Frequenz nur der entsprechende PWM-Abschnitt. An der PWM-Implementierung nach etlichen investierten Stunden zu scheitern, finde ich sehr enttäuschend.

## Franziska Walker

Ähnlich wie Eveline ist mir der Einstieg in das Thema schwer gefallen. Trotz Investition von viel Zeit, bin ich erst auf einen grünen Zweig gekommen mit der Hilfe aus den Inputs in der Klasse. Einmal begriffen, wie die Assembly-Sprache funktioniert, ist es gar nicht so schwer.

Ich freue mich mit Eveline und Christian in der Gruppe zu sein und bin überzeugt, wir profitieren gegenseitig voneinander.

Rückblickend habe ich bei dieser Arbeit viel gelernt. Allerdings wäre es mir leichter gefallen mit klaren, einführenden Hinweisen.

# Literaturverzeichnis

Akustika (2019). Wie hört der Mensch. Abgerufen am 5. November 2019 von: [http://www.  
akustika.ch/richtig-hoeren/rund-ums-richtige-hoeren/gehoer/wie-hoert-der-mensch/](http://www.akustika.ch/richtig-hoeren/rund-ums-richtige-hoeren/gehoer/wie-hoert-der-mensch/).

Android Wiki (2016). ARM (ARM-Architektur). Abgerufen am 30. Oktober 2019 von: <https://www.droidwiki.org/wiki/ARM>.

ARM Limited (2014). ARM Cortex-A53 MPCore Processor. Revision: r0p2. Technical Reference Manual. Abgerufen am 6. November 2019 von: [http://infocenter.arm.com/help/topic/com.arm.doc  
.ddi0500d/DDI0500D\_cortex\_a53\_r0p2\_trm.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf).

ARM Limited (2017). The A64 instruction set. Abgerufen am 30. Oktober 2019 von: <https://static.docs.arm.com/100898/0100/the_a64_Instruction_set_100898_0100.pdf>.

ARM Limited (2019a). Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile. Abgerufen am 6. November 2019 von: [https://static.docs.arm.com/ddi0487/ea/DDI0487E\_a\_  
armv8\_arm.pdf?\_ga=2.165976510.1414297968.1572975597-185487487.1568361410](https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf?_ga=2.165976510.1414297968.1572975597-185487487.1568361410).

ARM Limited (2019b). Arm CPU Architecture. One architecture, many capabilities. Abgerufen am 30. Oktober 2019 von: <https://developer.arm.com/architectures/cpu-architecture>.

ARM Limited (2019c). Cortex-A53. The Cortex-A53 processor is a high efficiency processor that implements the ARMv8-A architecture. Abgerufen am 30. Oktober 2019 von: [https://developer  
.arm.com/ip-products/processors/cortex-a/cortex-a53](https://developer.arm.com/ip-products/processors/cortex-a/cortex-a53).

ARM Limited (2019d). 2.2.9 General purpose registers. Abgerufen am 6. November 2019 von: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.100026_0102_00_en/jfa1411296377566.html>.

ARM Limited (2019e). Instruction Set Architecture. Abgerufen am 30. Oktober 2019 von: <https://developer.arm.com/architectures/instruction-sets>.

BG BAU (o.J.). Frequenzbewertung, Spitzen- und Dauerschallpegel. Abgerufen am 5. November 2019 von: [https://www.bgbau.de/themen/sicherheit-und-gesundheit/laerm-und-vibrationen/freq  
uenzbewertung-spitzen-und-dauerschallpegel/](https://www.bgbau.de/themen/sicherheit-und-gesundheit/laerm-und-vibrationen/frequenzbewertung-spitzen-und-dauerschallpegel/).

Broadcom Corporation (2018). BCM2837 ARM Peripherals. V2.1. FA (Ed.). Abgerufen am 5. November 2019 von: <https://github.com/raspberrypi/documentation/files/1888662/BCM2837-ARM-Peripherals.-.Revised.-.V2-1.pdf>.

Cockerell, P. (o.J.). ARM Assembly Language Programming. Abgerufen am 13. September 2019 von: <http://www.peter-cockerell.net/aalp/html/frames.html>.

DashCam Bros (2017). Format Memory Card to FAT32. Abgerufen am 8. November 2019 von: <https://dashcambros.com/pages/format-memory-card-to-fat32>.

FASMARM (Version 1.43) [Computer Software]. Tomasz Grysztar. Abgerufen am 8. November 2019 von: <https://arm.flatassembler.net/FASMARM_win32.ZIP>.

FFHS (2019a). FASMARM (Assembler, Alternative zur «GNU ARM ToolChain») [Moodle Eintrag]. Abgerufen am 8. November 2019 von: [https://moodle.ffhs.ch/mod/book/view.php?id=312320  
7&chapterid=43574](https://moodle.ffhs.ch/mod/book/view.php?id=3123207&chapterid=43574).

FFHS (2019b). GTI, Grundlagen der technischen Informatik, BSc INF 2019, ZH2, HS19/20, Dr. Einsele Farsideh. Grundlagen der Technischen Informatik. Projektarbeit GTI (Auftrag, Informationen, Dokumente). Technische Unterlagen zum ARMv8 [Moodle Eintrag]. Abgerufen am 8. November 2019 von: <https://moodle.ffhs.ch/course/view.php?id=6491#section-8>.

Fritzing (Version 0.9.3b) [Computer Software]. Berlin: Friends-Of-Fritzing e.V. Abgerufen am 9. November 2019 von: <http://fritzing.org/download/>.

Immler, C. (2018). FRANZIS Raspberry Pi Maker Kit Handbuch. Haar: Franzis Verlag GmbH.

ITU-R (2009). Recommendation ITU-R M.1677-1 (10/2009). International Morse code. Abgerufen am 30. Oktober 2019 von: <http://www.itu.int/dms_pubrec/itu-r/rec/m/R-REC-M.1677-1-200910-I!!PDF-E.pdf>.

Kampert, D. (2019). Einstieg in die ARMv6 Programmierung. Abgerufen am 6. November 2019 von: <https://www.kampis-elektroecke.de/raspberry-pi/armv6/>.

Mokutsuno (21. April 2017). How can I control GPIO by using DMA in Raspberry pi 3? [Web Log Eintrag]. Abgerufen am 5. November 2019 von: [https://www.raspberrypi.org/forums/  
viewtopic.php?p=1155194](https://www.raspberrypi.org/forums/viewtopic.php?p=1155194).

Philipp, M. (30. Juni 2017). Using the Raspberry Pi Timer for Embedded Environments [Web Log Eintrag]. Abgerufen am 6. November 2019 von: <https://www.studica.com/blog/raspberry-pi-timer-embedded-environments>.

Phillips, S.C. (2018). International Morse Code. Abgerufen am 30. Oktober 2019 von: <https://morsecode.scphillips.com/>.

Pyeatt, L.D. (2016). *Modern Assembly Language Programming with the ARM Processor*. Oxford, England and Cambridge, MA: Newnes, Elsevier.

Raspberry Pi Foundation (2018). Annual Review 2018. Abgerufen am 7. November 2019 von: <https://static.raspberrypi.org/files/about/RaspberryPiFoundationReview2018.pdf>.

Raspberry Pi Foundation (o.J.). GPIO. Abgerufen am 6. November 2019 von: <https://www.raspberrypi.org/documentation/usage/gpio/README.md>.

Schnabel, P. (2019a). LED – Leuchtdioden. Abgerufen am 6. November 2019 von: <https://www.elektronik-kompendium.de/sites/bau/0201111.htm>.

Schnabel, P. (2019b). Masse, Erde und Ground. Abgerufen am 9. November 2019 von: https://www.elektronik-kompendium.de/sites/grd/2206201.htm.

Smart Power Electronics (10. Mai 2018). #05 Setting the GPIO hardware PWM Frequency on the Raspberry Pi [YouTube Video]. Abgerufen am 5. November 2019 von: [https://www.you  
tube.com/watch?v=Zjr69DxgeW0](https://www.youtube.com/watch?v=Zjr69DxgeW0).

Storm I. (2018). Der neue Rasberry Pi 3B+ - schneller zum gleichem Preis. Abgerufen am 7. November 2019 von: [https://www.heise.de/ct/artikel/Der-neue-Raspberry-Pi-3B-schneller-zum-gleichem  
-Preis-3994036.html](https://www.heise.de/ct/artikel/Der-neue-Raspberry-Pi-3B-schneller-zum-gleichem-Preis-3994036.html).

Thiebaut, D. (2015). Tutorial: Assembly Language with the Raspberry Pi. Abgerufen am 6. November 2019 von: [http://www.science.smith.edu/dftwiki/index.php/Tutorial:\_Assembly\_Language\_  
with\_the\_Raspberry\_Pi](http://www.science.smith.edu/dftwiki/index.php/Tutorial:_Assembly_Language_with_the_Raspberry_Pi).

Wikipedia-Autoren (2019). *Bare Machine*. In Wikipedia, die freie Enzyklopädie. Abgerufen am 7. November 2019 von: <https://en.wikipedia.org/w/index.php?title=Bare_machine&oldid=924972756>.

# Anhang

## Quellcode: kernel.asm

PERIPHERAL\_BASE = $3F000000    ;defines a variable for addressing the peripheral base

GPIO\_BASE =  $200000           ;defines a variable for addressing the base of the GPIOs

GPFSEL1 = $4                   ;defines a variable for addressing GPIO function select register 1

GPSET0 = $1C                   ;defines a variable for addressing GPIO pin output set 0

GPCLR0 = $28                   ;defines a variable for addressing GPIO pin output clear 0

TIMER\_BASE = $3000             ;defines a variable for addressing the base of the system timer

TIMER\_CNT = $4                 ;defines a variable for addressing the lower 32 bits of the system

                               ; timer counter

format binary as 'img'

include 'LIB\FASMARM.INC'

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;                                USED GENERAL PURPOSE REGISTERS                                ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; There are 15 general purpose registers (R0 - R14), but two of them have a special status and ;

; should normally not be used as data registers: R13 holds the stack pointer and R14 the load  ;

; register. In this program, the following general purpose registers are in use - so when      ;

; changing the code, make sure you don't accidently overwrite any data:                        ;

;                                                                                              ;

; R0-R3: to load and store various data                                                        ;

; R5-R8: for the Timer macro                                                                   ;

; R11-R12: to store return addresses                                                           ;

;                                                                                              ;

; Registers R4, R9 and R10 are currently not being used.                                       ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;                                        STEP 1: TIMER                                         ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; As a first step, we write a macro command, which accepts a parameter named "time" and        ;

; addresses the system timer. As the system timer runs independently of the cpu clock rate, it ;

; enables us to get accurate timing (real-time) in microseconds.                               ;

;                                                                                              ;

; The system timer base is in ST (= 0x3F003000)                                                ;

; The system timer counter lower 32 bits is in CLO (= 0x3F003004)                              ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

macro Timer time {

  local .wait                  ;defines a local loop called ".wait"

  imm32 r8, time               ;defines R8 as a 32-bit register and moves the value of "time"

                               ; (given as parameter

                               ;when the Timer macro is called upon) into it. R8 = requested

                               ; duration in microseconds

;initialising the Timer

  mov r6, PERIPHERAL\_BASE      ;R6 = 0x3F000000

  orr r6, r6,TIMER\_BASE        ;R6 = (0x3F000000 | 0x00003000) = 0x3F003000

  ldr r7, [r6,TIMER\_CNT]       ;R7 = saves the initial time in microseconds (we get that time

                               ; from the system timer at 0x3F003004)

  .wait:

   ldr r5, [r6,TIMER\_CNT]      ;R5 = current time in microseconds (we get that time from the

                               ; system timer at 0x3F003004)

   sub r5, r5,r7               ;R5 = elapsed time (current time - initial time)

   cmp r5, r8                  ;compares the elapsed time with the requested duration

  blt .wait                    ;if R5 < requested duration, return to .wait and execute loop

                               ; again

}

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;                             STEP 2: Defining the GPIOs as outputs                            ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; GPIO17 is in GPFSEL1 (= 0x3F200004) at bits 23-21.                                           ;

; GPIO18 is in GPFSEL1 (= 0x3F200004) at bits 26-24.                                           ;

;                                                                                              ;

; To define the GPIOs as outputs, we have to set their values in GPFSEL1 to 001. As both pins  ;

; are in the same register, we can define them as outputs in one and the same step. The pins   ;

; lie next to each other, so we want to set the values in bits 24 to 21 to 1001 (= decimal 9). ;

; With a following logical shift left, the first 1 is on bit 21 and the second 1 on bit 24.    ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;initialising the GPIOs

mov r0, PERIPHERAL\_BASE         ;R6 = 0x3F000000

orr r0, r0, GPIO\_BASE           ;R6 = (0x3F000000 | 0x00200000) = 0x3F200000

mov r1, #9

lsl r1, #21

str r1, [r0, GPFSEL1]           ;GPIO17 and GPIO18 ports are set as outputs.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;                                   STEP 3: The SOS morse code                                  ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Here it is: The morse code "SOS" as an endless loop.                                          ;

;                                                                                               ;

; Of course, the morse code is adjustable at will. We can write sentences containing words and  ;

; numbers, as all 26 letters of the latin alphabet and numbers 0-9 exist as separate branches.  ;

; Just write "bl letter\_[your letter]" or "bl number\_[your number]" inside the loop - or remove ;

; the loop altogether if you want to have your sentence sent only once.                         ;

;                                                                                               ;

; As the code structure contains several nested branches, it is important to bear in mind that  ;

; the command "bl" (branch with link) saves the current instruction set with all set flags from ;

; the program counter (R15) in R14 before branching into the subroutine.                        ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

loop:

  bl letter\_s

  bl letter\_o

  bl letter\_s

  bl new\_word

b loop

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;                              STEP 4: The branch for the time unit                             ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; The branch for the time unit (one dit) can be found here.                                     ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

time\_unit\_dit:                  ;one dit is the basic time unit from which all elements can be

                                ; derived

  Timer 240000                  ;wait 240000 microseconds (= 240 milliseconds), the duration

  bx lr                         ;corresponding to a speed of 5 words per minute.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;                        STEP 5: The branches for dit, dah, and new word                        ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; The branches for dit, dah, and new word can be found here. In here, the LED and audio speaker ;

; are turned on or off for the duration of one, three or seven dits (as for a dit, a dah or a   ;

; new word, respectively).                                                                      ;

;                                                                                               ;

; To turn the GPIOs on, we have to set bits 17 and 18 (for GPIO17 and GPIO18) in register       ;

; GPSET0 (= 0x3F20001C) to 1. The pins lie next to each other, so we want to set the values in  ;

; bits 18 to 17 to 11 (= decimal 3). With a following logical shift left, the first 1 is on bit ;

; 17 and the second 1 on bit 18.                                                                ;

;                                                                                               ;

; To turn the GPIOs off, we have to set bits 17 and 18 (for GPIO17 and GPIO18) in register      ;

; GPCLR0 (= 0x3F200028) to 1. The pins lie next to each other, so we want to set the values in  ;

; bits 18 to 17 to 11 (= decimal 3). With a following logical shift left, the first 1 is on bit ;

; 17 and the second 1 on bit 18.                                                                ;

;                                                                                               ;

; For a lower or higher audio pitch, some values in both dit and dah have to be adjusted. The   ;

; default frequency for the output is 2000Hz. For a higher pitch, the frequency has to become   ;

; higher and for a lower pitch, the frequency has to become lower. The frequency is set by the  ;

; value in the parameter of the Timer macro. As default, the value is set to 500, so every 500  ;

; microseconds, the signal is turned on and off. 1000000 microseconds (= 1 second) divided by   ;

; 500 result in the abovementioned 2000Hz. For the standard pitch A ("Kammerton") of 440Hz, the ;

; Timer parameter has to be set to 2273 (1000000 / 440). Keep in mind, however, that the signal ;

; length is also going to be longer or shorter, if only the Timer parameter is adjusted. If the ;

; default signal length of 240 milliseconds is not to be changed, use the following formula:    ;

; counter (value in the "mov r2" command = (240000 / (Timer + Timer).                           ;

;                                                                                               ;

; A note to the "new word" branch: one space between two words corresponds to seven dits. As    ;

; each letter has already a pause of one dah (= three dits) at the end of its last signal (as a ;

; space between letters), we have to insert only four more dits for a new word.                 ;

;                                                                                               ;

; Remember that in our "loop" code, the command "bl" had the program counter saved in R14. To   ;

; return from a "bl" branch to the instruction where it was called upon, we can just use the    ;

; "bx: lr" command at the end of the branch (as it is done in the branch "time\_unit\_dit"        ;

; above). This leads to an exchange of the instruction sets in the link register (R14) and the  ;

; register for the program counter (R15).                                                       ;

;                                                                                               ;

; However, if inside a "bl" branch another "bl" is used, the content in the link register (R14) ;

; is lost, as it is overwritten with the new "bl" command. Therefore, it is necessary to save   ;

; the content in R14 in another register at the beginning of each "bl" branch that uses "bl"    ;

; commands for itself (nested branches). This enables us to return to the main program even if  ;

; we use several nested branches. To do so, we just have to rewrite the "bx" command so it      ;

; addresses the register with the saved original instruction set (e.g. "bx: r11).               ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

dit:

  movs r11, r14                 ;sets the root from which the subroutine was started into R11

                                ; (return address)

  mov r1, #3

  lsl r1, #17

  mov r2, #240                  ;a counter is used to turn GPIO17 and GPIO18 on and off 240 times

                                ; for 500 microseconds (= 0.5 milliseconds) each

  countdown\_dit:                ;This corresponds to a frequency of 2000Hz for a duration of

                                ; 240 milliseconds.

   str r1, [r0, GPSET0]         ;LED and speaker turn on ...

   Timer 500                    ;wait 0.5 milliseconds

   str r1, [r0, GPCLR0]         ;LED and speaker turn off ...

   Timer 500                    ;wait 0.5 milliseconds

   sub r2, #1

   cmp r2, #0

  bne countdown\_dit

  bl time\_unit\_dit              ;wait for one dit

  bx r11

dah:

  movs r11, r14                 ;sets the root from which the subroutine was started into R11

                                ; (return address)

  mov r1, #3

  lsl r1, #17

  mov r2, #720                  ;a counter is used to turn GPIO17 and GPIO18 on and off 720 times

                                ; for 500 microseconds (= 0.5 milliseconds) each

                                ;This corresponds to a frequency of 2000Hz for a duration of

                                ; 720 milliseconds.

  countdown\_dah:

   str r1, [r0, GPSET0]         ;LED and speaker turn on ...

   Timer 500                    ;wait 0.5 milliseconds

   str r1, [r0, GPCLR0]         ;LED and speaker turn off ...

   Timer 500                    ;wait 0.5 milliseconds

   sub r2, #1

   cmp r2, #0

  bne countdown\_dah

  bl time\_unit\_dit              ;wait for one dit

  bx r11

new\_word:

  movs r12, r14                 ;sets the root from which the subroutine was started into R12

                                ; (return address)

  mov r3, #4

  countdown\_new\_word:           ;a counter is used to call upon the branch time\_unit\_dit four

                                ; times.

   bl time\_unit\_dit

   sub r3, #1

   cmp r3, #0

  bne countdown\_new\_word

  bx r12

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;                       STEP 6: The branches for all letters and numbers                       ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; The branches for all letters and numbers can be found here.                                  ;

;                                                                                              ;

; A note to the insertion of "bl time\_unit\_dit" at the end of each branch: one space between   ;

; two letters corresponds to three dits. As each letter has already a pause of one dit before  ;

; the last command "bx" inside the dit or dah element (as a space between signals), we have to ;

; insert only two more dits for a new letter.                                                  ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

letter\_a:                       ;di-dah

  movs r12, r14

  bl dit

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_b:                       ;dah-di-di-dit

  movs r12, r14

  bl dah

  bl dit

  bl dit

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_c:                       ;dah-di-dah-dit

  movs r12, r14

  bl dah

  bl dit

  bl dah

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_d:                       ;dah-di-dit

  movs r12, r14

  bl dah

  bl dit

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_e:                       ;dit

  movs r12, r14

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_f:                       ;di-di-dah-dit

  movs r12, r14

  bl dit

  bl dit

  bl dah

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_g:                       ;dah-dah-dit

  movs r12, r14

  bl dah

  bl dah

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_h:                       ;di-di-di-dit

  movs r12, r14

  bl dit

  bl dit

  bl dit

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_i:                       ;di-dit

  movs r12, r14

  bl dit

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_j:                       ;di-dah-dah-dah

  movs r12, r14

  bl dit

  bl dah

  bl dah

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_k:                       ;dah-di-dah

  movs r12, r14

  bl dah

  bl dit

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_l:                       ;di-dah-di-dit

  movs r12, r14

  bl dit

  bl dah

  bl dit

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_m:                       ;dah-dah

  movs r12, r14

  bl dah

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_n:                       ;dah-dit

  movs r12, r14

  bl dah

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_o:                       ;dah-dah-dah

  movs r12, r14

  bl dah

  bl dah

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_p:                       ;di-dah-dah-dit

  movs r12, r14

  bl dit

  bl dah

  bl dah

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_q:                       ;dah-dah-di-dah

  movs r12, r14

  bl dah

  bl dah

  bl dit

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_r:                       ;di-dah-dit

  movs r12, r14

  bl dit

  bl dah

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_s:                       ;di-di-dit

  movs r12, r14

  bl dit

  bl dit

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_t:                       ;dah

  movs r12, r14

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_u:                       ;di-di-dah

  movs r12, r14

  bl dit

  bl dit

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_v:                       ;di-di-di-dah

  movs r12, r14

  bl dit

  bl dit

  bl dit

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_w:                       ;di-dah-dah

  movs r12, r14

  bl dit

  bl dah

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_x:                       ;dah-di-di-dah

  movs r12, r14

  bl dah

  bl dit

  bl dit

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_y:                       ;dah-di-dah-dah

  movs r12, r14

  bl dah

  bl dit

  bl dah

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

letter\_z:                       ;dah-dah-di-dit

  movs r12, r14

  bl dah

  bl dah

  bl dit

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

number\_0:                       ;dah-dah-dah-dah-dah

  movs r12, r14

  bl dah

  bl dah

  bl dah

  bl dah

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

number\_1:                       ;di-dah-dah-dah-dah

  movs r12, r14

  bl dit

  bl dah

  bl dah

  bl dah

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

number\_2:                       ;di-di-dah-dah-dah

  movs r12, r14

  bl dit

  bl dit

  bl dah

  bl dah

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

number\_3:                       ;di-di-di-dah-dah

  movs r12, r14

  bl dit

  bl dit

  bl dit

  bl dah

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

number\_4:                       ;di-di-di-di-dah

  movs r12, r14

  bl dit

  bl dit

  bl dit

  bl dit

  bl dah

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

number\_5:                       ;di-di-di-di-dit

  movs r12, r14

  bl dit

  bl dit

  bl dit

  bl dit

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

number\_6:                       ;dah-di-di-di-dit

  movs r12, r14

  bl dah

  bl dit

  bl dit

  bl dit

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

number\_7:                       ;dah-dah-di-di-dit

  movs r12, r14

  bl dah

  bl dah

  bl dit

  bl dit

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

number\_8:                       ;dah-dah-dah-di-dit

  movs r12, r14

  bl dah

  bl dah

  bl dah

  bl dit

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

number\_9:                       ;dah-dah-dah-dah-dit

  movs r12, r14

  bl dah

  bl dah

  bl dah

  bl dah

  bl dit

  bl time\_unit\_dit

  bl time\_unit\_dit

  bx r12

1. Quelle aus dem Moodle des Moduls GTI.Bsc INF 2019.ZH2.HS19/20 (nicht öffentlich zugänglich) der FFHS. [↑](#footnote-ref-1)
2. Quelle aus dem Moodle des Moduls GTI.Bsc INF 2019.ZH2.HS19/20 (nicht öffentlich zugänglich) der FFHS. [↑](#footnote-ref-2)
3. Quelle aus dem Moodle des Moduls GTI.Bsc INF 2019.ZH2.HS19/20 (nicht öffentlich zugänglich) der FFHS. [↑](#footnote-ref-3)