

**Design und prototypische Implementierung
einer Embedded-Software Architektur
für ein Funkanalysengerät**

**Design and prototypical implementation
of an embedded software architecture
for a radio analysis device**

Bachelorarbeit
zur Erlangung des Grades *Bachelor of Science*

an der
Hochschule Niederrhein
Fachbereich Elektrotechnik und Informatik
Studiengang *Informatik*

vorgelegt von
Tim Krüger
1197031

Krefeld, den 12. Januar 2022

Prüfer: Prof. Dr. Jürgen Quade
Zweitprüfer: Prof. Dr. Jens Brandt

Eidesstattliche Erklärung

Name: Tim Krüger

Matrikelnr.: 1197031

Titel: Design und prototypische Implementierung einer Embedded-Software Architektur
für ein Funkanalysegerät

Ich versichere durch meine Unterschrift, dass die vorliegende Arbeit ausschließlich von mir verfasst wurde. Es wurden keine anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt.

Die Arbeit besteht aus 54 Seiten.

Krefeld, 12.01.22
Ort, Datum

Krüger
Unterschrift

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	1
1.2 Aufgabenstellung	1
2 Stand der Technik	2
2.1 Kurzschlussanzeiger	2
2.2 Mikroprozessoren und Mikrocontroller	3
2.3 STM32	4
3 Architektur und Design	7
3.1 Hardware	7
3.2 Software	12
3.3 Debugging	23
4 Implementierung	25
4.1 Aufsetzen der Projektstruktur	26
4.2 Funktionalität	29
4.3 Utility	30
4.4 Display-Treiber	31
4.5 Interrupts	38
4.6 Applikation	39
5 Fazit und Ausblick	47
5.1 Zusammenfassung	47
5.2 Ergänzungen	48
5.3 Ausbau und Erweiterung	48

1 Einleitung

1.1 Motivation

Embedded Systems sind omnipräsent. Egal ob im Smartphone, in IOT-Geräten oder im Auto. Täglich agieren wir mit tausenden der kleinen Chips und sie sind aus unserem Alltag nicht mehr wegzudenken. Auch in der Industrie verlässt man sich seit Jahren auf programmierbare Mikroprozessoren. Flexibilität, Wiederverwendbarkeit von Programmen und eine kostengünstige Anschaffung sprechen oft gegen den Einsatz von dedizierter Hardware.

Diese großflächige Abhängigkeit von Mikrochips hat ihre Schattenseiten. Stand Oktober 2021 herrscht noch immer eine Chip-Knappheit. Die Preise von Consumer Electronic sind auf einem Allzeithoch und man konnte dieses Jahr beobachten, wie ganze Produktionsketten aufgrund fehlender Chips heruntergefahren wurden. Dazu kommt, dass die Fertigung, vor allem der Highend-Chips (7nm und kleiner), auf ein paar wenige Firmen konzentriert ist. Die genauen Details sind komplex und würden den Rahmen der Arbeit sprengen.

Die aktuelle Situation hat die Hardware-Entscheidungen dieser Bachelorarbeit maßgeblich mitgeprägt.

Die Anforderungen an Embedded Systems sind andere, als man diese bei der Programmierung von x86-Systemen findet. Es geht um Echtzeitfähigkeiten, Energieeffizienz und um extrem begrenzte Hardware-Ressourcen. Ein Bereich, der seine ganz eigenen Herausforderungen mit sich bringt.

Im Gespräch mit meinem Arbeitgeber, einem mittelständischen Elektrotechnik-Unternehmen, hat sich der Wunsch nach der Entwicklung einer Software-Lösung für ein neues Produkt ergeben. Wir stellen Produkte zur Überwachung, Analyse und Fehlersuche in Mittelspannungsnetzen her und haben bei einigen Kunden spezielle Anforderungen, für die derzeit keine richtige Lösung vorhanden ist.

Unsere Überwachungs-Geräte werden in den Verteilnetzen im Mittelspannungsbereich u.a. an Freileitungen eingesetzt und kommunizieren im Nahbereichs-Funk mit einer Zentraleinheit. Diese sammelt, bündelt und verschickt erfasste Daten. In einigen Umgebungen ist die Funkqualität zwischen den Geräten und der Zentraleinheit eine große Herausforderung. So gibt es bspw. Probleme mit Funk-Abschattungen durch Stahlmaste.

Das Ziel der Bachelorarbeit ist die Software-Architektur für ein Gerät zu konzipieren, mit dem man die Funkqualität messen und somit die beste Montageposition der Einheiten bestimmen kann.

1.2 Aufgabenstellung

Es soll sich bei dem Gerät um ein robustes Handheld-Device, mit Display und Buttons handeln. Ein Hardware-Prototyp wird zur Verfügung gestellt (siehe Abbildung 1). Das Gerät soll Batterie betrieben werden.

Bei neuen Produkt-Entwicklungen wird hier im Unternehmen mittlerweile oft auf STM32-Mikrocontroller von *STMicroelectronics* gesetzt (mehr dazu in Kapitel 2.3). Dies hat den Vorteil eines einheitlichen Produktpportfolios. Durch Experten im Team kann man bei Wiederverwendung der gleichen Architektur oder (besser) sogar des exakt gleichen Mikrocontrollers den Energieverbrauch stark optimieren. Man kann sich außerdem am Programm-Code von vorhandenen Projekten orientieren. Dies ist deshalb wichtig, da später geplant ist, dass das Gerät unser hauseigenes Binärprotokoll entschlüsseln und verschiedene weitere, auch Langzeit-, Überwachungsfunktionen einnehmen können soll. Auch Batterilaufzeiten von mindestens einer Woche sind dann erforderlich.

Die Anforderungen sind also vielfältig. In dieser Bachelorarbeit geht es um die Konzeption einer Embedded-Software Architektur welche modular, effizient und leicht erweiterbar ist. Grundsätzliche Funktionen und Module des Gerätes sollen initialisiert, implementiert und getestet werden. Das Ziel ist ein bedienfähiger Prototyp, welcher in Zukunft Funktests im Laborumfeld und im Feld (im Rahmen von Inbetriebnahmetests) durchführen kann.

Die Aufgaben beinhalten:

- Aufsetzen eines leichtgewichtigen Embedded-Projekts mit einem Echtzeitbetriebssystem
- Grundinitialisierung: Konfigurieren der RTOS-Tasks, GPIO-Schnittstellen, Timer und Taktraten
- Entwicklung eines modularen Display-Treibers
- Schreiben diverser Interrupt-Service-Routinen zur Reaktion auf externe Events
- Entwicklung und prototypische Implementierung einer State-Machine-Architekture zur Auswertung von Events, Starten von Tests und zur Aktualisierung des Displays
- Integration des hauseigenen Funk-Treibers
- Planung eines ersten Tests "Discovery der Geräte im Laborumfeld"

Durch die Situation am Weltmarkt sind viele Prototypen-Boards, wie wir sie sonst verwenden, kaum zu bekommen. Wir sind deshalb auf die hier vorhandene Hardware beschränkt. Dies betrifft auch Peripherie-Komponenten, wie das Display. Für den Prototypen wurden zwei Platinen aus bereits vorhandenen Produkten verwendet. Es werden Display und Buttons von der oberen Platine und Mikrocontroller, Funkmodul und Stromversorgung von der unteren verwendet. Die beiden Platinen sind in einer "Huckepack-Lösung" mit Fädel-Drähten verbunden. Es müssen zwei Platinen verwendet werden, da auf der oberen ein veralteter Mikrocontroller verbaut ist, welcher den modernen Ansprüchen nicht genügt und in großen Teilen noch in Assembler programmiert werden musste. Ein serienreifer Geräte-Entwurf, untergebracht in einem einzelnen Gehäuse, ist in Arbeit.

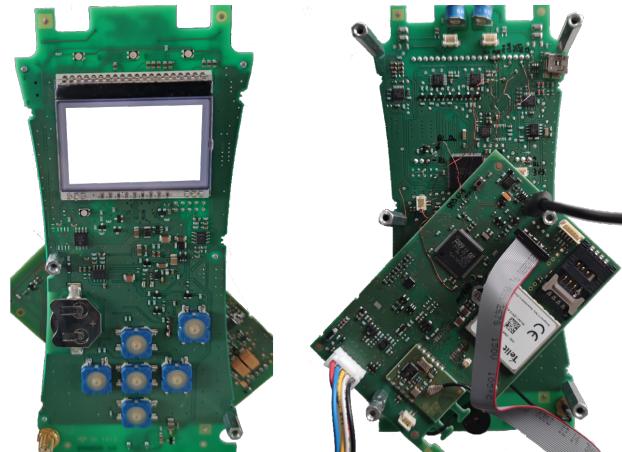


Abbildung 1: Bild des Prototypen,
Quelle: [1].

2 Stand der Technik

2.1 Kurzschlussanzeiger

Wie in Kapitel 1.1 erwähnt, findet diese Bachelorarbeit im Umfeld eines mittelständischen Elektrotechnik-Unternehmens statt. Der Dipl.-Ing. H. Horstmann GmbH. Das Unternehmen hat sich auf die Herstellung von Produkten zur Fehlersuche und Analyse in Mittelspannungsnetzen spezialisiert. Vom Hardware-Entwurf, über die Platinenbestückung, bis hin zur Programmierung der Mikrocontroller, werden alle Entwicklungs- und Produktionsschritte inhouse realisiert. Auch Anwendungsprogramme zur Bedienung und Steuerung der Produkte für Endanwender entwickelt das Unternehmen selber. Im Folgenden wird ein

spezielles Produkt genauer beleuchtet, die aktuelle Version des hauseigenen Kurzschlussanzeigers. Es ist gerade die Inbetriebnahme dieses Gerätes, welche diese Bachelorarbeit zum Großteil initiiert hat.

Grundsätzlich sind Kurzschlussanzeiger, an Freileitungen montierte Geräte, zur Erfassung/Messung verschiedener Eckdaten und zur Erkennung/Lokalisierung von Kurzschlägen. Die aktuelle Generation des Horstmann-Kurzschlussanzeigers, genannt "Smart Navigator 2.0" (siehe Abbildung 2), und dessen Vorgänger, bilden die Grundlage des Produkt-Portfolios des Unternehmens und finden international Einsatz.



Abbildung 2: Smart Navigator 2.0,
Quelle: [2].

Der Smart Navigator 2.0 ist für Mittelspannung bis 69kV ausgelegt. Das Gerät versorgt sich induktiv über den Leiterstrom und bereit 5A reichen zur Versorgung aus. Wie der Name des Gerätes schon vermuten lässt, ist es "smart". Es ist in der Lage, die erfassten Daten über die Cloud an eine Leitwarte weiterzuleiten und Over-Air-Updates zu empfangen. Das Gerät wird mit einer speziellen Halterung an einer Freileitung montiert und besitzt im unteren Bereich drei High-Power-LEDs, welche Monitorings- und Fehlerlokalisierungszwecken dienen. Das Gerät kann Spannungsbrüche und Stromabfälle erkennen. Zusätzlich werden Fehlerstromrichtung und Leiterseittemperatur erfasst [3]. Kunden werden über Web-Schnittstellen und Anwendungsprogramme Möglichkeiten geboten, erfasste Daten auszulesen und Geräte umzukonfigurieren. Eine Einheit besteht aus mehreren Kurzschlussanzeigern und einer zentralen Sende-/Empfangseinheit. Die Geräte kommunizieren untereinander im Nahbereichs-Funk auf 868MHz. Informationen und Daten werden von einer Zentralenheit gesammelt und per GSM an die Leitwarte verschickt.

2.2 Mikroprozessoren und Mikrocontroller

Im Folgenden werden die allgemeinen Gemeinsamkeiten und Unterschiede von Mikroprozessoren und Mikrocontrollern aufgezeigt und definitorisch eingeführt.

Ein Mikroprozessor, i.d.R. als MPU (Microprocessor Unit) bezeichnet, ist einfach nur ein Prozessor. Er integriert keinen externen Speicher, hat keine eigene Spannungswandlung und auch keine Controller um mit Peripherie kommunizieren zu können. Ein Mikroprozessor benötigt externe Komponenten, welche diese Funktionen integrieren [5].

Ein Mikrocontroller, i.d.R. als MCU (Microcontroller Unit) bezeichnet, hingegen, enthält exakt diese Komponenten, zusammen mit einem Mikroprozessor, auf einem Chip. Ein Mikrocontroller implementiert diverse I/O-Komponenten, Speicher, Zeit-/Taktgeber und Schnittstellen. Er enthält auch Komponenten zur Stromversorgung/Spannungswandlung und benötigt so nur eine Versorgungsleitung. Mikrocontroller sind äußerst praktisch, wenn die Anforderungen an ein Produkt von vornherein bekannt sind. Sie bilden die Grundlage von vielen Embedded Systems [5].

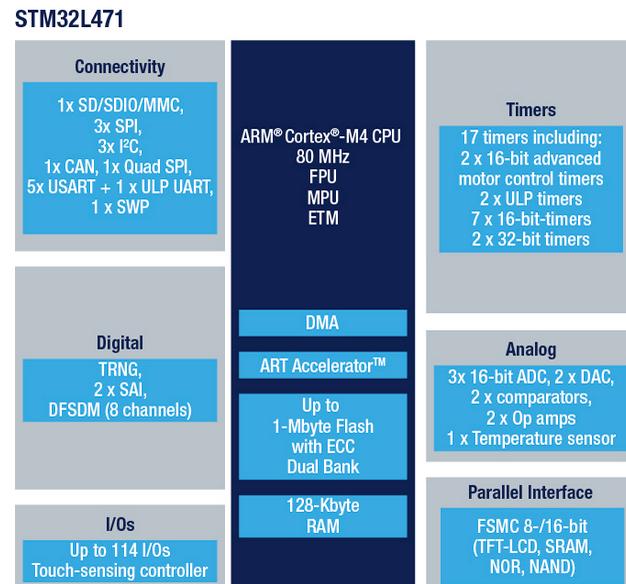


Abbildung 3: Verwendeter Mikrocontroller STM32L471,
Quelle: [4].

Mikroprozessoren hingegen, geben keinerlei Vorgaben, was die Speichergröße oder die externen Peripheriemöglichkeiten angehen [5]. Die Implementierung obliegt allein dem Mikrocontroller- bzw. Mainboard-Hersteller. Mikroprozessoren finden einzeln oft in Desktop-Rechnern Anwendung und sind ansonsten Teil eines jeden Mikrocontrollers. Hersteller von Mikroprozessoren, wie bspw. *ARM*, verkaufen Lizenzen bzw. Nutzungsrechte für ihre Prozessoren und überlassen Mikrocontroller-Herstellern, wie bspw. *STMicroelectronics*, die Implementierung weiterer Funktionalitäten [6] (siehe Abbildung 3).

2.3 STM32

2.3.1 Allgemeines

Im Folgenden findet ein Einstieg in die Reihe der STM32-Mikrocontroller statt. Es wird einen Vergleich mit anderen Mikrocontrollern und einen kurzen Überblick über das Feature-Set des verwendeten Controllers geben.

Die STM32-Mikrocontroller gehören zu den derzeitigen Highend-Controllern. Verbaut sind *ARM*-Kerne. Geboten wird ein breites Spektrum an Leistungsfähigkeiten (Cortex-M0- bis Cortex-M7-Prozessoren) und Energieeffizienz-Klassen. Die Dokumentation von *STMicroelectronics* wird als sehr gut empfunden [6]. Auch zur Programmierung von *ARM*-Cortex-Prozessoren gibt es reichlich Materialien. Es stehen außerdem viele verschiedene Prototypen-Boards zur Auswahl. Prototypen-Boards sind Platinen, auf denen sich neben einem STM-Mikrocontroller, auch Pin-Header (oft Arduino-kompatibel) und bereits vorinstallierte, dedizierte Hardware-Debugger befinden (siehe Abbildung 4). Programmieren, Flashen und Debuggen wird dadurch deutlich erleichtert. Bezuglich der Programmierung sind auch die guten Compiler, wie bspw. *ARMCC* oder *IARCC*, zu erwähnen. Durch die bekannte *ARM*-Architektur sind starke Optimierungen möglich.

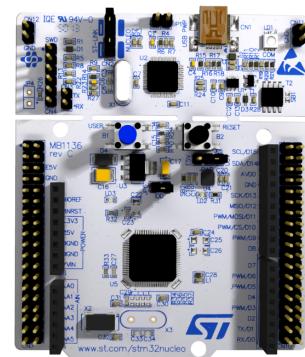


Abbildung 4: Nucleo-64
Prototyping-Board,
Quelle: [7].

Processor	MHz	Cores	CoreMark	CoreMark/MHz
Espressif ESP32-S2	240	1	472.81	1.97
STM32L476	80	1	273.55	3.42
TI MSP430F5529	25	1	27.70	1.11
Atmel ATmega644	20	2	10.81	0.54
Atmel ATmega2560	8	1	4.25	0.53

Tabelle 1: CoreMark™ MCU-Benchmark-Ergebnisse,
Quelle: [8].

Tabelle 1 kann man einen Leistungsvergleich einiger beliebter Mikrocontroller mit dem STM32L476 entnehmen. Als Vergleichswerte wurden die Ergebnisse des CoreMark™-Benchmarks des *Embedded Microprocessor Benchmark Consortiums* angeführt. Die Performance des STM32L476 ist identisch zu dem in dieser Arbeit verwendeten STM32L471, weswegen sich die Werte eignen. Die beiden Controller unterscheiden sich lediglich in einigen Peripherieoptionen [9] [10].

Auffällig ist die starke Performance des ESP32, welche auf die hohe Taktrate zurückzuführen ist. Diese fällt, im Vergleich zur Konkurrenz, deutlich höher aus. Der STM32 erreicht mit einem Drittel des Taktes, etwas mehr als die Hälfte der Leistung des ESP32. Was zur besten Performance pro MHz in diesem Vergleich führt. Ergänzend muss erwähnt werden, dass der STM32L4 einen Cortex-M4-Prozessor, leistungstechnisch eher Mittelklasse, verbaut hat. Den ESP32 gibt es außerdem in einer deutlich leistungsfähigeren

Dual-Core-Variante. Der MSP430-Controller von Texas Instruments, welcher hier im Unternehmen immer noch verwendet wird, erreicht weniger als ein Drittel der Leistung pro Takt im Vergleich zum STM32. Die ATmega-Controller, welche oft auf Arduino-Boards verbaut werden, fallen in diesem Leistungsvergleich stark zurück.

Leistungstechnisch bewegt sich der STM32L4 in diesem Vergleich demnach im oberen Bereich. Seine richtige Stärke spielt der Controller nun im Bereich Peripherieoptionen (siehe Abbildung 3) und beim Thema Energieeffizienz aus. Dazu kommen etliche Ultra-Low-Power-Funktionalitäten.

Processor	Clockspeed	Run	Sleep/Standby (with RTC)	Shutdown
Espressif ESP32-S2	240 MHz	79.16 µA/MHz	20 µA	1.0 µA
STM32L476	80 MHz	100 µA/MHz	0.42 µA	0.03 µA
TI MSP430F5529	8 MHz	290 µA/MHz	2.10 µA	0.18 µA
Atmel ATmega644	1 MHz	240 µA/MHz	/	0.1 µA
Atmel ATmega2560	1 MHz	500 µA/MHz	/	0.1 µA

Tabelle 2: Vergleich des MCU-Energieverbrauchs,

Quellen: [11] [10] [12] [13] [14].

Tabelle 2 zeigt den Energieverbrauchs der verschiedenen Mikrocontroller in diversen Betriebsmodi im Vergleich. Beachtet werden müssen die verringerten Taktraten des MSP430 und der AVR-Controller. Zu anderen Taktraten machen die Datenblätter keine Angaben. Bei den AVR-Controllern ebenso wenig zu Sleep- bzw. Standby-Modes, in denen noch eine Realtime-Clock mitläuft. Zusätzlich muss festgehalten werden, dass alle Werte des ESP32 mit deaktiviertem Modem angegeben sind. Diese herstellerseitig integrierten Funk-Fähigkeiten sind ein großer Pluspunkt des ESP32, würden (eingeschaltet) den Stromverbrauch aber signifikant erhöhen.

Auffällig an den Werten ist der sehr niedrige Energieverbrauch des STM32. Er ist der Konkurrenz, in diesem Vergleich, in fast allen Aspekten überlegen. Lediglich beim Vergleich der benötigten Energie pro MHz, siegt der ESP32. Dies ist auf seinen sehr hohen Takt zurückzuführen. Zusätzlich muss zum ESP32 festgehalten werden, dass er nicht primär für Ultra-Low-Power-Applikationen entworfen wurde. Seine Stärken liegen durch vorinstallierte Wifi- und Bluetooth-Fähigkeiten in anderen Bereichen. Der MSP430 fällt in diesem Vergleich, trotz seines höheren Taktes, hinter den ATmega644 zurück.

Wie eingangs erwähnt, ist Energieeffizienz eines der Hauptkriterien. Geräte hängen Jahre an Freileitungen, am Mast oder sind unerreichbar in Unterwasser-Messstationen verbaut. Dabei kommt es auf jedes Mikroampere an Stromverbrauch an. Auch bei batteriebetriebenen Remote-Devices, wie dem Funkanalysegerät für diese Bachelorarbeit, ist eine hohe Energieeffizienz des Mikrocontrollers zwingend erforderlich. Mit dem STM32L4 bietet sich einem ein sehr energieeffizienter Mikrocontroller, welcher durchaus auch leistungstechnisch mit anderen modernen Controllern mithalten kann.

2.3.2 Programmierung

In diesem Abschnitt werden die Programmieroptionen, die einem bei der Entwicklung von Embedded-Software für die STM32-Plattform zur Verfügung stehen, vorgestellt. Die Programmierung von Mikrocontrollern funktioniert über Memory-Mapped-I/O. Dies bedeutet, dass Mikrocontroller-Peripherie auf bestimmte Speicherbereiche gemapped ist. Diese Bereiche modifiziert man mit den üblichen Methoden die Programmiersprachen, wie C/C++, bieten. So setzt man z.B. zur Aktivierung von Bus-Takten oder zum Einschalten von Peripherie-Komponenten verschiedene Bits in bestimmten Registern.

Aufbauend auf der direkten Bit-Manipulation gibt es verschiedene Abstraktionsschichten. Nennenswert sind dabei *CMSIS* und *HAL*.

Bei *CMSIS* (Cortex Microcontroller Software Interface Standard) handelt es sich um eine Sammlung von Funktionen, Makros und Definitionen, welche sowohl den Prozessorkern von *ARM*, als auch den Mikrocontroller von *STM* selbst betreffen. Dort sind alle Komponenten, inklusive ihrer Registernamen und Speicheradressen bereits beschrieben [6]. Die Vorteile von *CMSIS* sind die sprechenden Namen und Definitionen, die zur Verfügung gestellt werden. Die Funktion einer Komponente ist direkt ersichtlich und es handelt sich nicht mehr um eine beliebige Speicheradresse.

Ein *HAL* (Hardware-Abstraction-Layer) setzt darauf auf und “umhüllt“ vorhandene Makros in vollwertigen Funktionen. Die Vorteile sind noch klarere Funktions- und Komponenten-Bezeichnungen und die stärkere Integration vom Compiler, welcher bei tief geschachtelten Makros nicht immer ideal unterstützen kann. Zusätzlich ist die Portierung von Projekten, bei Verwendung desselben Hardware-Abstraction-Layers, auf andere Hardware-Plattformen deutlich leichter. Als Nachteil könnte man zusätzlichen “Boiler-Plate-Code“ nennen.

Listing 1 kann man einen direkten Vergleich der beiden Programmier-Schichten entnehmen. Der Programm-Code dient dem Aktivieren des Bus-Taktes der Peripherie-Komponente GPIOA im RCC-Register (Reset and Clock Control Register).

```

1 // *** Beispielhaftes Hardware-Abstraction-Layer (HAL) *** -> R. Jesse
2
3 void HAL_GpioInitPort(GPIO_TypeDef * port)
4 {
5     if (GPIOA == port)
6     {
7         RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
8     }
9 }
10
11 // *** CMSIS MCU-Device-File *** -> stm32f446xx.h
12
13 typedef struct
14 {
15     [...]
16
17     __IO uint32_t AHB1ENR; // RCC AHB1 peripheral clock register, Offset: 0x30
18
19     [...]
20
21 } RCC_TypeDef;
22
23 #define RCC_AHB1ENR_GPIOAEN_Pos (0U)
24 #define RCC_AHB1ENR_GPIOAEN_Msk (0x1UL << RCC_AHB1ENR_GPIOAEN_Pos) // 0x00000001
25 #define RCC_AHB1ENR_GPIOAEN RCC_AHB1ENR_GPIOAEN_Msk
26
27 // *** CMSIS MPU-Device-File *** -> core_cm4.h
28
29 #define __IO volatile // Defines 'read / write' permissions

```

Listing 1: MCU-Programmierebenen (beispielhaft für STM32F446RE),
Quellen: [6] [15] [16].

Bei der in Listing 1 beschriebenen HAL-Funktion handelt es sich um eine exemplarische Implementierung, wie sie beispielsweise im Buch von R. Jesse vorkommt [6]. Im Projekt findet die offizielle HAL-Library von *STM* Verwendung. Diese wäre allerdings aus Platzgründen nicht abdruckbar gewesen.

Bei der Nutzung des Hardware-Abstraction-Layers von *STM* ergeben sich einige Vorteile. Durch Software-Unterstützung gibt es bspw. visuelle Möglichkeiten Schnittstellen, Komponenten oder auch Timer/Taktraten zu konfigurieren. Gerade letztere sind bei modernen Mikrocontrollern durch die Vielzahl an Bussen und Taktquellen nicht immer trivial zu konfigurieren. Es ist außerdem möglich sich Linker- und Startup-Scripts automatisch generieren zu lassen. Dies reduziert die Fehleranfälligkeit.

Linker-Scripts sind Scripte, welche vom Linker ausgeführt werden. Die Hauptaufgabe eines solchen Script ist zu beschreiben, wie verschiedene Text- und Datensektionen einer Inputdatei auf eine Outputdatei gemapped werden sollen. Ein Linker führt immer ein Linker-Script aus. Sollte kein Script vorhanden sein, wird ein Default-Script verwendet [17]. Ein allgemeines Linker-Script beinhaltet u.A. das Mapping für text-, data- und bss-Sektionen [18]. Bei Embedded Systems, speziell bei Custom-Hardware kann es vorkommen, dass man eigene Speicherbausteine verwendet und Speicherbereiche entsprechend ihrer neuen Funktionsweise ummappen muss. In so einem Fall ist eine Anpassung des Linker-Scripts erforderlich.

Ein Startup-Script, auch Startup-Code genannt, ist ein Stück Software, welches die Maschine auf die Ausführung eines Programms vorbereitet. Es kann wahlweise in Assembly oder in C geschrieben werden. Die Aufgaben vom Startup-Script beinhalten das Deaktivieren aller Interrupts, das Initialisieren des Stack-Pointers, das Initialisieren der idata-Sektion, das Setzen von Nullen in allen uninitialisierten Speicherbereichen und das Aufrufen von *main()* [18]. Es ist bei Embedded Systems auch das Startup-Script, welches einen „return“ von *main()* mit einer entsprechenden Behandlung abfangen würde. Bei x86-Systemen würde an dieser Stelle das Betriebssystem übernehmen.

3 Architektur und Design

In den folgenden Kapiteln wird ein Einblick in den Projekt-Entwurf dargelegt. Angefangen beim Hardware-Prototypen, über den Software-Stack, bis hin zu den Debugging-Werkzeugen, wird jede verwendete Komponente vorgestellt und erläutert.

3.1 Hardware

In diesem Kapitel geht es um die Hardware des Projektes. Nach einer Übersicht über den Prototypen wird sich einzelnen Design- und Technologie-Entscheidungen gewidmet.

3.1.1 Prototyp

Der Hardware-Prototyp besteht aus zwei Platinen, welche aus bereits existierenden Produkten stammen. Die beiden Platinen sind in einer „Huckepack-Lösung“ miteinander verschraubt und mit Fädel-Drähten miteinander verdrahtet (siehe Abbildung 1). Die linke Platine (vergleiche auch Abbildung 5) wird nachfolgend als *obere Platine*, die rechte Platine als *untere Platine* bezeichnet.

Die obere Platine beinhaltet u.A. ein Display-Modul, 6 Buttons und einen MSP430-Controller. Bei dem Display handelt es sich um das Model *DOGM128-6* aus der DOGM Graphic Series von Electronic Assembly. Es ist ein 128x64 großes Dot-Matrix-Display mit Hintergrundbeleuchtung, welches per SPI angesteuert wird [19]. Die 6 Buttons sind nicht in Hardware entprellt und erfordern u.U. später eine speziellere Behandlung in Software. Der verbaute MSP430-Controller ist mittlerweile recht alt und entspricht, wie vorhin bereits gezeigt, nicht mehr den Standards, die an Geschwindigkeit und Energieverbrauch gestellt

werden. Er erfordert an etlichen Stellen außerdem noch den Einsatz von Inline-Assembly, um spezielle Instruktionen durchzuführen [20].

An dieser Stelle kommt die untere Platine zum Einsatz. Bei dieser handelt es sich um den Kern eines der neuesten Produkte der Firma. Die Platine beinhaltet u.A. ein Funkmodul, ein GSM-Modul, verschiedene Speicherbausteine, ein Shiftregister, ein GPS-Modul, ein Accelerometer, 3 LEDs, einen FTDI-Chip und einen STM32L4-Controller. In diesem Projekt wird nur ein Bruchteil der Komponenten verwendet werden.

Abbildung 5 kann man das Layout des Prototypen, mit allen verwendeten Komponenten entnehmen.

Bei dem verwendeten Funkmodul handelt es sich um eine Ultra-Low-Power, Integrated-ISM-Band Sub-GHz Sende-Einheit von Microchip (*MRF89XAM8A*). Die unterstützten Frequenzbereiche reichen von 863-870 MHz. Die Geräte kommunizieren auf 863 MHz. Das Funkmodul wird identisch zum Display per SPI angesteuert [22], genaue Details folgen in Kapitel 3.1.2. Das genaue Modell des STM32L4-Controllers ist der *STM32L471VGT6*. Dabei handelt es sich, wie vorhin bereits gezeigt, um einen modernen Mikrocontroller mit starken Low-Power-Funktionalitäten. Besonders wichtig für dieses Projekt sind die 3 SPI-Lines und die Ultra-Low-Power-Timer, die der Controller bietet.

Der genaue Funktionsumfang des Controllers kann Abbildung 3 entnommen werden. Erwähnenswert ist außerdem der FTDI-Chip, welcher sich auf der Platine befindet.

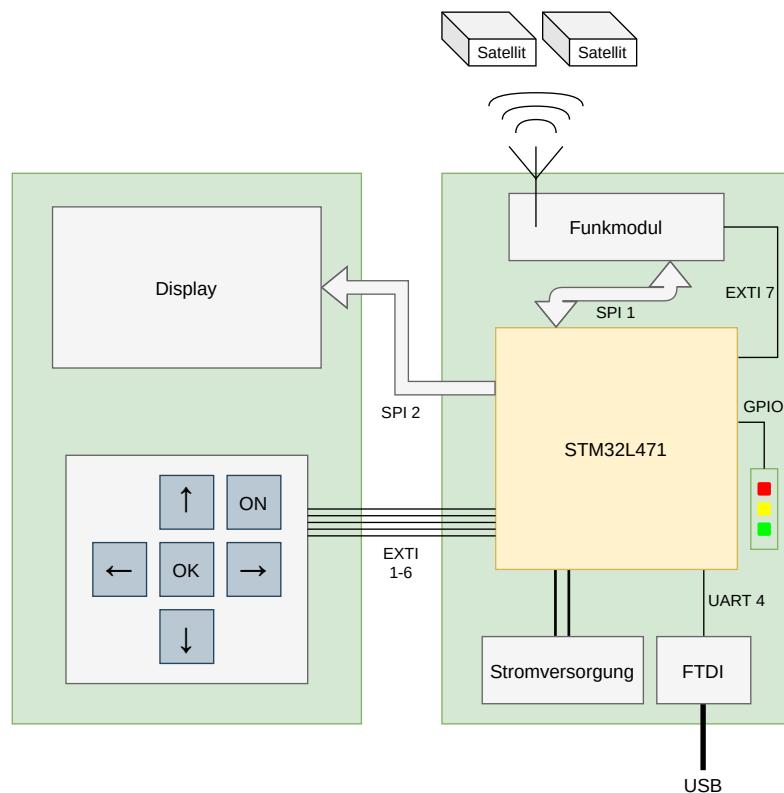


Abbildung 5: Layout des Prototypen,
Quelle: [21].

Ein *FTDI*-Chip, Kürzel für den Chip der Firma *Future Technology Devices International*, ist ein Pegelwandler-Schaltkreis, der einen Übergang von der seriellen Schnittstelle RS-232 auf USB ermöglicht [23]. Der Chip beinhaltet außerdem einen USB-Controller, welcher einen USB-Host implementiert. In unserer Fertigung wird der Chip mit den entsprechenden Identifizieren (Manufacturer-ID und Device-ID) programmiert, sodass das Gerät als Horstmann-Device erkannt wird.

Der Chip ermöglicht das Mapping einer internen UART-Schnittstelle auf einen USB-Port. Durch einen Treiber erscheint das Gerät PC-seitig als virtueller COM-Port. Dies ermöglicht das Verschicken von Daten über die UART-Schnittstelle mit einer PC-seitigen Auswertung. Auch das bekannte *printf-Debugging* wird dadurch ermöglicht.

Die weiteren Komponenten der unteren Platine werden vorerst nicht verwendet. Eine spätere Version, welche das GSM-Modem nutzt, ist aber vorstellbar.

Dadurch, dass beide Platinen eigenständigen Produkten entstammen, besitzen beide separate Stromversorgungs- und Spannungswandlung-Komponenten. In Absprache mit der Hardware-Entwicklung, wurde sich für die Nutzung der Stromversorgungs-Komponenten der unteren Platine entschieden. Dies ist die logische Entscheidung, da sich auf dieser Platine der verwendete Controller befindet und die andere Platine in diesem Projekt nur Peripherie-Komponenten stellt. Auf eine Darstellung der VCC- und VDD-Leitungen in Abbildung 5 wurde verzichtet. Beide Platinen, inklusive ihrer Stromversorgungs-Komponenten, sind ursprünglich für einen Batterie- bzw. Akku-Betrieb entworfen worden, was sich ideal für dieses Projekt anbietet. Die verwendete Akku-Kapazität und Details wie die Display-Laufzeit oder Default-Helligkeit werden allerdings etwas Finetuning in Bezug auf die gewünschte Akku-Laufzeit erfordern, da für die untere Platine ursprünglich nie eine Display-Komponente vorgesehen war.

Abbildung 5 kann man außerdem die Schnittstellen bzw. Busse entnehmen, mit denen die verschiedenen Komponenten am Mikrocontroller angeschlossen sind.

Der FTDI-Chip ist Controller-seitig logischerweise per UART angeschlossen. In diesem konkreten Fall mit UART 4. Das Funkmodul und das Display werden per SPI 1 und 2 angesteuert. Beachtet werden muss dabei die Bidirektionale Verbindung von SPI 1. Das Funkmodul ist in der Lage Daten zurückzusenden bzw. ausgelesen zu werden. Das Display besitzt eine solche Funktionalität nicht. Über die Unidirektionale Verbindung SPI 2 werden lediglich Daten bzw. Kommandos an das Display gesendet, ohne dass eine Rückantwort erwartet wird. Die Unterscheidung zwischen Daten- und Kommandobytes ermöglicht eine separate Leitung, mit einem entsprechenden Low-/High-Pegel. Exakte Details zur Implementierung folgen in Kapitel 4. Das Funkmodul besitzt außerdem eine separate EXTI-Leitung (External Interrupt). Auf dieser Leitung bekommt der Mikrocontroller einen Interrupt, falls das Funkmodul Daten fertig empfangen oder versendet hat. Die Buttons besitzen ebenfalls alle eigene Interrupt-Leitungen, welche bei Button-Presses auslösen.

Bei der Konfiguration von SPI-Komponenten und bei der Nutzung von Interrupts, bieten sich verschiedene Möglichkeiten. Diese werden in den folgenden Kapiteln dargestellt und es wird erläutert, warum eine bestimmte Option verwendet wurde.

3.1.2 SPI-Protokoll

SPI (Serial Peripheral Interface) ist ein serielles Schnittstellen-Protokoll zur Kommunikation zwischen einem Mikrocontroller und einem oder mehreren Peripherie-Geräten. Diese werden in der Regel als "Slaves" bezeichnet. Der Controller generiert bei SPI ein Taktsignal (*SCLK*), ein Chip-Select/Slave-Select-Signal (*CS/SS*) und ein Serial-Data-Out/Master-Out-Slave-In-Signal (*SDO/MOSI*). Die angesprochenen Geräte antworten u.U. mit einem Serial-Data-Out-Signal (*SDO*) ihrerseits, welches auf Mikrocontroller-Seite als Serial-Data-In (*SDI*) oder Master-In-Slave-Out (*MISO*) bezeichnet wird [24].

Bei der Verwendung von mehreren SPI-Geräten bieten sich verschiedene Möglichkeiten die Geräte zu verschalten. Nennenswert sind dabei zwei Optionen. Erstens, alle SPI-Geräte über separate SPI-Lines zu betreiben und zweitens, die Geräte in einer Daisy-Chain zu verschalten.

Unter einer Daisy-Chain im Zusammenhang mit SPI ist das kaskadenartige hintereinander Schalten von Peripherie-Geräten gemeint. Der Output des ersten Gerätes wird zum Input des Zweiten, der Output des Zweiten zum Input des Dritten und so weiter (siehe Abbildung 6). Es existieren Bus-artige CS- und CLK-

Lines, welche vom Mikrocontroller kontrolliert werden. Der Output des letzten Gerätes wird zurück zum Controller geführt.

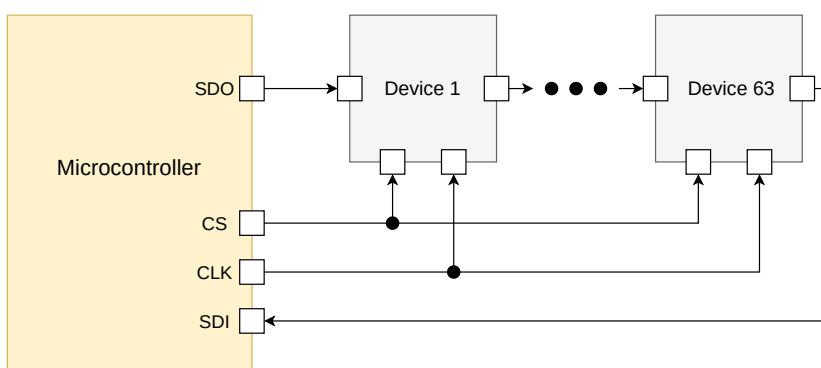


Abbildung 6: SPI Daisy-Chain Konfiguration,
Quelle: [25].

Ein Vorteil ist die (möglicherweise) hohe Geschwindigkeit, es kann dasselbe (hohe) Takt-Signal wie bei einem Gerät verwendet werden. Ein weiterer Vorteil ist das einfache Interface. Es werden nur eine SDO- und SDI-Line auf Mikrocontroller-Seite benötigt. Diese kann man leicht in Software verarbeiten und es werden weniger Pins benötigt, als wenn für jedes Gerät eine separate SPI-Line verwendet werden würden [24]. Ein Nachteil

ist, dass man ein Delay bei der Datenübertragung an die Slaves hat. Die Anzahl der Taktzyklen, welche zur Übertragung benötigt werden, ist proportional zur Position des Gerätes in der Daisy-Chain [26]. Dies ist besonders dann problematisch, wenn Bausteine eigentlich zeitgleich betrieben werden müssten, z.B. beim Kopieren von Daten von einem Baustein in einen anderen. Nachteilig ist außerdem, dass man beim Takt-Signal an das langsamste SPI-Gerät in der Kette gebunden ist.

Diese Daisy-Chain-Lösung funktioniert und skaliert auch nur bis zu einer bestimmten Anzahl an Geräten. Möchte man mehr Geräte mit einer SPI-Leitung ansprechen, existieren Lösungen mit mehreren Daisy-Chains. Dabei wählt man über den CS-Bus kein einzelnes Gerät mehr, sondern eine ganze Daisy-Chain aus und steuert alle Geräte in dieser an [24].

Insgesamt sind solche komplexeren Verschaltungen in diesem Projekt aber nicht notwendig. Es ist trotzdem wichtig, sich die Optionen bewusst zu machen, da selbst High-End-Controller nur eine begrenzte Anzahl an SPI-Lines besitzen. Dadurch, dass in diesem Projekt aber nur zwei SPI-Geräte verwendet werden, wurde sich für eine Lösung mit separaten SPI-Lines entschieden. Dies hat u.A. den Vorteil, dass man das Display (mit bis zu 20 MHz) deutlich schneller, als das Funkmodul (mit bis zu 1 MHz) takten kann [27] [22] und die Bausteine gleichzeitig betrieben werden können. Der höhere Takt ist praktisch, wenn man größere Datenmengen, wie bspw. Bild-Daten oder Animationen, an das Display übertragen möchte.

3.1.3 Interrupts

Auch Interrupts sind keine von vornherein festgelegte Technologie-Entscheidung. Man hat zwar bei den verwendeten Komponenten einige Vorgaben, bspw. löst das Funkmodul zwangsläufig einen Interrupt aus, bei den Buttons gäbe es aber auch durchaus andere Lösungsansätze, als 6 separate Interrupt-Leitungen zu verwenden.

Buttons sind eine Form von externem Input. Externer Input kann zufällig (zu einem unbekannten Zeitpunkt), periodisch, langsam oder schnell eingehen [18]. In diesem Projekt kommen sowohl die Button-Presses, als auch die Antwort vom Funkmodul zu vollkommen unvorhersagbaren Zeitpunkten. Die Alternative zur Verwendung von Interrupts wäre Polling. Bei Polling würde man in regelmäßigen Zeittintervallen den Zustand eines GPIO-Pins überprüfen und bei einer Änderung bzw. bei einem gewünschten Zustand in eine entsprechende Behandlung gehen. Ein Vorteil ist, dass dies erst einmal einfacher zu implementieren ist. Es handelt sich dabei um ein simpleres geistiges Modell (siehe Listing 2), als man dies bei asynchronen Interrupts hat.

```

1  while(true)
2  {
3      if (HAL_GpioReadPin( port , pin ) == GPIO_PIN_SET)
4      {
5          // Do something
6      }
7
8      HAL_DelayMS(200) ;
9  }

```

Listing 2: Polling für einen GPIO-State,
Quellen: [28].

Ein Nachteil von Polling ist der (große) Overhead. Polling, so wie es hier implementiert ist, ist blockierend. Der Prozessor, bzw. der Task in einem RTOS, blockiert und erledigt nichts Weiteres in der Zeit.

Ein weiteres Problem bzw. eine weitere Frage, die man sich bei nicht vorhersagbaren Ereignissen stellen muss, ist, wie regelmäßig man auf neuen Input überprüfen möchte. Im Falle der Button-Presses möchte man keinen verpassen, aber auch keinen doppelt verarbeiten.

In dem in Listing 2 implementierten Beispiel prüft man alle 200 ms auf einen gesetzten Pin. Button-Presses, welche dazwischen stattfinden, auch wenn dies schnell für einen Menschen wäre, würden nicht erfasst werden. Gleichzeitig prüft man aber auch permanent den Zustand eines Pins, selbst wenn kein Input erfolgt. Änderungen bei der Taktung des Prozessors, bei Bus- oder Peripherie-Takten, der Temperatur des Chips oder beim Energieverbrauch können auch ungewollte Änderungen beim Polling mit sich führen [18].

Einige Quellen listet als weiteren Nachteil von Polling die schwere “Debuggability“, da man den exakten Zeitpunkt des Inputs schwer replizieren kann. Meiner Meinung nach, sind Interrupts u.U. noch schwieriger zu debuggen. Interrupts können theoretisch immer auftreten. In jedem Code-Abschnitt, nicht nur in einem while(true)-Loop, könnte es zu einer Unterbrechung kommen und bei großer Software wird es extrem schwierig jeden “Edge-Case“ zu prüfen.

Ein wirklicher, weiterer Nachteil von Polling ist der Energieverbrauch. Durch die ständige Arbeit wird es schwer für den Prozessor, oder für den Scheduler bei einem RTOS-Task, Zeit zu finden, um in Sleep-Modes zu gehen. Interrupts sind an dieser Stelle deutlich effizienter. Sie bieten sich ideal dafür an, dass sich der Prozessor in einem Low-Power-Modus befindet und nur aufwacht, wenn es notwendig ist [18].

Im Kern sind Interrupts extern ausgelöste Service-Requests. Bei Eintreffen eines Interrupts unterbricht der Prozessor seinen gewöhnlichen Programmablauf und springt in die Abarbeitung einer Interrupt-Service-Routine. So ein Sprung ist nicht kostenfrei. Die ISR muss lokalisiert, Register müssen gerettet, die ISR muss ausgeführt und die Register müssen wiederhergestellt werden. Der Prozessor arbeitet in der Zeit auch an nichts anderem. Die Routinen sollten deshalb so kurz wie möglich gehalten werden [18]. Idealerweise setzt man nur eine boolsche State-Variable und führt eine Auswertung zu einem späteren Zeitpunkt durch [6].

Dies bietet sich ideal für die Button-Auswertung an. Eine State-Machine im Main-Loop würde die entsprechende Auswertung und ein Aufrufen nachfolgender Prozeduren erledigen. Konkrete Implementierungsdetails folgen in Kapitel 3.2.1. Wie vorhin bereits erwähnt, sind die Buttons nicht in Hardware entprellt. Dies muss bei der Implementierung der ISR beachtet werden. Buttons könnten prellen und dies könnte ein sehr schnelles mehrfach Auslösen des Interrupts zur Folge haben. Dies wiederum könnte zu unvorhersagbaren Fehlern führen, welche schwer rekonstruierbar sind, da sie nicht immer auftreten würden und das letzte Event nicht immer dem tatsächlichen Status (gedrückt/nicht gedrückt) entsprechen muss. Lösungsansätze dafür werden ebenfalls in Kapitel 3.2.1 besprochen.

3.2 Software

Mit einem nun vorhandenen Hardware-Prototypen, geht es an die Planung der Software-Architektur. Ähnlich zum vorherigen Kapitel, wird es einen Highlevel-Überblick über die Projektstruktur geben, bevor sich Design-Entscheidungen gewidmet wird. Auch eine Schnittstellenbeschreibung des zu schreibenden Treibers und eine Erläuterung des verwendeten Software-Stacks folgt in diesem Kapitel.

3.2.1 Projektstruktur

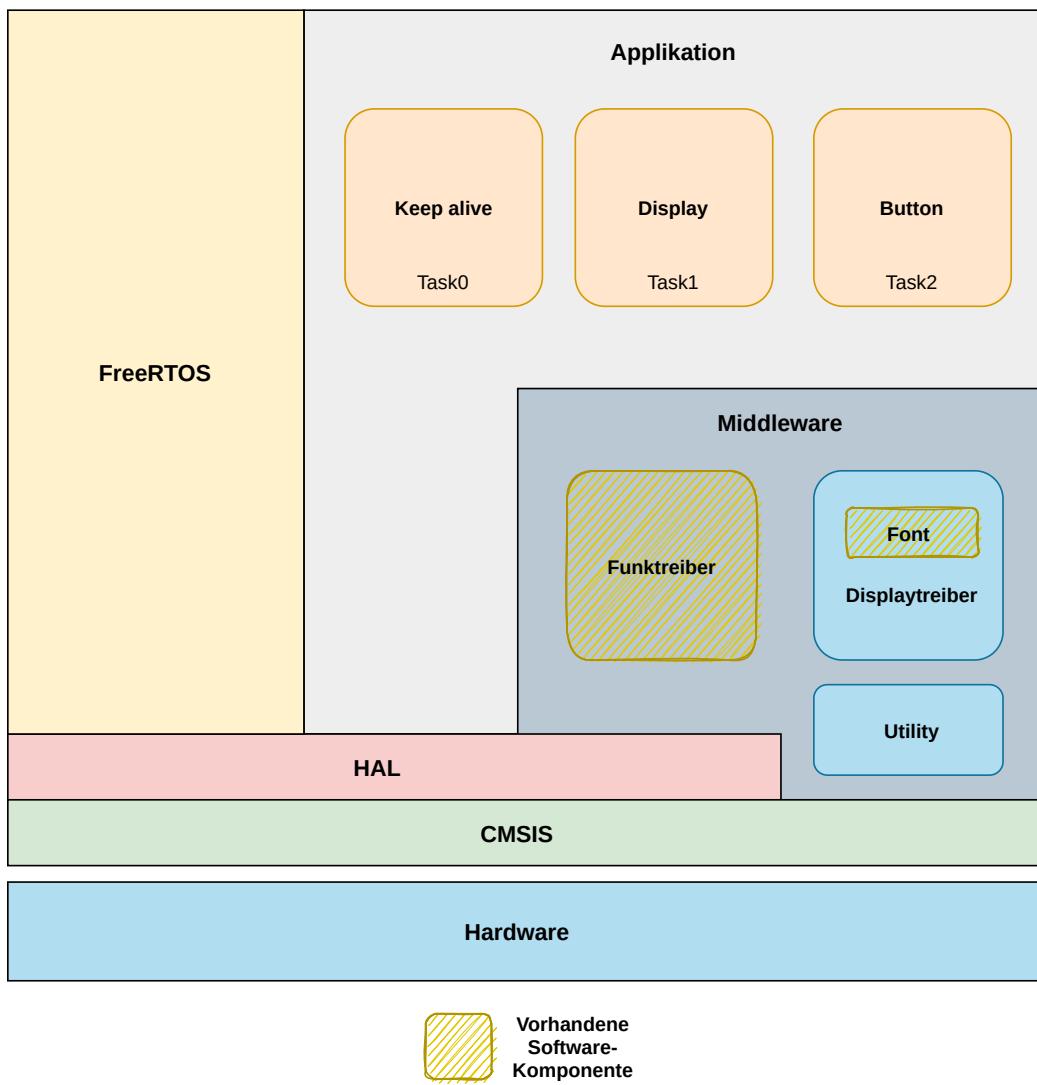


Abbildung 7: Software-Komponenten im Projekt,
Quelle: [29].

Abbildung 7 kann man einen Überblick der verschiedenen Software-Module, mit ihrer Position im Technology-Stack entnehmen. Die Grundlage der Software und eine Plattform, auf der sie laufen kann, bietet die Hardware. Diese wurde, zusätzlich zu CMSIS und HAL, bereits vorgestellt.

Aufbauend auf dem Hardware-Abstraction-Layer und sich über die ganze Applikation erstreckend, befindet sich das Echtzeitbetriebssystem *FreeRTOS*.

Warum sollte man überhaupt ein Echtzeitbetriebssystem nutzen?

Die Alternative wäre die *Bare-Metal-Programmierung*.

Unter *Bare-Metal-Programmierung* versteht man das Schreiben von Programm-Code, für eine spezielle Hardware-Plattform, ohne ein dazwischenliegendes Betriebssystem. Die Vorteile sind kleinere, energieeffizienter und schnellerer Code. Bei Mikrocontrollern mit besonders kleinem Speicher, kommt diese Art der Programmierung oft zum Einsatz. Die Nachteile sind, dass man alle Funktionalitäten, die einem ein Betriebssystem bietet, falls man sie benötigt, selbst implementieren muss [30]. Zusätzlich ist die Entwicklung durch die exakte Abstimmung auf die Hardware zeitintensiver und der Code ist nicht portabel. Einige Beschreibungen der Bare-Metal-Programmierung gehen einen Schritt weiter und definieren neben dem Verzicht auf ein Betriebssystem, einen Verzicht auf jegliche Hardware-abstrahierende Schichten [31]. Die Vor- und Nachteile sind inhaltlich dieselben, fallen nach dieser Definition aber noch stärker aus.

Ein Echtzeitbetriebssystem bietet einem, wie ein normales Betriebssystem erst einmal auch, Mechanismen zum Erstellen und Kontrollieren von Tasks bzw. Threads. Auch Kontext-Switching-Funktionalitäten, verschiedene Scheduling-Algorithmen, Mechanismen zum Allokieren von System-Ressourcen und Kontrollstrukturen für Multitasking stehen einem zur Verfügung. Auch verschiedene User-Level- bzw. prioritätsbasierte Funktionalitäten bietet einem ein OS. Gerade letztere sind von hoher Bedeutung im Embedded-Bereich. Verschiedene Tasks haben oft auch verschiedene Echtzeit-Anforderungen. Ein RTOS garantiert diese [32].

Gleichzeitig bieten einem ein RTOS als Grundlage, eine skalierbare Plattform mit vielen vorinstallierten Funktionen und Mechanismen. Größere Projekte könnten bspw. eine Circular-Queue oder ein Event-Message-System benötigen. Es ist praktisch, wenn diese bereits vorinstalliert sind und eine Integration vorgesehen ist. Die Aufteilung von Aufgaben in Tasks fördert die Modularisierung und bietet eine klare Struktur bei der Programm-Entwicklung. Gleichzeitig sind asynchrone, von einem Scheduler aufgerufene Tasks, gerade in der Embedded-Entwicklung sehr praktisch. Im letzten Kapitel wurde über blockierende/nicht-blockierende Programm-Abschnitte und den Einsatz von Interrupts gesprochen. Die Aufteilung in Tasks bietet einem die Möglichkeit, dass das ganze System reaktiv bleibt, obwohl ein einzelner Task gerade blockiert (bei Einsatz eines entsprechenden Scheduling-Verfahrens bspw. *Round-Robin* mit 10ms Ausführungszeit).

Die starke Förderung der Komponenten-Modularisierung, die Verantwortungs-Trennung, die Skalierbarkeit des Systems und die Tatsache, dass man ein Gerät hat, welches zwangsläufig reaktiv bleiben muss, haben zur Entscheidung geführt, ein Echtzeitbetriebssystem zu verwenden. Der STM32L471 erfüllt mit seinem 1-Mbyte Flash-Speicher und 128-Kbyte RAM (siehe Abbildung 3) auch jegliche Hardware-Voraussetzungen die ein RTOS stellt.

Als RTOS wurde sich *FreeRTOS* entschieden. Es handelt sich dabei um ein populäres Echtzeitbetriebssystem, welches unter der permissiven MIT-Lizenz veröffentlicht wird. *FreeRTOS* ist leichtgewichtig, unterstützt ein breites Spektrum an Hardware und wird hier im Unternehmen seit Jahren als Standard neben *Keil RTX* eingesetzt [33]. Die Integration von *FreeRTOS* ist bei *STM32*-Mikrocontrollern unkompliziert und in der *STM32*-Konfigurationssoftware *STM32CubeMX*, bei der Nutzung von CMSIS, bereits vorgesehen. Implementierungs- und Konfigurationsdetails folgen in Kapitel 4.1.

Verwendet werden die RTOS-Funktionalitäten von der Applikation. Eine Applikation hat viele Aufgaben. Das Starten von Funktionen, die Verarbeitung von Interrupt-Service-Routinen, das Starten von Tasks-/Threads und das Ansprechen von diverser physikalischer Peripherie über Geräte-Treiber [32].

Eine RTOS-Applikation teilt sich in zwei Bereiche auf. Einmal in den Teil, bevor man den Scheduler startet, und einmal in den Teil danach. Der Grund liegt darin, dass man keine Kontrolle mehr hat, wann ein bestimmter Task läuft, sobald der Scheduler einmal aufgerufen wurde. In dem Teil vor dem Scheduler-Aufruf, läuft Setup-Code. Man wartet auf eine stabile Stromversorgung, initialisiert Peripherie und erstellt

Tasks. Startet man nun den Scheduler, ruft dieser nach dem eingestellten Algorithmus und nach den eingestellten Prioritäten die Tasks auf. Ein Task ist nichts anderes als ein while(true)-Loop, der nicht returnen darf. Er muss explizit gelöscht werden [33].

Ich habe mich in diesem Projekt für die Erstellung von drei Tasks (Keep-Alive, Display und Button) entschieden (siehe Abbildung 7). Ein vierter Task, der Funk-Task, wird vom Funk-Treiber zur Verfügung gestellt. Abbildung 8 kann das Datenflussdiagramm der Anwendung entnommen werden, welches den Datenfluss innerhalb des Taskgebildes beschreibt.

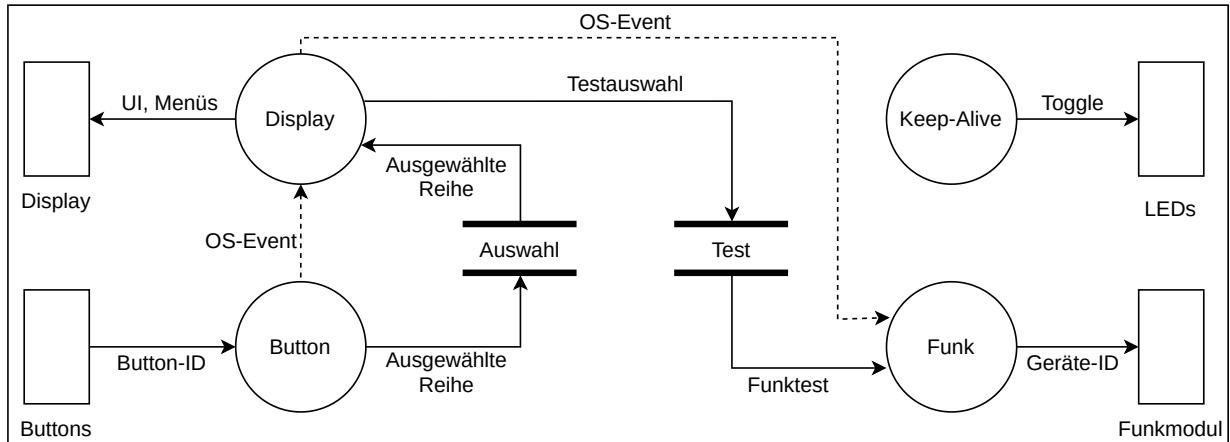


Abbildung 8: Datenflussdiagramm der Anwendung,
Quelle: [34].

Der Button-Task ist eine einzige State-Machine. Dort werden die Zustände, welche durch die Interrupts gesetzt werden, verarbeitet und zurückgesetzt. Eingaben werden bei einer Zeilenauswahl ausgewertet und an den Display-Task weitergeleitet.

Der Display-Task dient der UI-Darstellung und dem Update des Displays. Auch Menü-Wechsel und Fortschrittsanzeigen werden in diesem Task verarbeitet und dargestellt. Intern arbeitet eine State-Machine verschiedene Zustände, in denen man sich befinden kann, ab und aktualisiert die Anzeige entsprechend. Es ist auch der Display-Task, der Kenntnis darüber hat, in welchem Menü man sich befindet und welche Option als letztes angewählt wurde. Der Display-Task verwendet den von mir geschriebenen Display-Treiber, welcher im übernächsten Kapitel vorgestellt wird. Bei Auswahl eines Tests, nimmt der Display-Task Kontakt zum Funk-Task auf.

Der Funk-Task ist ein asynchroner Task, welcher vom Funk-Treiber zur Verfügung gestellt wird. Der Funk-Treiber ist Teil der Middleware und in diesem Projekt bereits vorhanden. Durch das Senden von speziellen OS-Events, wird der Funk-Task angesprochen, welcher dann wiederum über den Funk-Treiber das Funk-Modul anspricht. So können durch Übergabe, bspw. der Geräte-IDs, Devices in der Nähe angefunkt werden.

Der Keep-Alive-Task lässt die drei On-Board-LEDs (siehe Abbildung 5) blinken. Während der Entwicklung dienen diese LEDs dem Debugging. Im späteren Gerät ist eine Anzeige für ON (Grün), Standby (Gelb) und Fehler (Rot) geplant.

Einen genauen Einblick in die Tasks wird es in Kapitel 3.2.4 geben.

Als letztes kann man Abbildung 7 die Middleware-Schicht entnehmen.

Diese Middleware- bzw. Treiber-Schicht beinhaltet alle grundlegenden Funktionen, von denen die Applikation Gebrauch macht. In diese Schicht entfallen jegliche Peripherie-Treiber, die für dieses Projekt benötigt werden. Auch Utility-Funktionen und Low-Level-Funktionalitäten, welche man implementiert und aus der Applikation aufruft, gehören in diese Schicht. Das Ziel einer solchen Schicht, ist die unterliegende Hardware weiter zu abstrahieren und eine einheitliche Schnittstelle zur Peripherie zu bieten.

Treiber benötigen tiefergehende Kontrolle über die Hardware als die Applikation. Der Übergang zwischen HAL und CMSIS, zwischen Hardware-API-Aufruf und direkter Registermanipulation ist dort fließend. Ein Zugang zu beiden Programmier-Hierarchien ist deshalb gegeben (siehe Abbildung 7).

Der Funk-Treiber ist die große Software-Komponente in diesem Projekt, welche gestellt wurde. Es handelt sich dabei um einen firmeninternen Geräte-Treiber, welcher die Softwareseite des Microchip-Funkmoduls *MRF89XAM8A* und das hauseigene Binärprotokoll implementiert. Für den europäischen Markt findet der Funk auf 868 MHz statt. Zur Integration des Treibers müssen eine SPI-Schnittstelle, einige GPIO-Pins und das RTOS initialisiert sein.

Ein weiterer Teil der Middleware sind die Utility-Funktionen. Darunter fallen u.A. UART-printf-Funktionen und diverse Error-Handler. Die Funktionen in diesem Modul dienen hauptsächlich Debugging- und Analysezwecken und sollen die Entwicklung unterstützen.

Der Display-Treiber stellt, mit den Tasks, den größten Teil des zu schreibenden Codes in diesem Projekt. Es handelt sich dabei um die Implementierung der Softwareseite des Display-Moduls *DOGM128-6*. Die genaue Umsetzung, eine Beschreibung der geplanten API und eine Erläuterung der Font-Library, findet sich im übernächsten Kapitel wieder. Zunächst muss geplant werden, nach welchem Standard der Code geschrieben werden soll.

3.2.2 Programmierrichtlinie

Es gibt viele Programmierparadigmen in der Software-Entwicklung und die Programmiersprache C lässt einem viele Freiheiten, diese auch umzusetzen. Aufgrund dieser recht losen Natur der Sprache haben sich im Laufe der Jahre Regelwerke, welche einen strikteren C-Standard definieren, ergeben.

Der ISO-C-Standard erlaubt beispielsweise eine erhebliche Menge an Variationen zwischen Compilern. So existieren "implementation-defined", "unspecified" und "undefined" - Verhaltensweisen. Dies führt dazu, dass selbst identische Quellcode-Dateien, welche mit zwei konformen ISO-C-Compilern kompiliert wurden, ein verschiedenes Laufzeitverhalten haben können [35].

Bei einem Coding-Regelwerk handelt es sich um Konventionen und Vorschriften, an die sich ein Programmierer hält, welche aber nicht zwangsläufig vom Compiler erzwungen werden. Die Einhaltung von standardisierten Regelwerken kann allerdings von statischen Code-Analyse-Tools überprüft werden. Ein solches Regelwerk dient der einer Qualitäts- und Zuverlässigkeitsteigerung der Software. Auch Wartbarkeit, Lesbarkeit und Wiederverwendbarkeit von Programm-Code werden gesteigert. Das Ziel ist eine vollständige Compiler-Unabhängigkeit des Quellcodes [6].

Ein solches Regelwerk ist der MISRA-C-Standard.

Bei MISRA-C (Motor Industry Software Reliability Association) handelt es sich um den Programmierstandard der Automotive-Industrie. MISRA-C definiert über 100 verschiedene Regeln [6]. Definiert wird ein sichereres Subset der Programmiersprache C, welches aber auch deutlich restriktiver ist. Die MISRA-C-Guidelines befinden sich in ihrer dritten Edition und sind seit über 20 Jahren im Einsatz. Ähnliche Regelwerke existieren auch für C++, weil einem dort dieselben Probleme begegnen [35].

Hier im Unternehmen existiert eine eigene Coding-Richtlinie für die Programmiersprache C. Definiert sind circa 60 Regeln und Konventionen, viele davon MISRA-C konform. Insgesamt ist der Standard hier aber permissiver. Großen Wert wird auf die Namensgebung von Variablen, Funktionen, Dateien, sowie auf Modul-beschreibende Kommentarblöcke gelegt. Insgesamt ist ein in MISRA-C geschriebenes Modul, solange es sich an unsere interne Namensgebung hält, auch immer mit unserer Programmierrichtlinie konform.

3.2.3 Schnittstellenbeschreibung des Display-Treivers

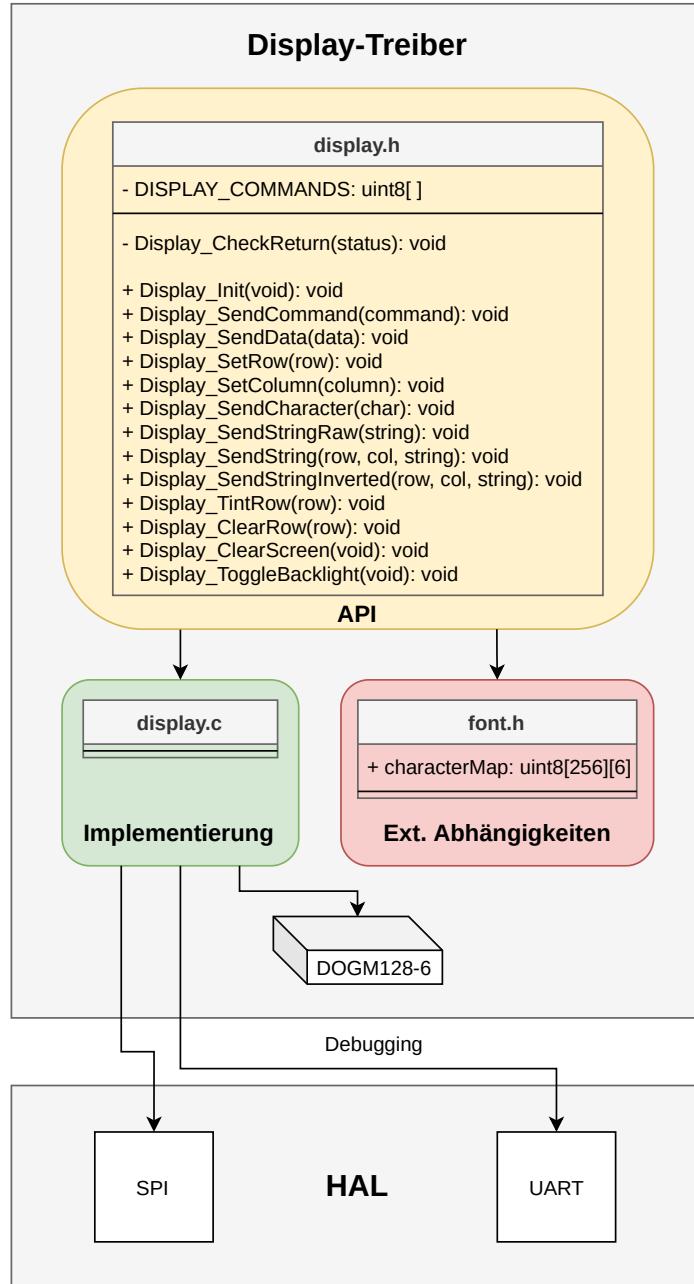


Abbildung 9: Schnittstellen-Entwurf des Display-Treibers,
Quelle: [36].

Mit einer nun definierten Coding-Richtlinie, geht es an die Konzeption der Display-Schnittstelle. Abbildung 9 kann man die Schnittstellenbeschreibung des Display-Treibers entnehmen. Wie in Kapitel 1.2 bereits beschrieben, ist das Ziel, eine wiederverwendbare Display-Schnittstelle zu schreiben, welche für dieses Projekt konkret den Treiber des Display-Moduls *DOGM128-6* implementiert.

Als eine gute Treiber-Schnittstelle wird firmenintern eine Schnittstelle definiert, welche:

1. Komplexität abstrahiert
2. Implementierungsdetails versteckt
3. Modular ist
4. Klar interne Abhängigkeiten kommuniziert
5. Über separate Debug-/Release-Modes (falls nötig) verfügt
6. “Easy-to-use“ ist

Zu (1) und (2): Der Plan ist so wenig interne Details wie notwendig preis zu geben.

Der Display-Treiber soll unabhängig vom darunter liegenden Hardware-Baustein verwendet werden können. Funktionen wie *Display_Init()*, *Display_SendString()* und *Display_ClearScreen()* sind vollkommen unabhängig vom Display-Modul.

Für Display-spezifische Kommandos werden “sprechende“ Namen bzw. Definitionen verwendet. Solche Kommandos sind notwendig, falls man das Display aus der Anwendung heraus konfigurieren möchte (bspw. für On/Off- oder Kontrasteinstellungen). Als Schnittstelle dient die Datei *display.h*. Die *display.c*-Datei (Umbenennung bei mehreren Display-Modulen) implementiert den konkreten Treiber.

Zu (3) und (4): Der Display-Treiber ist vollkommen modular und kann unabhängig vom Projekt verwendet werden.

Die einzige intere Abhängigkeit des Display-Treibers ist die Font-Library. Diese Abhängigkeit wird in der Header-Datei klar kommuniziert und muss für neue Display-Module entsprechend ergänzt bzw. ausgetauscht werden.

Die Font-Library ist eine gestellte Software-Komponente aus einem alten Projekt, welche ursprünglich Teil der mitgelieferten Firmware des Displays war. Die Font-Library enthält ein zweidimensionales Array, welches ein Mapping von ASCII-Codes auf die Display-spezifischen Byte-codes vornimmt. Die Font-Library enthält einen Teil des ASCII-Satzes und bietet einem ansonsten die Möglichkeit eigene Zeichen zu hinterlegen. Dort werden später auch einige Anpassungen, wie die Ergänzung von Logos und Loading-Animationen, erforderlich sein.

Zu (5): Separate Debug- und Release-Builds sind vorgesehen.

Bspw. wird die Funktion *Display_CheckReturn()* in der Debug-Version einen Error-Code über UART loggen. In der Release-Version wird dieses Feature aus Performance-Gründen abgeschaltet. Die Unterscheidung der beiden Versionen wird über “Compile-Schalter“ sichergestellt.

Zu (6): Dies wird die Nutzung durch mich, meine Kollegen und eine abschließende Code-Review zeigen.

Mit dem nun vorhandenen Entwurf eines Display-Treibers geht es an die Planung der Tasks, welche diesen verwenden werden.

3.2.4 Interner Aufbau der Tasks

Die allgemeine Projektstruktur und der Aufbau der Applikation wurden in Kapitel 3.2.1 bereits betrachtet. In diesem Kapitel folgt der Blick in die Tasks. Die Funktionsweise, der interne Aufbau und die Inter-Task-Kommunikation werden für den Button-, Display- und Funk-Task beleuchtet. Auf eine Betrachtung des Keep-Alive-Tasks, welcher keine interne Logik besitzt, wurde verzichtet.

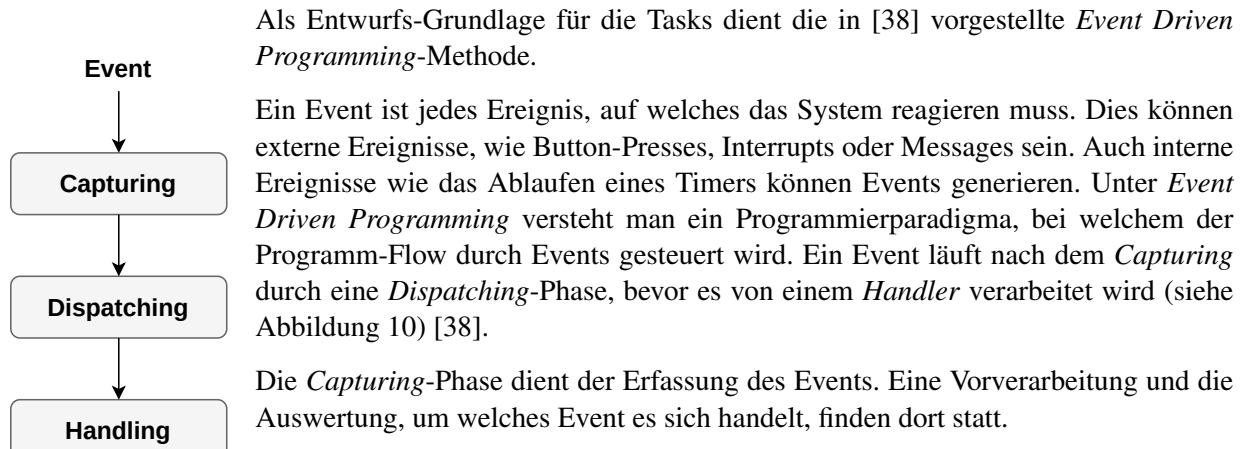


Abbildung 10:
Event Driven
Program,
Quelle: [37].

Die *Capturing*-Phase dient der Erfassung des Events. Eine Vorverarbeitung und die Auswertung, um welches Event es sich handelt, finden dort statt.

In der *Dispatching*-Phase wird der passende Handler für das Event ermittelt und aufgerufen. Auch eine Fehlerbehandlung, falls das Event nicht verarbeitet werden kann, würde hier stattfinden.

Die *Handlers* implementieren die Logik, welche im Fall eines bestimmten Events ausgeführt wird.

Unterschieden wird in zustandslose und zustandsbehaftete Event-Driven-Programs bzw. Zustandsautomaten. Zustandslosigkeit meint, dass das Programm unabhängig vom vorhergegangenen Zustand ist und sich diesen nicht merken muss. Ein zustandsbehaftetes Programm auf der anderen Seite ist abhängig vom vorhergegangenen Zustand und muss sich diesen stets merken [38].

In diesem Projekt finden ausschließlich zustandsbehaftete Event-Driven-Programs bzw. Zustandsautomaten Anwendung.

Unter einem endlichen Zustandsautomaten (engl. Finite State Machine, kurz FSM) versteht man ein Modell, welches aus einer endlichen Anzahl an Zuständen, Übergängen zwischen diesen und Aktionen, welche ausgeführt werden sollen, besteht. Jeder Zustandsautomat hat einen Ausgangszustand und definiert eine Menge an Regeln, welche für einen Zustandsübergang erfüllt sein müssen. Eingehende Events dienen als Auslöser für die Evaluation dieser Regeln [38].

Es gibt verschiedene Ansätze einen endlichen Zustandsautomaten zu implementieren [38]. Eine übliche Technik ist die Implementierung via Switch-Case-Statements [39], welche auch in diesem Projekt gewählt wurde.

An Events gibt es in diesem Projekt ausschließlich extern ausgelöste Interrupts. Dies sind zum einen die vom Funkmodul ausgelösten Interrupts und zum anderen die Button-Presses, welche Interrupts auslösen. Eine Begründung, warum Button-Interrupts gewählt wurden, wurde in Kapitel 3.1.3 gegeben. Auf den vom Funk-Modul ausgelösten Interrupt wird später im Kapitel eingegangen.

Die Button-Interrupts werden von den Interrupt-Handlern „genuine“. Die weitere Verarbeitung wird vom Button-Task übernommen, in welchem intern ein Zustandsautomat arbeitet. Diesen kann man Abbildung 11 entnehmen.

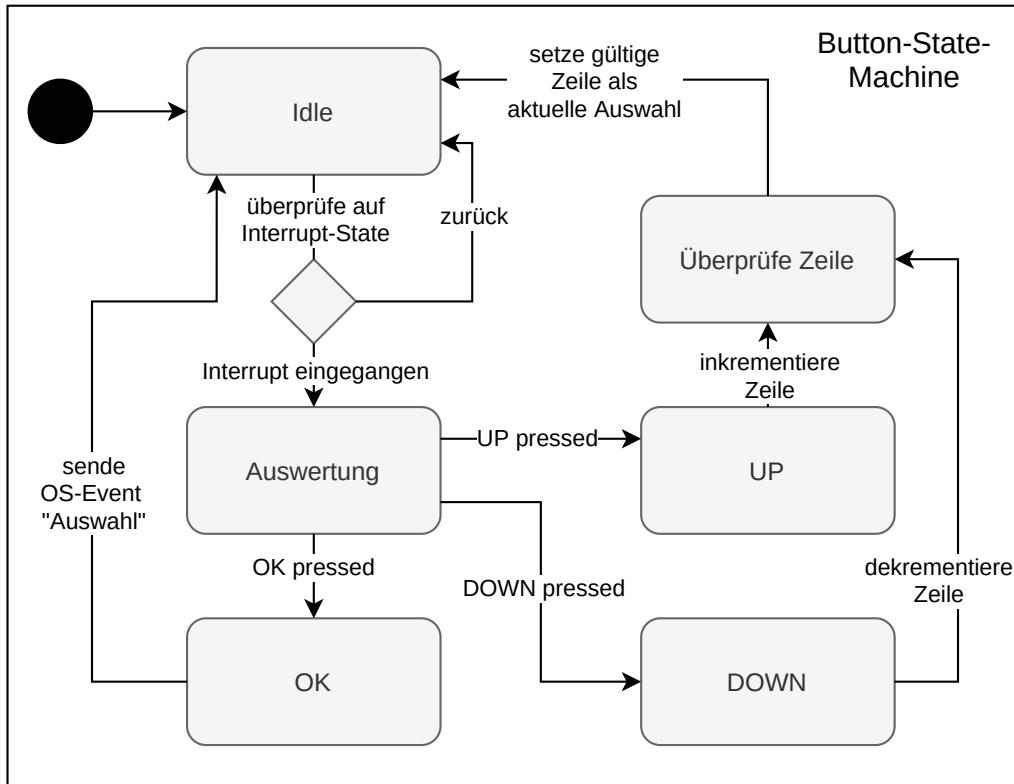


Abbildung 11: Interner Aufbau des Button-Tasks,
Quelle: [40].

Der Button-Task ist für das *Dispatching* und ein erstes Low-Level-*Handling* zuständig. Die Button-State-Machine beginnt im Idle-Zustand. In diesem befindet sie sich, bis eine boolsche Variable, welche nur ein Button-Interrupt setzen kann, gesetzt wurde.

Bei Eingang eines Button-Presses wird in den Zustand „Auswertung“ gewechselt.

Sollte der UP- oder DOWN-Button gedrückt worden sein, wird in den entsprechenden Folgezustand gewechselt. Abhängig vom Zustand wird die aktuell gewählte Zeile entweder inkrementiert oder dekrementiert. Danach wird in den Zustand „Überprüfe Zeile“ übergegangen. Dieser Zustand kennt die zuletzt und aktuell ausgewählte Zeile. Sollte sich die aktuelle Zeile außerhalb des Bildschirmrands befinden, wird die aktuelle Zeile auf die zuletzt gesetzte Zeile zurückgesetzt. Anschließend wird die aktuelle Zeile als „aktuelle Auswahl“ gesetzt.

Sollte der OK-Button gedrückt worden sein, wird ein OS-Event bzw. eine OS-Message „Auswahl“ mit der aktuellen Auswahl generiert.

Bei jedem der drei Button-Presses wird die boolsche Variable, welche der Interrupt gesetzt hat, zurückgesetzt.

Der Display-Task ist für die Menü-Auswahl, das Starten von Verbindungstests und das Update des Displays verantwortlich. Intern arbeitet im Display-Task ein Zustandsautomat, welcher Abbildung 12 entnommen werden kann.

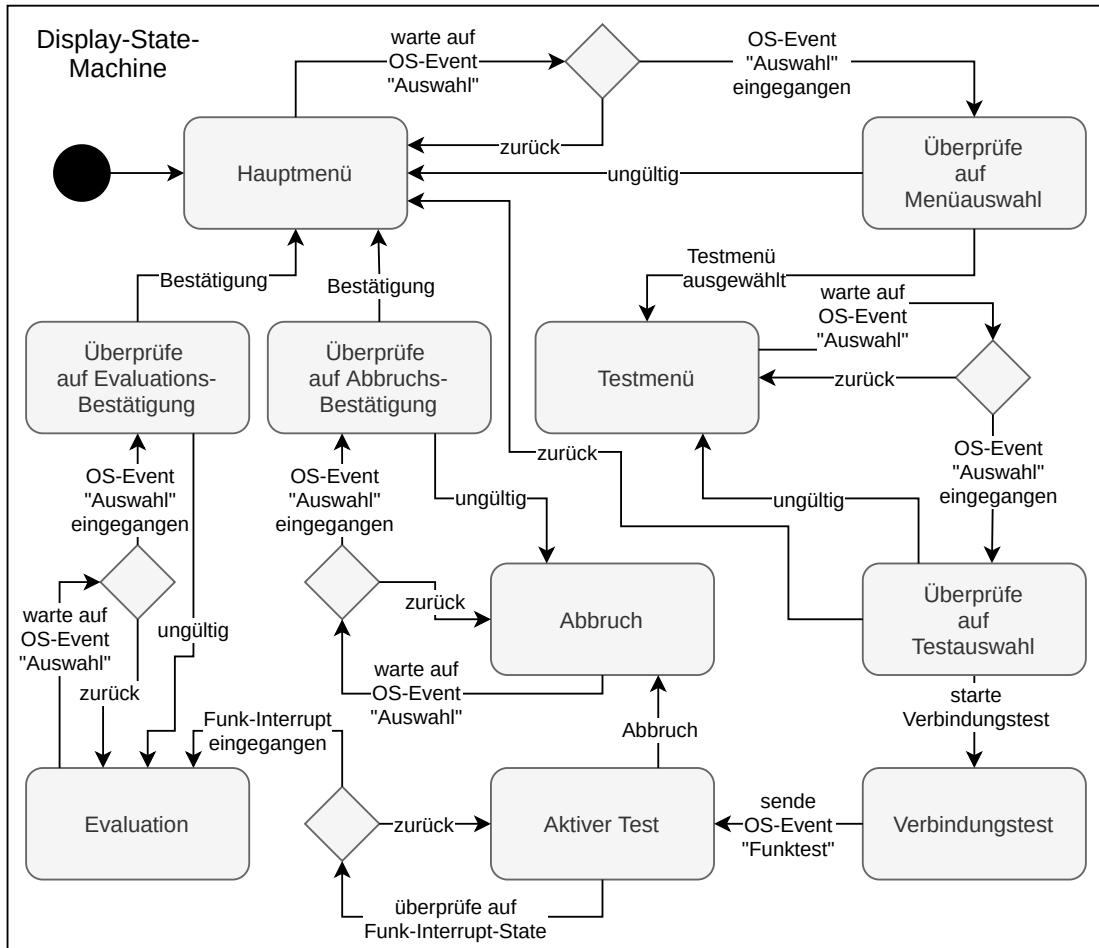


Abbildung 12: Interner Aufbau des Display-Tasks,
Quelle: [41].

Die Display-State-Machine beginnt im Zustand „Hauptmenü“. In diesem Zustand werden Firmenlogo, Geräte-Name und eine Auswahl an Optionen auf dem Display angezeigt. Es wird auf den Eingang des OS-Events „Auswahl“ gewartet, welches vom Button-Task gesendet wird.

Bei Eingang des OS-Events „Auswahl“ wird in den Zustand „Überprüfe auf Menüauswahl“ gewechselt. Sollte eine Zeile ausgewählt worden sein, auf der derzeit keine Funktionalität hinterlegt ist, gilt die Anfrage als ungültig und es wird zurück in den Zustand „Hauptmenü“ gewechselt. An der Anzeige auf dem Display ändert sich nichts. Sollte die Testmenü-Option, die derzeit einzige Funktionalität, ausgewählt worden sein, wird in den Zustand „Testmenü“ gewechselt.

In diesem Zustand wird die Anzeige des Displays aktualisiert. Es wird das aktuell gewählte Untermenü angezeigt. Dies besteht aus dem Firmenlogo, der Menü-Bezeichnung, der zur Auswahl stehenden Tests und einer „return“-Option. Der Zustand wartet auf ein OS-Event „Auswahl“. Bei Eingang des Events wird in den Zustand „Überprüfe auf Testauswahl“ gewechselt.

Dieser Zustand überprüft die ausgewählte Zeile. Sollte auf der ausgewählten Zeile eine Funktionalität hinterlegt sein, wird in den entsprechenden Nachfolge-Zustand gewechselt. Bei einer ungültigen Einga-

be wird in den Zustand "Testmenü" zurück gewechselt. Eine hinterlegte Funktionalität ist die "return"-Funktion, welche zurück in den Zustand "Hauptmenü" führt. Die zweite hinterlegte Funktionalität ist der "Verbindungstest". Bei Auswahl dieser Funktion wird in den Zustand "Verbindungstest" gewechselt.

Im Zustand "Verbindungstest" wird die Meldung "Connection test initialized" auf dem Display angezeigt. Es wird außerdem das OS-Event "Funktest" generiert und in den Zustand "Aktiver Test" gewechselt.

In diesem Zustand wird auf das Eingehen eines Funk-Interrupts gewartet. Er ist außerdem mit einem maximalen Timeout ausgestattet, um bei einem nicht Erreichen des Funk-Moduls/der Funk-Empfänger in den Zustand "Evaluation" wechseln zu können.

Bei Erreichen desTimeouts wird in den Zustand "Abbruch" gewechselt. Auf dem Display wird eine Fehlermeldung mit dem Abbruchsgrund angezeigt. Der Zustand wartet auf den Eingang des OS-Events "Auswahl". Bei Eingang des Events wird in den Zustand "Überprüfe auf Abbruchbestätigung" gewechselt. Bei einer Auswahl der Zeile "Bestätigen" wird in den Zustand "Hauptmenü" gewechselt. Bei jeder anderen Eingabe wird zurück in den Zustand "Abbruch" gewechselt und an der Darstellung auf dem Display ändert sich nichts.

Bei Eingang eines Funk-Interrupts wird in den Zustand "Evaluation" gewechselt. Dieser Zustand ruft Prozeduren des Funk-Treibers auf, um die vom Funkmodul gesendeten Daten auszulesen und auszuwerten. Abschließend wird eine Übersicht mit den Test-Ergebnissen, bspw. der Anzahl an erreichten Geräten, auf dem Display angezeigt. Danach wartet der Zustand auf den Eingang des OS-Events "Überprüfe auf Evaluations-Bestätigung". Bei Eingang des Events wird in diesen Zustand gewechselt. Der Zustand prüft ob die Zeile "Bestätigen" ausgewählt wurde. Bei Auswahl der Zeile wird in den Zustand "Hauptmenü" gewechselt. Bei jeder anderen Auswahl wird in den Zustand "Evaluation" zurück gewechselt und an der Darstellung auf dem Display ändert sich nichts.

Als letztes möchte in den Funk-Task betrachten. Der Funk-Task ist Teil des Funk-Treibers und muss von mir nur rudimentär angesprochen werden. Die Implementierung liegt bereits vor. Ein vereinfachter Aufbau kann Abbildung 13 entnommen werden.

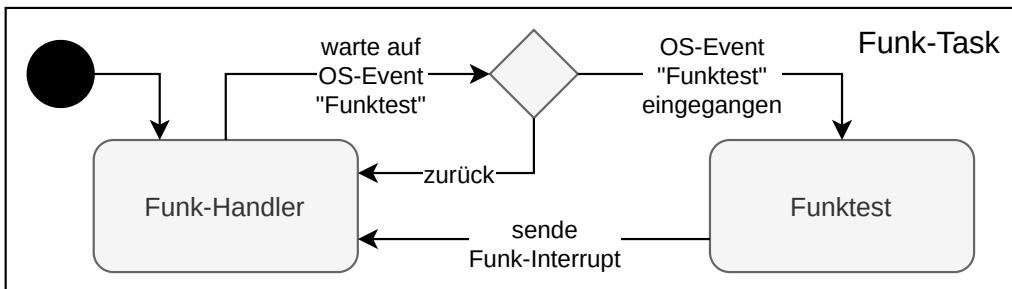


Abbildung 13: Interner Aufbau des Funk-Tasks,
Quelle: [42].

Der Funk-Task beginnt im Funk-Handler, in welchem auf das OS-Event "Funktest" gewartet wird, welches der Display-Task auslöst. Bei Eingang des Events wird ein Funktest initiiert und Geräte in der Nähe werden angefunkt. Bei Abschluss des Funktests sendet das Modul einen Interrupt. Der Funk-Handler kann dann die Daten zur Auswertung, bspw. wieviele Geräte erreicht wurden, über die SPI-Schnittstelle des Funk-Models auslesen. Über eine Schnittstelle werden diese Daten dem Display-Task verfügbar gemacht.

Im nächsten Kapitel folgt die Betrachtung des Software-Stacks, mit dem die vorgestellte Anwendung gebaut werden wird.

3.2.5 Software-Stack

Als Software-Stack bieten sich einem viele Möglichkeiten. Konkret wurden zwei Lösungen in Erwägung gezogen. Auf der einen Seite eine leichgewichtige Toolchain aus Texteditor mit Commandline-Tools und auf der anderen Seite eine vollwertige Embedded IDE.

Embedded IDEs wie *Keils uVision5*, *Visual Studio Embedded* oder *STM32CubeIDE* haben den Vorteil, dass Packages, welche zum Ansteuern des Mikrocontrollers benötigt werden, automatisch installiert werden. Außerdem kann man mit ihnen einfach zwischen verschiedenen Builds (Debug/Release) wechseln und sie bieten einen zentralen Ort, an dem programmiert, geflashed und debugged werden kann. Auch Makefiles und Projektdateien werden automatisch generiert. Solche IDEs sind dafür aber auch schwergewichtig und unter Umständen langsam.

Auf der anderen Seite steht eine Kombination aus einem Texteditor und Commandline-Tools. Ein solcher Software-Stack bietet große Flexibilität und volle Kontrolle über den gesamten Software-Building-Prozess. Ein solcher Software-Stack erfordert allerdings auch mehr Aufwand bei der Aufsetzung und Einarbeitung. Mächtige Texteditoren wie *Visual Studio Code* und *Atom* bieten durch Add-Ons mittlerweile IDE-ähnliche Entwicklungsumgebungen. Die Trennung zwischen Editor und IDE "verschwimmt" dadurch.

Als Software-Stack für dieses Projekt wurde sich für eine Mischung der beiden Möglichkeiten entschieden. Als Texteditor wurde *Visual Studio Code* gewählt. *VSCODE* ist schnell, bietet einem gute Add-Ons für die STM32/Cortex-Programmierung und ich bin mit den Shortcuts vertraut. Bei der Verwendung von *git* bietet einem der Editor weitergehende Möglichkeiten. So kann man direkt im Source-Code die Änderungen zwischen den Versionen sehen oder sich eine Übersicht über die Branches verschaffen.

Für die Code-Generierung wurde sich für *STM32CubeMX* von STMicroelectronics entschieden. Es handelt sich dabei um eine der *STM32CubeIDE* ähnliche Umgebung, welcher Compiling-, Flashing- und Debugging-Möglichkeiten fehlen. *CubeMX* ist dadurch leichtgewichtiger, bietet einem aber dieselben Code-Generierungsoptionen. Die Initialisierung von Peripherie, die Konfiguration von Taktquellen, die Installation von Gerätedateien und die Erstellung von Projekten mit HAL/CMSIS-Bibliotheken werden von diesem Programm übernommen. Für die Code-Generierung bietet einem *CubeMX* die Möglichkeit, die Kompilierungsumgebung, bspw. eigene Makefiles oder eine Embedded IDE, auszuwählen.

Für den abschließenden Teil des Software-Building-Stacks, wurde sich für *Keils uVision5* entschieden. Der integrierte Compiler, *ARMCC*, ist der Compiler, welcher hier im Unternehmen am häufigsten verwendet wird. Er bietet starke Optimierungsfunktionen. Gleichzeitig werden mögliche Kompatibilitätsprobleme bei der Integration des Funk-Treibers vermieden.

Als Tool zum Flashen und Debuggen des Mikrocontrollers bzw. der Applikation wird *Ozone* von *Segger* verwendet. Einen tieferen Einblick in dieses Tool und allgemein in den gesamten Debugging-Stack wird es im nächsten Kapitel geben.

3.3 Debugging

3.3.1 Debugging-Stack

Debugging ist in der Software-Entwicklung ein ebenso wichtiges Thema wie das Schreiben des Programm-Codes selber. Die richtigen Werkzeuge unterstützen bei der Suche nach Bugs und können helfen, die Qualität der Software zu erhöhen.

Beim Debugging von Embedded-Anwendungen stellen sich zusätzliche Herausforderungen. Dadurch, dass man Code entwickelt, welcher zwangsläufig auf einer anderen Maschine läuft, müssen andere Lösungen als beim x86-Debugging gefunden werden. In diesem Projekt kommt dazu, dass es sich bei der Hardware um einen Prototypen handelt. Dies fügt dem Debugging eine weitere Dimension hinzu, da man bei einem Fehler, anders als bei einem Evaluation-Board vom Hersteller, erst einmal nicht weiß, ob es sich um einen Software- oder Hardware-Bug handelt.

Bei der Fehlersuche in Embedded Systems bieten sich verschiedene Möglichkeiten.

Die erste Option wäre die *Simulation*. Darunter ist in der Regel eine MPU/MCU-Software-Simulation gemeint, in die man eine Binary laden kann. Die Simulation versucht den Prozessor und die internen Register exakt abzubilden. Je nach Simulation existiert auch I/O-Support. Die Vorteile der Simulation eines Embedded Systems ist der (oft) günstige Preis und die einfache Bedienung. Der große Nachteil ist, dass sich das System u.U. nicht so wie in der realen Welt verhält [18]. MPU/MCU-Simulatoren gibt es zwar viele, dennoch kann diese Lösung das System nie als ganzes prüfen.

Die zweite Option wäre die *Emulation*. Dabei emuliert dedizierte Hardware die MPU/MCU und interagiert mit dem I/O auf der Zielplatine [18]. Diese Option bietet einem bessere Debugging-Möglichkeiten für einen Hardware-Prototypen als die Software-Simulation. Software-Bugs können hier klarer von “Verdrahtungsproblemen“ getrennt werden. Nachteilig sind der hohe Preis, da zusätzliche, dedizierte Hardware benötigt wird und der größere Aufwand beim Aufsetzen der Debugging-Umgebung.

Die dritte Option wäre das *Onboard-Debugging*. Dabei wird die Software direkt auf dem Zielgerät laufen gelassen. Es existieren verschiedene Hard- und Software-Lösungen, welche einem bei dieser Art des Debuggings unterstützen. Der große Vorteil ist, dass sich die Hardware exakt wie in der realen Welt verhält, es ist die reale Hardware. Nachteilig ist allerdings, dass der Debugging-Prozess schwieriger ist. “Verdrahtungsprobleme“ können hier erst einmal nicht klar von Software-Fehlern getrennt werden, da man das System als “Ganzes“, in der Interaktion von Hard- und Software, betrachtet. Zusätzlich benötigen I/O-Komponenten u.U. eine aufwändigere Behandlung um separat debuggt werden zu können, speziell bei einem Hardware-Prototypen.

Trotz dieser Nachteile wurde sich für die dritte Option entschieden, da am Ende dieses Projektes ein bedienfähiger Prototyp stehen soll, welcher als Gesamtsystem, aus Hard- und Software, funktionieren muss.

Für das Onboard-Debugging benötigt man Hardware-Debugger. Diese nutzen Schnittstellen, wie JTAG oder SWD, um die Hardware auszulesen.

JTAG (Joint Test Action Group) ist die geläufige Bezeichnung für den IEEE-Standard 1149.1 und beschreibt eine herstellerunabhängige Debugging-Lösung. Ursprünglich für das Auslesen von Flip-flops in einer Schaltung entworfen, wurde der Standard mittlerweile signifikant erweitert [43]. SWD (Serial Wire Debug) ist ein ARM-spezifisches Debugging-Protokoll. Im Gegensatz zu JTAGs üblicherweise verwendetem 5-Pin-Interface, besitzt SWD nur 2-Pins. SWD ist moderner und versucht JTAG zu ersetzen. Im Gegensatz zu JTAG ist SWD bspw. in der Lage über einen Pin *printf*-Output via UART- oder Manchester-Protokoll zum PC zu verschicken [44].

Hardware-Debugger ermöglichen es einem, über eben genannte Protokolle, Speicher- und Registerwerte auszulesen oder die Ausführung des Programms durch Breakpoints zu unterbrechen [43].

Bei Hardware-Debuggern gibt es einige Optionen.

Bspw. besitzen STM32-Evaluations-Boards, wie man es in Abbildung 4 sehen kann, im oberen Drittel eine separate Platine, auf der sich ein dedizierter Hardware-Flasher/Debugger befindet. Diese Platinen können auch abgetrennt und unabhängig von dem ursprünglichen Board verwendet werden. Synergie-rend existieren auf PC-seite proprietäre Lösungen wie die Debugging-Suite der *STM32CubeIDE* oder Open-Source Lösungen wie *OpenOCD*. Solche Software ist PC-seitig notwendig, da der Cross-Debugger nicht direkt über die JTAG-/SWD-Schnittstelle kommunizieren kann. Software-Tools ziehen PC-seitig einen Debug-Server auf, mit dem der Debugger kommuniziert. Dieser Debug-Server übersetzt die Komman-dos des Debuggers in die notwendigen JTAG-/SWD-Befehle [43].

Neben integrierten Hardware-Debuggern, wie man sie auf Prototypen-Boards findet, existieren auch exter-ne Hardware-Flasher/Debugger. Eine populäre Option ist dabei das Produkt *j-link* von *Segger*. Ex-terne Hardware-Debugger sind deutlich teurer als Prototypen-Platinen, bieten einem aber auch stärkere Debugging-Fähigkeiten. *Segger* bietet bspw. ein gesamtes Debugging-Ecosystem aus Hard- und Softwa-re an. Dieses bietet einem u.A. Features wie kon-di-tionelle Breakpoints (welche Auslösen wenn ein bestimmer Wert geschrieben wurde), Code-Tracing, Code-Coverage-Analysen und Echtzeit-Dissasembly [46].

Aufgrund dieser extensiven Debugging-Fähigkeiten, der Tatsache, dass solche Hardware hier im Unterneh-men schon vorliegt und der Prototyp schwer genug zu debuggen wird, wurde sich für eine Debugging-Lösung von *Segger* entschieden.

Abbildung 14 kann man eine Übersicht des Debugging-Stacks entnehmen. Die Abbildung zeigt die Zusam-menarbeit zwischen dem PC, als Plattform zum Debuggen, dem Hardware-Flasher/Debugger *j-link*, wel-cher die Prozessorregister ausliest und dem Mikrocontroller, welcher die Binary ausführt.

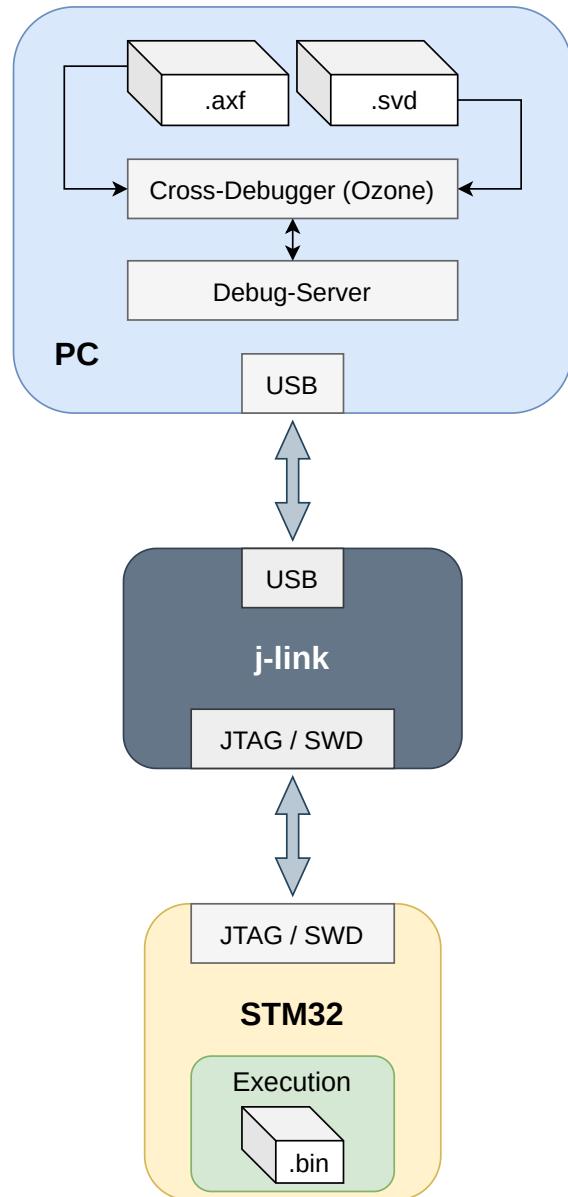


Abbildung 14: Übersicht des Debugging-Stacks,
Quelle: [45].

Ozone ist der Cross-Debugger von *Segger*. *Ozone* benötigt zum Debuggen eine axf-Datei und (optional) eine svd-Datei.

Die axf-Datei ist eine spezielle Datei, welche vom *ARM-Linker* generiert wird. Sie enthält neben der ausführbaren Binary, zusätzliche Debugging-Informationen, wie Zeilennummern und Einsprungadressen. Durch den Debugger kann man dann Watch- und Call-Stack einsehen [47]. Geflashed wird der Mikrocontroller nur mit der ausführbaren Binary.

Die svd-Datei ist eine vom Mikrocontroller-Hersteller zur Verfügung gestellte Datei, welche Speicherregionen mit den dazugehörigen Informationen (GPIO, SRAM, Flash etc.) mapped.

Diese zusätzlichen Informationen erleichtern das Debugging ungemein und erlauben u.a. den Zugriff auf gerätespezifische Peripherie-Register.

3.3.2 Messgeräte

Zusätzlich zum vorgestellten Debugging-Stack werden ein Oszilloskop (*Keysight MSOX2024A*) und ein Logic-Analyzer (*Kingst LA2016*) verwendet. Diese Werkzeuge sind unabdingbar bei der Arbeit mit einem Hardware-Prototypen. „Verdrahtungsfehler“ passieren schnell und wenn es nicht funktioniert ist es schwer herauszufinden, ob es an der Software oder an der Hardware liegt. Möglichkeiten die elektrischen Signale abzugreifen und unabhängig auszulesen, können sehr dabei helfen den Fehler einzugrenzen.

Ein Digital-Logic-Analyzer ist ideal zum Arbeiten/Debuggen von Mikrocontrollern geeignet. Er bietet deutlich mehr Pins, zum gleichzeitigen Analysieren von Signalen, als ein Oszilloskop. Dies ermöglicht einem komplexe Signalverläufe von Protokollen, wie bspw. SPI, I2C oder auch 8-Bit-parallele Busse, mitzuverfolgen. Zusätzlich ist eine Auswertung der Daten PC-seitig durch Software möglich.

Die Hauptaufgabe eines Oszilloskops ist die Messung von analogen Signalen.

Moderne Oszilloskope können allerdings noch wesentlich mehr. Sie können beispielsweise Signale selber generieren oder komplexe Protokolle wie USB in Echtzeit analysieren und darstellen.

Zur Analyse/Messung der Stromversorgung, z.B. der externen Batterieversorgung oder der Versorgungsleitungen für das Display, wird ein Oszilloskop verwendet.

4 Implementierung

In diesem Kapitel geht es um die Umsetzung des in Kapitel 3 vorgestellten Entwurfs. Es wird mit dem Aufsetzen der Projektstruktur und der Initialisierung der Peripherie-Komponenten begonnen. Diese bieten die Grundlage für das Coding, Compiling, Flashing und Debugging des Projektes. Danach wird die grundständliche Funktionalität der Hardware getestet. Es folgt darauf die Vorstellung und Erläuterung der Hauptfunktion des Utility-Moduls. Fortgefahren wird danach mit der Implementierung des Display-Treibers. Bis auf die API erforderte der Display-Treiber nicht allzu viel Konzeption, da man bei der technischen Umsetzung an die Vorgaben des Datenblattes gebunden ist. Weiter geht es dann mit dem Schreiben der Interrupt-Service-Routinen, welche auf die Buttons-Presses reagieren sollen. Dieses Kapitel wird recht kurz ausfallen, da hier nicht sonderlich viel Code zu schreiben ist. Im Gegensatz dazu steht das nachfolgende Kapitel, das Schreiben der Tasks. Der hier geschriebene Code macht, mit dem Display-Treiber zusammen, 90% des in diesem Projekt geschriebenen Codes aus. Die Tasks haben in Kapitel 3 auch den größten Teil der Planung ausgemacht.

4.1 Aufsetzen der Projektstruktur

Begonnen wird mit der Erstellung eines neuen Projektes in *STM32CubeMX*, dem Code-Generation-Tool von *STMicroelectronics*. Die Programmiersprache C und der verwendete Mikrocontroller, der *STM471VGT6*, werden ausgewählt. Es wird **keine** Platine ausgewählt, da es sich um einen Prototypen handelt und kein Evaluations-Board, welches von *STM* vertrieben wird, verwendet wird. In den weitergehenden Einstellungen kann man den Software-Stack spezifizieren. Dort wird ausgewählt, dass sowohl HAL, als auch CMSIS verwenden werden sollen. Außerdem wird ausgewählt, dass als Toolchain *Keils MDK-ARM* verwendet werden soll. Bei Abschluss erzeugt *CubeMX* ein Projekt mit allen benötigten HAL- und CMSIS-Libraries und einer Projektdatei für *Keils uVision5*, in welche diese schon eingebunden sind. Dann wird die nächste Ansicht geöffnet.

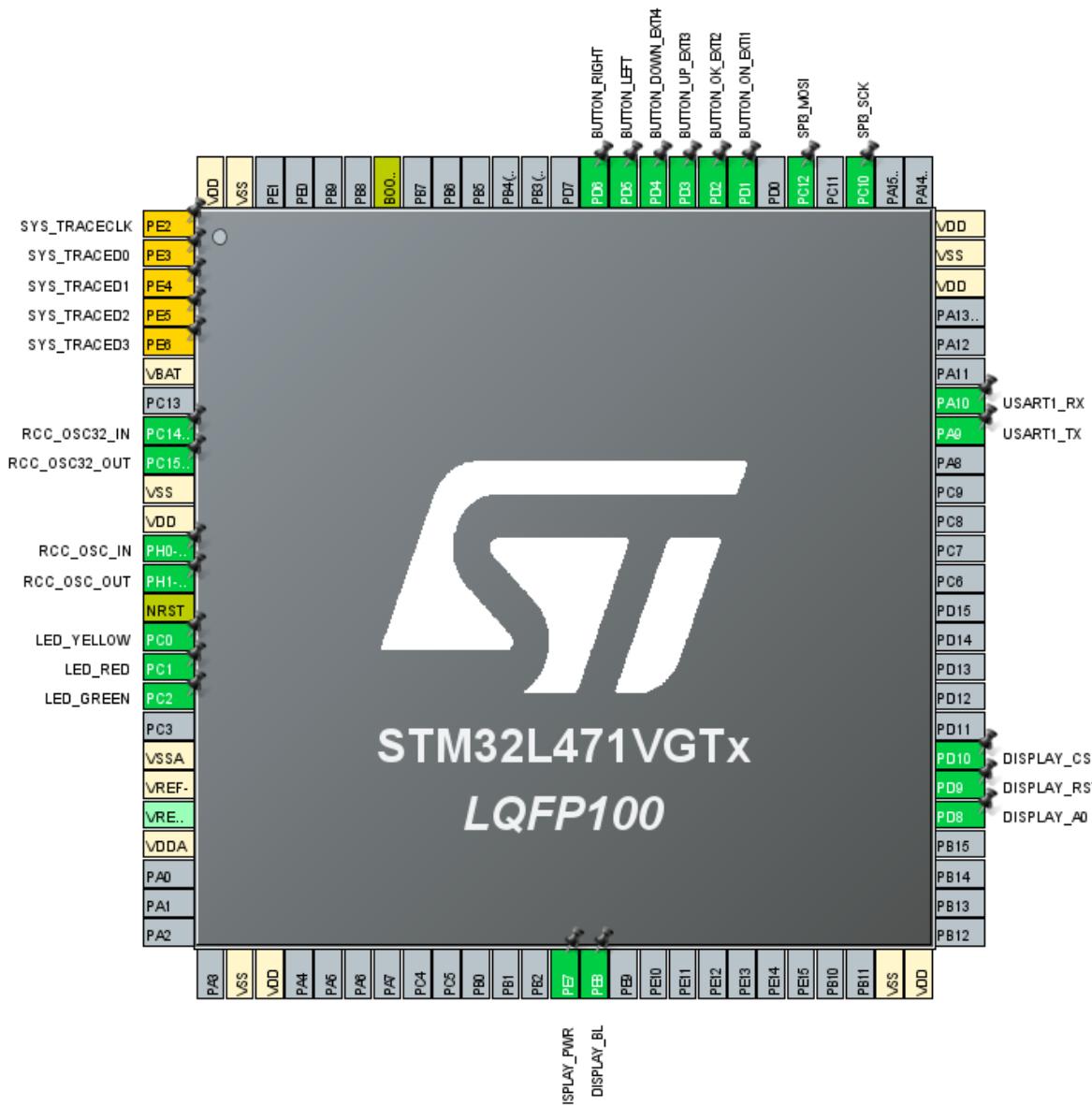


Abbildung 15: Konfiguration der Peripherie in CubeMX,
Quelle: [48].

In dieser werden einem Optionen zur Konfiguration der Peripherie bzw. Pins geboten. Auf der einen Seite

sieht man das Package des verwendeten Mikrocontrollers (siehe Abbildung 15) und auf der anderen Seite die möglichen Konfigurationen. Einem werden hier u.A. Möglichkeiten geboten Timer zu aktivieren, Schnittstellen zu nutzen oder A/D-Wandler einzuschalten.

Im Bezug auf das Prototypen-Datenblatt, wird die Peripherie konfiguriert.

Es werden die Onboard-LEDs (PC0-PC2), die Button-Interrupt-Leitungen (PD1-PD4), die Displaystromversorgung (PE7-PE8), die Displaysteuerung (PD8-PD10), als auch SPI- und USART-Schnittstellen aktiviert (vergleiche auch Abbildung 5).

Die Interrupts werden auf einer fallenden Flanke aktiviert und es wird angegeben, dass die Leitungen mit einem externen Pull-Up-Widerstand versehen sind. Für den Anfang ist keine Funktion für die Links- und Rechtstasten vorgesehen, weswegen die Interrupts auf diesen erst einmal deaktiviert werden. Die Pins welche mit RCC gekennzeichnet sind (siehe Abbildung 15, linke Seite) dienen als Input/Output für einen externen Quarz-Kristall, welcher als Taktgeber fungiert. Die Pins welche mit SYS_TRACE beginnen (PE2-PE6) sind standardmäßig aktiviert und dienen als Schnittstelle für Tracing- und Debugging-Funktionalitäten. Weitergehend sind etliche VSS- und VDD-Pins aktiviert.

Weiter geht es mit der Konfiguration der SPI- und USART-Schnittstellen. Wie in vorhergegangen Kapiteln bereits erwähnt, kann das Display mit einer maximalen Geschwindigkeit von 20 MBit/s getaktet werden. Ich habe mich anfänglich für eine Geschwindigkeit von 4 MBit/s entschieden, da schnelle Datenänderungen für Animationen, Grafiken o.ä. erst einmal nicht geplant sind und die Geschwindigkeit ausreichen sollte.

Weitergehend sind zusätzliche Konfigurationen an der SPI-Schnittstelle notwendig. Es handelt sich bei SPI um einen “De-facto-Standard“, weswegen sich die Implementierungen unterscheiden können. Dabei hilft ein Blick ins Datenblatt der verwendeten Peripherie-Komponente.

Der Display-Controller erwartet eine SPI-Kommunikation im Motorola-Format, mit 8-Bit Datengröße und dem MSB (Most Significant Bit) first [27]. Dies wird entsprechend konfiguriert. Weiter geht es mit den Einstellungen der Clock-Polarity (CPOL) und der Clock-Phase (CPHA) des SPI-Protokolls. Die Clock-Polarity entscheidet darüber, ob der SPI-Clock (SCLK) low oder high idlet. Der verwendete Display-Controller erwartet hier CPOL = 1, weswegen die Option auf high gesetzt wird. Die Clock-Phase entscheidet, auf welcher Taktsignal-Kante, der Ersten oder der Zweiten, die Dataline gesampled werden soll. Der Display-Controller erwartet ein Sampling auf der zweiten Kante, der Rising Edge [27]. Dies wird entsprechend konfiguriert.

Bei der Konfiguration der USART-Schnittstelle sind etwas weniger Parameter zu beachten. Man ist, je nach Pin, in der maximal möglichen Geschwindigkeit beschränkt, im Endeffekt müssen die Einstellungen der Baud-Rate, Word-Length und des Parity- bzw. Stop-Bits aber nur mit dem Terminal-Emulation-Programm auf PC-seite übereinstimmen, welches man auch konfigurieren kann. Konfiguriert wird USART mit der maximal möglichen Geschwindigkeit von 115200 Bit/s, bei einer Datengröße von 8-Bit, inklusive eines Parity-Bits.

Danach werden der Nested-Vectored-Interrupt-Controller (NVIC) und Timer6 aktiviert. Letzterer dient als Taktgeber für das RTOS. Die Wahl des Timers ist willkürlich, hier kann jeder der verfügbaren Timer verwendet werden, da alle Timer in der Lage sind, den gewünschten Basistakt von 1 ms zu liefern. 1 ms bietet sich als Basistakt an, weil die darauf aufsetzenden Software-Timer im RTOS entsprechend leichter konfigurierbar sind.

Die Aktivierung des RTOS findet man in *CubeMX* unter der Option *Middleware*. Dort wird *FreeRTOS*, mit *CMSISv1* als API und dem eben genannten Timer6 als Taktgeber, aktiviert. Hier könnte man auch zusätzliche Optionen wie Stackgröße oder Kernel-Tickrate konfigurieren. Diese Optionen werden auf ihren Default-Werten belassen. Die Default-Stackgröße von 128 Byte ist für die recht einfachen Tasks

ausreichend, dies wurde im Rahmen eines Integrationstests bestätigt. Hier könnte man außerdem Tasks, Semaphoren und Mutexe anlegen. Ich bevorzuge allerdings, diese später im Code “per Hand“ zu erzeugen. Als Scheduling-Verfahren stehen ein *Round-Robin*- und ein prioritätsbasiertes, unterbrechendes Scheduling-Verfahren zur Auswahl. Aufgrund der recht kurzen, einfachen Tasks, wird das prioritätsbasierte Scheduling-Verfahren gewählt. Die Tasks werden später alle mit derselben Priorität konfiguriert werden. So ist sichergestellt, dass alle Tasks ihre Arbeit komplett erledigen und nicht währenddessen unterbrochen werden können. Um ein Blockieren von Tasks und eine hundertprozentige CPU-Auslastung zu verhindern, wird jeder Task eine *osDelay*- oder *osWait*-Funktion integrieren, welche den Task in den entsprechenden Sleep- oder Waiting-State versetzt.

Dies schließt die Konfiguration der Peripherie ab.

Fortgesetzt wird mit der Taktung der CPU und der verschiedenen Busse.

Die STM32-Controller sind komplexe Highend-Mikrocontroller, welche viele verschiedene Peripherie-Komponenten, auf vielen verschiedenen Bussen bieten. All diese können aus verschiedenen Taktquellen gespeist und zu unterschiedlichen Taktraten getaktet werden. Um die Konfigurationen zu erleichtern, bietet *CubeMX* auch hier visuelle Einstellungsmöglichkeiten. Abbildung 16 kann einen Ausschnitt einer Taktfluss-Einstellung entnehmen.

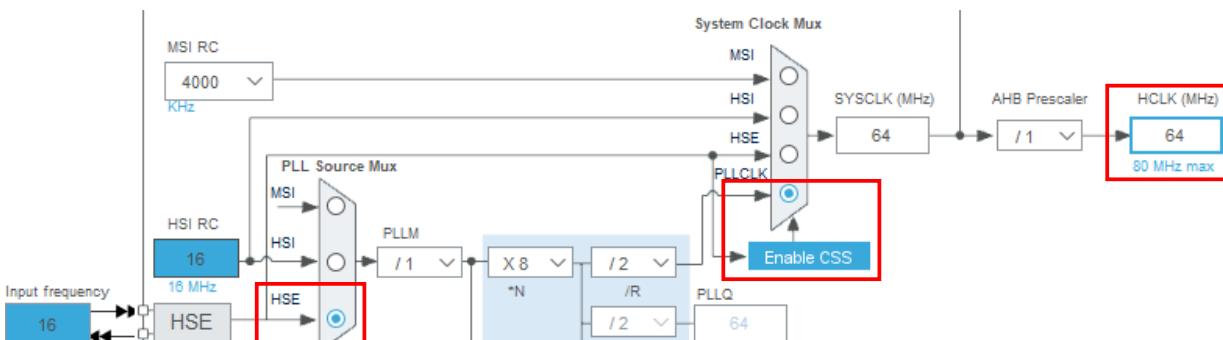


Abbildung 16: Konfiguration der Taktraten in CubeMX,
Quelle: [49].

HSE steht für *High Speed External* und beschreibt den vorhin angesprochenen externen Quarz-Kristall. Der Nachteil vom HSE ist, dass er stets ein externes Bauteil benötigt. Im Gegensatz zum HSI (High Speed Internal) braucht er auch etwas mehr Zeit, bis das Taktsignal stabil ist. Der HSE ist dafür sehr präzise [6].

Als CPU-Takt werden 64 MHz, der insgesamt möglichen 80 MHz, gewählt. Dies hat sich in anderen Projekten, in denen dieser Controller verwendet wurde, als guter Mittelweg zwischen Performance und Energieverbrauch herausgestellt.

Der verbaute Quarz liefert ein Taktsignal von 16 MHz. Die restlichen Einstellungen, wie Multiplikatoren, Teiler oder Bus-Taktraten ermittelt *CubeMX* dann automatisch. In Abbildung 16 kann man mit verfolgen, wie aus dem Eingangstaktsignal (16 MHz) durch den N-Multiplikator (8) und den R-Teiler (2) ein SYSCLK-Signal von 64 MHz entsteht. Dies ergibt nach einem Durchlauf durch den AHB-Prescaler (1) einen finalen CPU-Takt von 64 MHz. Als PLL bezeichnet STM ihre interne Clock-Generation-Engine. CSS steht für Clock-Security-System und beschreibt ein System, welches bei einem Oszillator-Fault greift, um ein “sauberes“ Herunterfahren der Hardware zu ermöglichen. Dazu generiert der Controller einen Hardfault-Interrupt, der von der Applikation entsprechend behandelt werden kann [50].

4.2 Funktionalität

4.2.1 Tests

Mit dem erstellten Projekt als Grundlage geht es an erste Funktionstests. Ziel ist die Sicherstellung der Funktionalität aller Peripherie-Komponenten inkl. ihrer korrekten Konfigurationen. Ich führe diese Tests bewusst sehr früh durch, da bisher keine Software geschrieben wurde und man so Implementierungsfehler, sofern sie sich nicht in der Treiber-Schicht befinden, ausschließen kann. Es erfolgen ausschließlich Aufrufe der HAL-API. Alle Tests werden in der *main()*-Funktion nach einer vorhergegangen Initialisierung der entsprechenden Komponente durchgeführt. Tabelle 3 kann man die durchgeföhrten Tests entnehmen.

Komponente	HAL-Funktion	Ziel	Ergebnis
On-Board-LEDs	HAL_GPIO_TogglePin	Blinken der On-Board-LEDs	Passed
USART1	HAL_UART_Transmit	Ergebnis in Terminal-Emulation-Prog.	Passed
Buttons	EXTI_IRQHandler	Auslösen von Interrupts	Passed
Displaystromver.	HAL_GPIO_WritePin	Anschalten d. Hintergrundbel.	Passed
Display-SPI	HAL_SPI_Transmit	Ergebnis auf Digi.-Logic-Analyzer	Failed

Tabelle 3: Durchgeföhrte Funktionstests.

Der erste Test diente der Sicherstellung, dass die korrekten LED-Pins initialisiert wurden und mit der Stromversorgung des Controllers alles stimmt. Die 3 LEDs haben sich wie erwartet verhalten (Blinken bei voller Helligkeit) und der Test wurde erfolgreich absolviert.

Der zweite Test sollte die Funktionalität der USART1-Komponente feststellen. Die Komponente wurde von *CubeMX* initialisiert, ein Buffer mit Daten gefüllt und über USART1 verschickt. Auf PC-Seite wurde mit einem Terminal-Emulation-Programm der Inhalt des Buffers empfangen. Der Test wurde bestanden.

Der dritte Test sollte die korrekte Verdrahtung der Buttons zwischen den Platinen und die Hinterlegung der Interrupts auf den Buttons über *CubeMX* testen. Nach dem Schreiben der entsprechenden Interrupt-Handler (EXTI 1-4) wurde über den Debugger festgestellt, ob bei einem entsprechenden Button-Press, der entsprechende Interrupt auslöst. Alle Interrupts „feuerten“ korrekt. Der Test wurde bestanden.

Der vierte Test sollte die Verwendung der korrekten Pins der Displaystromversorgung sicherstellen. Nach dem Einschalten des Display-Versorgungs-Pins, wurde der Display-Backlight-Pin getoggled. Die Hintergrundbeleuchtung wurde dadurch an- und ausgeschaltet. Das Display verhielt sich wie erwartet. Der Test wurde bestanden.

Der fünfte Test sollte die SPI-Kommunikation zwischen Controller und Display sicherstellen. Die Leitungen A0 (Data- or Command-Bit), CS (Chip-Select), SCLK (Serial Clock) und SDO (Serial Data Out) wurden mit einem Digital-Logic-Analyzer verbunden. Die Initialisierung wurde mit *CubeMX* vorgenommen. Ein Buffer wurde mit Daten gefüllt und per *HAL_SPI_Transmit()* verschickt. Die korrekte Funktionsweise der A0- und CS-Leitungen konnte sichergestellt werden. Auch das Taktsignal schien erst einmal gewöhnlich. Es war eine alternierende Bitfolge mit einer Frequenz von 4 MHz. Auf der SDO-Leitung erschienen allerdings keine Daten. Der Test schlug fehl.

Ohne eine funktionierende SPI-Kommunikation, kann nicht mit der Implementierung des Display-Treibers begonnen werden. Es folgt die Fehlersuche.

4.2.2 Debugging

Äußere Schäden, wie bspw. ein abgerissener Fädel-Draht, waren nicht zu erkennen. Weiterhin wurde überprüft, ob die Pins, welche konfiguriert wurden, korrekt waren und den Angaben im Datenblatt des Hardware-Entwicklers entsprachen. Dies schien der Fall zu sein. Es wurde dann damit weitergemacht, die Initialisierung der SPI-Schnittstelle im Debugger schrittweise mitzuverfolgen. Anhand des Datenblattes wurde überprüft, ob die richtigen Bits in den richtigen Konfigurationsregistern gesetzt wurden (Direction, Data, Pull-up/Pull-down etc.). Nach einiger Zeit, die HAL-Funktionen sind intern stark geschachtelt, konnte die Korrektheit der Initialisierung verifiziert werden. Es wurden die korrekten Bits, in den korrekten Registern gesetzt. Auch die von mir willkürlich gewählte Daten (0xAA), welche verschickt werden sollten, erschienen im TX-Buffer-Register und wurden versandt. Nur auf der Leitung erschien kein Ergebnis.

Als nächstes wurden die zu versendeten Daten auf 0xFF geändert. Damit verschwand das Taktsignal auf der Leitung. Der Pegel der Leitung war permanent high. Eine weitere Änderung der Daten, auf 0x7F, führte dazu, dass das Taktsignal für einen Zyklus low und dann wieder high war. Dies brachte Gewissheit. Die Datenleitung liegt auf der Taktleitung. Die Position der eigentlichen Taktleitung konnte in Software leider nicht festgestellt werden und die Vermutung lag nahe, dass dafür der Prototyp auseinander geschraubt werden müsste. Ich teilte meine Erkenntnisse dem entsprechenden Hardware-Entwickler mit und dieser prüfte die Verdrahtung.

Die Datenleitung lag tatsächlich auf der Taktleitung und die Taktleitung auf einem gänzlich anderen Pin, welcher nicht SPI-fähig war. Nach einer Korrektur der zwei Fädel-Drähte wurde die Verbindung erneut geprüft. Die Kommunikation erschien wie erwartet auf dem Logic-Analyzer. Der Test wurde bestanden.

4.3 Utility

Utility-Funktionen wurden bereits in Kapitel 3.2.1 vorgestellt und ihre Position im Software-Stack kann man Abbildung 7 entnehmen.

```

1 void UART_printf(UART_HandleTypeDef* huart, const char* format, ...)
2 {
3     //Create and initialize buffer
4     char buffer[64];
5     memset(buffer, 0, sizeof(buffer));
6
7     //Create variable argument pointer
8     va_list args;
9
10    //Initialize argument pointer
11    va_start(args, format);
12
13    //Format message accordingly and save it into buffer
14    vsnprintf(buffer, sizeof(buffer), format, args);
15
16    //Send the buffer out via UART
17    HAL_UART_Transmit(huart, (uint8_t*)buffer, sizeof(buffer), 100);
18
19    //Clean up
20    va_end(args);
21 }
```

Listing 3: Funktionsimplementierung: UART_printf(),
Quellen: [51].

In diesem Kapitel wird die *UART_printf()*-Funktion vorgestellt. Es handelt sich dabei um die wichtigste Debugging-Funktion des Utility-Moduls, von der in den nachfolgenden Kapiteln stets Gebrauch gemacht wird.

Listing 3 kann man den Quellcode der Funktion entnehmen.

Der Funktion wird ein UART-Handle, ein Formatstring und eine beliebige Anzahl an zusätzlichen Parametern mitgegeben. Der UART-Handle identifiziert eindeutig die verwendete UART-Schnittstelle. Der Formatstring ist ein im “printf-Style“ formatierter String, welcher als Template für die Variadic-Argument-Funktionen dient. Die zusätzlichen Parameter werden in Kombination mit dem Format des Strings zu einer Zeichenfolge zusammengesetzt und in einem Buffer abgespeichert. Dieser Buffer wird über die angegebene UART-Schnittstelle versandt.

```

1 #ifdef DEBUG_BUILD
2
3     if (*status != HAL_OK)
4     {
5         UART_Printf(&huart1, "Error Tx\r\n");
6     }
7
8     UART_Printf(&huart1, "cmd: %#02x\r\n", *cmd);
9
10 #endif

```

Listing 4: Funktionsaufruf: *UART_Printf()*,
Quellen: [51].

Listing 4 kann man eine Verwendung der Funktion entnehmen, wie sie im Debug-Build des Display-Treibers vorkommt.

4.4 Display-Treiber

In diesem Kapitel wird die Implementierung des Display-Treibers vorgestellt. Eine Übersicht über die geplante Schnittstelle konnte man Abbildung 9 entnehmen.

4.4.1 Allgemeines

Ein Treiber, speziell ein Hardware-Treiber, ist eine softwareseitige Implementierung der Befehle, welche eine bestimmte Hardware verarbeiten kann. Ein Treiber bietet eine Schnittstelle zur Hardware und stellt eine gewisse Abstraktionsschicht dar [52].

Wie bereits dargestellt, implementiert der Display-Treiber die Softwareseite des Display-Moduls *DOGM128-6*. Genauer gesagt implementiert der Display-Treiber die Softwareseite des LCD-Controllers *ST7565R*, welcher auf dem *DOGM128-6* verbaut ist. Das Display-Modulbettet den LCD-Controller ein und bietet eine Platine mit Stromversorgung und Display in einem üblichen Formfaktor an [19]. Der geschriebene Treiber würde somit für alle Displays (mit gleicher Größe) funktionieren, welche den *ST7565R*-Controller verbaut haben.

Alle technischen und für diesen Treiber relevanten Informationen, findet man im Datenblatt des LCD-Controllers.

4.4.2 Initialisierung

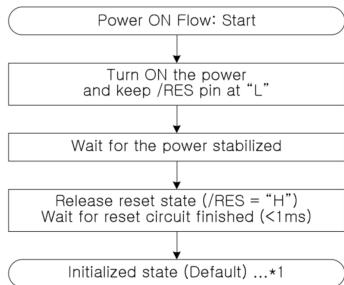


Abbildung 17: Beginn der Hardware-Initialisierung:

ST7565R,

Quelle: [53].

Um Befehle an das Display senden zu können, muss zuerst die Hardware initialisiert werden. Abbildung 17 kann man ein Verlaufsdiagramm entnehmen, welches den Beginn der Initialisierungsphase zeigt. Alle GPIO-Pins sind standardmäßig low geschaltet.

Begonnen wird mit dem Einschalten des Display-Power-Pins. Der Display-Reset-Pin ist active low und müsste zu diesem Zeitpunkt low sein. Er wird sicherheitshalber aber explizit low gesetzt. Nun wartet man darauf, dass sich die Circuit-Power stabilisiert. 200 ms haben sich bei diesem Display-Controller als guter Wert herausgestellt. Nach dem Release des Reset-Pins (Setzen auf high), wird 1 ms gewartet. Die Hardware ist damit initialisiert und bereit Befehle zu empfangen.

Listing 5 kann man die Implementierung des ersten Teils der Initialisierung entnehmen.

```

1 void Display_Init(void)
2 {
3     //Activate display power
4     HAL_GPIO_WritePin(DISPLAY_PWR_GPIO_Port, DISPLAY_PWR_Pin, GPIO_PIN_SET);
5
6     //Reset display (active low)
7     HAL_GPIO_WritePin(DISPLAY_RST_GPIO_Port, DISPLAY_RST_Pin, GPIO_PIN_RESET);
8
9     //Wait for stability
10    HAL_Delay(200);
11
12    //Release reset
13    HAL_GPIO_WritePin(DISPLAY_RST_GPIO_Port, DISPLAY_RST_Pin, GPIO_PIN_SET);
14
15    //Wait for reset (< 1ms – Display Controller ST7565R)
16    HAL_Delay(1);
17
18    {...}
19 }
```

Listing 5: Funktionsimplementierung: Display_Init() - Teil 1,
Quellen: [54].

Das Display erwartet nun Konfigurations-Befehle, bspw. für die Startreihe, die Displayorientierung oder den Kontrast. Abbildung 18 kann man einige Befehle entnehmen.

Command	Command Code								Function			
	A0	/RD	/WR	D7	D6	D5	D4	D3	D2	D1	D0	
(1) Display ON/OFF	0	1	0	1	0	1	0	1	1	1	0 1	LCD display ON/OFF 0: OFF, 1: ON
(2) Display start line set	0	1	0	0	1	Display start address					Sets the display RAM display start line address	
(3) Page address set	0	1	0	1	0	1	1	Page address			Sets the display RAM page address	

Abbildung 18: Konfigurations-Befehle: ST7565R,
Quelle: [53].

Bei den Kommandos handelt es sich um 8- oder 16-Bit-Befehle. Die A0-Leitung entscheidet darüber, ob das nachfolgende Byte als Befehls- oder als Datenbyte interpretiert wird. Die RD- und WR-Leitungen sind in der vorliegenden SPI-Konfiguration nicht angeschlossen und werden nicht verwendet. Es wird ausschließlich schreibend auf das Display zugegriffen. Details wie dieser Zugriff genau abläuft, folgen im nächsten Kapitel.

Bei den Kommandos entscheiden die ersten Bits darüber, um was für einen Befehl es sich handelt und die restlichen Bits definieren konkrete Details, welche den ausgewählten Befehl betreffen.

Für wiederverwendbare Kommandos wurden *extern declarations* in der Header-Datei des Display-Treibers angelegt. Die konkrete Implementierung, welches Byte der Befehl am Ende schickt, nimmt die Source-Datei vor. Listing 6 kann man einige Implementierungen entnehmen.

```

1  uint8_t DISPLAY_START_ROW      = 0x40; //Display start row 0
2  uint8_t DISPLAY_ADC           = 0xA1; //ADC reverse
3  uint8_t DISPLAY_COMMON_OUTPUT = 0xC0; //Normal COM0-COM64
4
5  {...}

```

Listing 6: Implementierung: Display-Befehle,
Quellen: [54].

Der erste Befehl setzt die Startzeile des Displays auf Zeile 0.

Der zweite Befehl konfiguriert die RAM-Data-Column-Address des Displays. Dieser Befehl wurde auf *reverse* gesetzt. Dies ermöglicht eine Spiegelung der eingegeben Spalten im Display-RAM, welche aufgrund der Montageposition des Display auf der Platine erforderlich ist. Die Spalten beginnen so am linken Rand des Displays, wie man es erwarten würde.

Der ST7565R-Controller enthält intern 65 Common-Output-Circuits und 132 Segment-Output-Circuits, welche es ihm ermöglichen, als einzelner Chip, ein 65x132 großes Dot-Display anzusteuern. Der dritte Befehl konfiguriert die Nutzung der Common-Output-Circuits auf COM0-COM64, sodass alle verwendet werden.

```

1  void Display_Init(void)
2  {
3      {...}
4
5      //Activate display backlight
6      HAL_GPIO_WritePin(DISPLAY_BL_GPIO_Port, DISPLAY_BL_Pin, GPIO_PIN_SET);
7
8      //Set chipselect
9      HAL_GPIO_WritePin(DISPLAY_CS_GPIO_Port, DISPLAY_CS_Pin, GPIO_PIN_SET);
10
11     //Software reset
12     Display_SendCommand(&DISPLAY_RESET);
13
14     //Initialization
15     Display_SendCommand(&DISPLAY_START_ROW);
16     Display_SendCommand(&DISPLAY_ADC);
17     Display_SendCommand(&DISPLAY_COMMON_OUTPUT);
18     {...}
19 }

```

Listing 7: Funktionsimplementierung: Display_Init() - Teil 2,
Quellen: [54].

Listing 7 kann man den zweiten Teil der Implementierung der Initialisierung entnehmen.

Begonnen wird mit dem Anschalten des Display-Backlights. Danach wird der Chip-Select-Pin auf high gesetzt. Der Display-Controller erwartet, dass der CS-Pin high idlet. Nach einem Software-Reset werden die ersten Konfigurations-Befehle an das Display gesandt.

Abbildung 19 kann man ein Verlaufsdiagramm, welches den Abschluss der Hardware-Initialisierung zeigt, entnehmen. Nach dem Vornehmen der gewünschten Einstellungen wird die Initialisierung durch das Setzen des RAMs abgeschlossen. Dies verhindert, dass zufällige Pixels eingefärbt sind. Danach wird das Display eingeschaltet.

Listing 8 kann man die Implementierung des letzten Teils der Initialisierung entnehmen.

```

1 void Display_Init(void)
2 {
3     {...}
4
5     // Initialization
6     {...}
7     Display_SendCommand(&DISPLAY_ORIENTATION);
8     Display_SendCommand(&DISPLAY_BIAS);
9     Display_SendCommand(&DISPLAY_POWER_CONTROL);
10    Display_SendCommand(&DISPLAY_BOOST_RATIO_0);
11    Display_SendCommand(&DISPLAY_BOOST_RATIO_1);
12    Display_SendCommand(&DISPLAY_DC_REGULATOR);
13    Display_SendCommand(&DISPLAY_VOLUME_MODE_0);
14    Display_SendCommand(&DISPLAY_VOLUME_MODE_1);
15    Display_SendCommand(&DISPLAY_INDICATOR_0);
16    Display_SendCommand(&DISPLAY_INDICATOR_1);
17
18    // Set visible part of DDRAM to avoid that random pixels are set
19    Display_ClearScreen();
20
21    Display_SendCommand(&DISPLAY_ON);
22 }
```

Listing 8: Funktionsimplementierung: Display_Init() - Teil 3,
Quellen: [54].

In der Initialisierung wurde die Funktion *Display_SendCommand()* verwendet, ohne dass diese näher erläutert wurde. Dies wird im nächsten Kapitel nachgeholt.

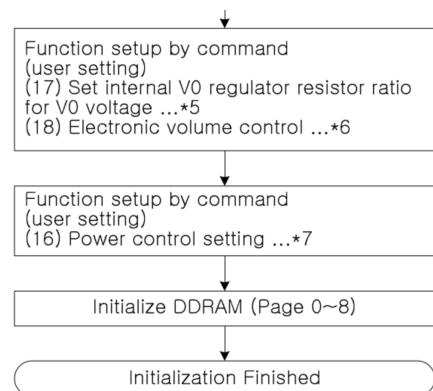


Abbildung 19: Abschluss der
Hardware-Initialisierung: ST7565R,
Quelle: [53].

4.4.3 Senden von Kommandos und Daten

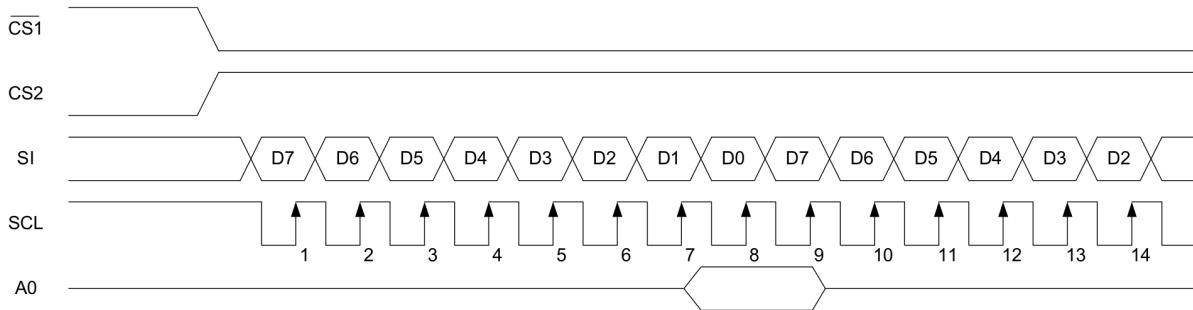


Abbildung 20: Timing-Diagramm der SPI-Verbindung: ST7565R,
Quelle: [53].

Ein Timing-Diagramm welches den Verlauf der SPI-Kommunikation zwischen Mikrocontroller und Display zeigt, kann man Abbildung 20 entnehmen. In der vorliegenden SPI-Konfiguration sind nur die Verbindungen /CS1, SI und A0 relevant. CS2 wird nicht verwendet und um das exakte Timing des SPI-Busses (SCL), bspw. wie lang ein Taktzyklus sein soll, „kümmert“ sich der SPI-Treiber der HAL-Library.

Im Timing-Diagramm kann man erkennen, dass zum Start einer Transmission, die /CS1-Line low gezogen werden muss. Danach folgen 8- oder 16-Bit an Daten. Alle 8-Bit überprüft der LCD-Controller die A0-Line. Ist diese low, wird das vorhergegangene Byte als Kommando interpretiert. Ist sie high, wird es als Datenbyte interpretiert.

```

1 void Display_SendData(uint8_t* data)
2 {
3     // Pull A0 high & chipselect low
4     HAL_GPIO_WritePin(DISPLAY_A0_GPIO_Port, DISPLAY_A0_Pin, GPIO_PIN_SET);
5     HAL_GPIO_WritePin(DISPLAY_CS_GPIO_Port, DISPLAY_CS_Pin, GPIO_PIN_RESET);
6
7     //Send command
8     HAL_StatusTypeDef status = HAL_SPI_Transmit(&hspi3, data, 1, 100);
9
10    // Pull A0 low & chipselect high again
11    HAL_GPIO_WritePin(DISPLAY_A0_GPIO_Port, DISPLAY_A0_Pin, GPIO_PIN_RESET);
12    HAL_GPIO_WritePin(DISPLAY_CS_GPIO_Port, DISPLAY_CS_Pin, GPIO_PIN_SET);
13
14    #ifdef DEBUG_BUILD
15        Display_CheckReturn(&status);
16        UART_printf(&huart1, "data: %#02x\r\n", *data);
17    #endif
18 }
```

Listing 9: Funktionsimplementierung: Display_SendData(),
Quellen: [54].

Listing 9 kann man die Implementierung der *Display_SendData()*-Funktion entnehmen. Die Implementierung der *Display_SendCommand()*-Funktion ist analog dazu. Die A0-Line wird lediglich low statt high gezogen.

4.4.4 Font-Library

Sonderlich komfortabel ist die Nutzung der *Display_SendData()*-Funktion nun allerdings noch nicht. Man würde es bevorzugen “gewöhnliche” Strings an das Display schicken zu können.

Wie genau kommt man aber von einem ASCII-Character in einem String zur *Display_SendData()*-Funktion im Display-Treiber, welche Display-spezifische Bytecodes erwartet? Dies ist die Aufgabe der Font-Library. Bevor man diese allerdings aufrufen kann, ist ein wenig Vorarbeit notwendig.

Diese Vorarbeit erledigen die Funktionen *Display_SendString()*, *Display_SendStringRaw()* und *Display_SendCharacter()*. Listing 10-12 kann man den Quellcode der drei Funktionen entnehmen.

```

1 void Display_SendString(uint8_t row, uint8_t col, const char* string)
2 {
3     //Sanity check
4     if(strlen(string) < 22)
5     {
6         Display_SetColumn(col);
7         Display_SetRow(row);
8         Display_SendStringRaw(string);
9     }
10 }
```

Listing 10: Funktionsimplementierung: *Display_SendString()*,
Quellen: [54].

```

1 void Display_SendStringRaw(const char* string)
2 {
3     //For every character in the string
4     for(uint8_t i = 0; i < strlen(string); i++)
5     {
6         Display_SendCharacter(string[i]); //Send character
7     }
8 }
```

Listing 11: Funktionsimplementierung: *Display_SendStringRaw()*,
Quellen: [54].

```

1 void Display_SendCharacter(uint8_t character)
2 {
3     //Look-Up
4     Display_SendData(&ui8CharMap[character][0]);
5     Display_SendData(&ui8CharMap[character][1]);
6     Display_SendData(&ui8CharMap[character][2]);
7     Display_SendData(&ui8CharMap[character][3]);
8     Display_SendData(&ui8CharMap[character][4]);
9     Display_SendData(&ui8CharMap[character][5]);
10 }
```

Listing 12: Funktionsimplementierung: *Display_SendCharacter()*,
Quellen: [54].

Display_SendString() ist die hauptsächliche API-Funktion, die man nutzen wird. Dieser werden die gewünschte Zeile, Spalte und der darzustellende String übergeben. Nach einem Check, ob der String auf dem Display darstellbar ist, Scrollen wird derzeit nicht unterstützt, und der Einstellung von Zeile und Spalte, wird die nachfolgende Funktion *Display_SendStringRaw()* aufgerufen.

Diese Funktion iteriert über alle Character im String und ruft für jeden die *Display_SendCharacter()*-Funktion auf.

Die Funktion `Display_SendCharacter()` führt einen Look-Up in der Character-Map der Font-Library durch. Der Index dieses Look-Up-Arrays ist der ASCII-Code des jeweiligen Zeichens. Der Wert an der gewählten Stelle ist der Bytecode, welcher die entsprechenden Pixel setzt. Jeder Character/jedes Zeichen besteht aus 6x8 Pixeln. Das Display erwartet, dass pro Character 6 Byte gesendet werden.

Abbildung 21 kann man das Pixel-Mapping beispielhaft für den Buchstaben "X" entnehmen.

D0 - D3 beschreiben das erste Nibble, wobei D3 das MSB ist.

D4 - D7 beschreiben das zweite Nibble, wobei D7 das MSB ist.

Das Display erwartet, dass das zweite Nibble immer als erstes gesendet wird, mit dem MSB first.

Dies führt beim Senden zu der Reihenfolge D7 - D4 und dann D3 - D0, je Byte.

The diagram illustrates the pixel mapping for the character 'X'. It consists of two parts: a 6x8 grid of pixels and a corresponding byte sequence table.

Pixel Grid: A 6x8 grid where black squares represent pixels set to 1 and white squares represent pixels set to 0. The grid shows the pattern of the character 'X'.

Byte Sequence Table:

D7	D6	D5	D4	D3	D2	D1	D0
00	01	02	03	04	05	06	07
83	82	81	80	7F	7E	7D	7C
S0	S1	S2	S3	S4	S5	S6	S8

Abbildung 21: Pixel-Mapping: ST7565R,
Quelle: [53].

Man beachte, dass man Zeile D7 und Spalte 00 bei gewöhnlichen Glyphen frei lässt, dies bietet Platz zu benachbarten Glyphen und verbessert die Lesbarkeit.
Irritierenderweise lässt das Datenblatt des LCD-Controllers bei Glyphen Spalte 05, die korrespondiere Font-Library aber Spalte 00, frei. Visuell spielt dies keine Rolle, man sollte nur einheitlich bleiben.

Der Bytecode für die erste Spalte des Buchstabens "X" wäre demnach:

D0 + D1 = **0x3**,

D5 + D6 = **0x6**.

Da man als erstes Byte, zum Freilassen von Spalte 00, stets 0x00 senden würde, wäre das zweite Byte dementsprechend **0x63**.

Ein Blick in die Character-Map der Font-Library bestätigt dies. Listing 13 kann man einige Character-Mappings entnehmen.

```

1 {0x00, 0x07, 0x18, 0x60, 0x18, 0x07}, // 86 V
2 {0x00, 0x7F, 0x20, 0x18, 0x20, 0x7F}, // 87 W
3 {0x00, 0x63, 0x14, 0x08, 0x14, 0x63}, // 88 X
4 {0x00, 0x03, 0x04, 0x78, 0x04, 0x03}, // 89 Y
5 {0x00, 0x61, 0x51, 0x49, 0x45, 0x43}, // 90 Z

```

Listing 13: Implementierung: Character-Look-Up-Table,
Quellen: [55].

Für die ASCII-Character 0 - 127 ist die Font-Library ASCII-konform und stellt die entsprechenden Bytecodes zur Verfügung. Ab Index 128 stehen einem eigene Character zur Verfügung. So ist dort beispiels-

weise unser Firmenlogo hinterlegt. Ich habe dort auch diverse Strukturen zum Bauen von Oberflächen, wie Pfeile, Balken und Linien, erstellt.

Dies schließt den Blick in die Implementierung des Display-Treibers ab. Im nächsten Kapitel wird die Implementierung der Interrupt-Service-Routinen erläutert.

4.5 Interrupts

Die Implementierung der Interrupts ist simpel. Man legt eine Priorität für den Interrupt fest und aktiviert diesen (siehe Listing 14). Der Code wurde anhand der Konfiguration, welche in Kapitel 4.1 vorgestellt wurde, automatisch von *CubeMX* erzeugt.

```

1 HAL_NVIC_SetPriority(EXTI1_IRQHandler, 5, 0);
2 HAL_NVIC_EnableIRQ(EXTI1_IRQHandler);
3
4 HAL_NVIC_SetPriority(EXTI2_IRQHandler, 5, 0);
5 HAL_NVIC_EnableIRQ(EXTI2_IRQHandler);
6
7 HAL_NVIC_SetPriority(EXTI3_IRQHandler, 5, 0);
8 HAL_NVIC_EnableIRQ(EXTI3_IRQHandler);
9
10 HAL_NVIC_SetPriority(EXTI4_IRQHandler, 5, 0);
11 HAL_NVIC_EnableIRQ(EXTI4_IRQHandler);

```

Listing 14: Implementierung: Interrupts,
Quellen: [56].

Die Implementierung der Interrupt-Handler ist ebenso “straightforward“. Da in der vorliegenden Konfiguration vier separate Interrupt-Leitungen verwendet werden, muss in der ISR auch keine weitergehende Auswertung, bspw. welcher Interrupt denn nun ausgelöst hat, stattfinden. Es wird lediglich das Pending-Flag des entsprechenden Interrupts zurückgesetzt und ein State, welcher den Button-Press eindeutig identifiziert, gesetzt. Details zu diesem State folgen im Kapitel über den Button-Task. Auch weiterführende Unterscheidungs-Logik ist erstmal nicht von Bedeutung, da jeder Interrupt dieselbe Priorität hat.

Listing 15 kann man die Implementierung der ersten beiden Interrupt-Handler entnehmen. Die Implementierung der anderen beiden ist analog dazu (anderer State und anderes Pending-Flag).

```

1 void EXTI1_IRQHandler(void) //ON-Button
2 {
3     //Clear pending flag
4     EXTI->PR1 |= EXTI_PR1_PIF1;
5
6     pressedButton = ON_BUTTON;
7 }
8
9 void EXTI2_IRQHandler(void) //OK-Button
10 {
11     //Clear pending flag
12     EXTI->PR1 |= EXTI_PR1_PIF2;
13
14     pressedButton = OK_BUTTON;
15 }
16
17 { ... }

```

Listing 15: Implementierung: Interrupt-Handler,
Quellen: [57].

4.6 Applikation

In diesem Kapitel wird die Implementierung der Applikation inkl. aller Tasks vorgestellt. Wie in vorhergegangen Kapiteln bereits erwähnt, macht der hier geschriebene Code den größten Teil in diesem Projekt aus.

Die Implementierung, aller nachfolgend gezeigten Code-Snippets, findet im Modul *main.c* statt. In einer *main.h*-Datei werden Definitionen, Typedefs und statische Variablen abgelegt. Dies dient lediglich optischen Separierungs-Gründen. *main.h* ist **keine** Schnittstelle und sollte **niemals** an einer anderen Stelle als *main.c* inkludiert werden.

4.6.1 Initialisierung

Listing 16 kann die Implementierung der *main*-Funktion entnommen werden.

```

1  int main(void)
2  {
3      /* Initialization of all peripherals, the flash interface and the systick */
4      {...}
5
6      /* Wait for stability. Workaround for now */
7      HAL_Delay(10000);
8
9      /* Create the threads */
10     osThreadDef(keepAliveTaskName, KeepAliveTask, osPriorityNormal, 1, 128);
11     defaultTaskHandle = osThreadCreate(osThread(keepAliveTaskName), NULL);
12
13     {...}
14
15     /* Create the queue */
16     osMessageQDef(msgQueueName, 1, uint32_t);
17     msgQueueID = osMessageCreate(osMessageQ(msgQueueName), NULL);
18
19     /* Create the memory pool */
20     osPoolDef(memPoolName, 1, uint32_t);
21     memPoolID = osPoolCreate(osPool(memPoolName));
22
23     /* Initialize display */
24     Display_Init();
25
26     /* Start scheduler */
27     osKernelStart();
28
29     /* We should never get here as control is now taken by the scheduler */
30     while (1)
31     {
32
33     }
34 }
```

Listing 16: Implementierung: main() - Funktion,
Quellen: [57].

Begonnen wird mit Reset und Initialisierung der Peripherie. Dies erledigt Code, welcher von *CubeMX* generiert wurde. Anschließend wird 10 Sekunden auf Hardware-Stabilität gewartet. Der Wert ist die Empfehlung eines anderen Entwicklers, welcher die Hardware, speziell die Batterieversorgung, gut kennt.

Weiter geht es mit der Erstellung der Tasks. Einem Task wird ein eindeutiger Name, eine Callback-Funktion, eine Priorität, die maximale Anzahl an Instanzen und eine Stacksize mitgegeben. Ich habe mich bei allen Tasks für dieselbe Priorität (**normal**), eine maximal mögliche Instanz von **1** und die Default-Stacksize von 128 Byte entschieden.

Danach wird die Message-Queue erstellt. Diese bildet den Kommunikationskanal zwischen Button- und Display-Task. Sie bietet Platz für eine Message. Da Button-Presses intern im Button-Task sowieso nur alle 300 ms verarbeitet werden, genügt dies.

Anschließend wird der Memory-Pool erstellt. Der Memory-Pool stellt den Speicherbereich, der für die Messages innerhalb der Queue gebraucht wird, zur Verfügung. Die Größe wird auf 32-Bit gesetzt, was für die Übergabe eines Pointers innerhalb der Message genügt. Dies ermöglicht es mir später auch komplexere Strukturen über die Message-Queue zu verschicken, ohne die Größe dieser anpassen zu müssen. In diesem Fall ist, bei meiner Architektur, der Sender für die Allokation der Nachricht und der Empfänger für die Freigabe verantwortlich.

Die Handles, welche die Tasks, die Queue und den Memory-Pool eindeutig identifizieren, sind statische Variablen, welche in der *main.h* definiert wurden.

Nach einem Aufruf der *Display_Init()*-Funktion, welche bereits in Kapitel 4.4.2 vorgestellt wurde, wird der Scheduler gestartet. Die Kontrolle des Programms wurde nun an das RTOS abgegeben.

4.6.2 Keep-Alive-Task

Der Keep-Alive-Task dient dem Zweck anzuzeigen, dass die Hardware noch “lebt” und man sich nicht in einem Hardfault oder einer Endlosschleife befindet. Man hat so außerdem während der Entwicklung einen Task zur Verfügung, in den man “schnell mal“ temporäre Debugging-Funktionalitäten auslagern kann, welche nicht aus dem Button- oder Display-Task heraus aufgerufen werden können.

Der Code kann Listing 17 entnommen werden.

```

1 void KeepAliveTask(void const* argument)
2 {
3     for (;;)
4     {
5         HAL_GPIO_TogglePin(LED_YELLOW_GPIO_Port, LED_YELLOW_Pin);
6         HAL_GPIO_TogglePin(LED_RED_GPIO_Port, LED_RED_Pin);
7         HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
8         osDelay(1000);
9     }
10 }
```

Listing 17: Implementierung: Keep-Alive-Task,
Quellen: [57].

Der Task togglet die drei vorhandenen On-Boards-LEDs jede Sekunde.

4.6.3 Button-Task

Der Button-Task ist für die Auswertung der Button-Presses und eine erste Low-Level-Vorverarbeitung verantwortlich. Wie im Kapitel über die Interrupt-Implementierung bereits erwähnt, existiert ein State, welcher den Button-Press eindeutig identifiziert.

Listing 18 kann der verantwortliche Code aus der *main.h* entnommen werden.

```

1  { ... }
2
3  /* Typedefs ----- */
4  typedef enum
5  {
6      DEFAULT = 0,
7      ON_BUTTON,
8      OK_BUTTON,
9      UP_BUTTON,
10     DOWN_BUTTON
11 } buttonPress_t;
12
13 { ... }
14
15 //Buttonstate for interrupts
16 static volatile buttonPress_t pressedButton;
17
18 { ... }

```

Listing 18: Implementierung: Button-State-Variable,
Quellen: [57].

Wichtige Entwurfsentscheidungen, die an dieser Stelle erwähnt werden müssen.

Erstens, die *pressedButton*-Variable ist statisch, im ganzen Modul bekannt und wird von allen Interrupts gleichermaßen schreibend verwendet. Wie verhält sich dies bei kaskadierenden Interrupts?

Dadurch, dass alle Interrupts dieselbe Priorität haben, können sich diese nicht gegenseitig unterbrechen. Sie werden im *Nested-Vectored-Interrupt-Controller* gequeued und nacheinander abgearbeitet.

Was zum zweiten Punkt führt.

Sollte man es schaffen, einen zweiten Button-Press auszulösen, bevor der vorhergehende ausgewertet wurde, wird der neue Button-Press als aktueller gesetzt und der vorhergehende verworfen. Für einen menschlichen Nutzer, bei der vorhandenen Hardware, dürfte dies allerdings äußerst schwierig werden.

Es handelt sich dabei um die bewusste Design-Entscheidung, immer nur eine Nachricht zu verarbeiten. Dies wurde so entschieden, weil davon ausgegangen wird, dass derart schnelle Zustandsänderungen der Buttons unbeabsichtigt sind. Diese Entscheidung wurde, konsistent mit der Queue-Length von 1 und dem internen Delay im Button-Task von 300 ms, getroffen.

Das Design wird später u.U. Anpassungen erfordern. Dies wird ein Praxistest zeigen.

Nach einer Initialisierung der im Button-Task lokal benötigten Variablen, geht es in die State-Machine des Button-Tasks.

Listing 19 kann die Implementierung entnommen werden.

```

1 // Init
2 static uint32_t selectedRow = DEFAULT_START_ROW;
3 static uint32_t* memPointer = NULL;
4
5 for (;;)
6 {
7     // Print current arrow
8     PrintArrow(selectedRow);
9
10    switch(pressedButton)
11    {
12        case ON_BUTTON:
13            { ... }
14
15        case OK_BUTTON:
16            { ... }
17
18        case UP_BUTTON:
19            { ... }
20
21        case DOWN_BUTTON:
22            { ... }
23    }
24
25    osDelay(300);
26 }
```

Listing 19: Implementierung: Button-State-Machine,
Quellen: [57].

Der *ON_BUTTON*-Case toggled derzeit nur das Backlight des Displays. Hier wäre in Zukunft Platz für weitergehende Energie- oder Shutdown-Modi. Mehr dazu in Kapitel 5.2 und 5.3.

Der *UP_BUTTON*- und *DOWN_BUTTON*-Case verhalten sich ähnlich, weswegen in der Folge, nur einer der beiden betrachtet wird. Listing 20 kann der Quellcode des *UP_BUTTON*-Cases entnommen werden.

```

1 case UP_BUTTON:
2
3     //Check boundaries
4     if(selectedRow > 2)
5     {
6         //Delete old arrow
7         DeleteArrow(selectedRow);
8
9         //Move one row up
10        selectedRow--;
11
12        //Print new arrow
13        PrintArrow(selectedRow);
14    }
15
16    pressedButton = DEFAULT;
17    break;
```

Listing 20: Implementierung: UP_BUTTON-Case,
Quellen: [57].

Das Display besitzt 8 Zeilen, welche intern mit 0 (oberste Zeile) und 7 (unterste Zeile) referenziert werden. Wie man Listing 19 entnehmen kann, wird die Start-Zeile mit *DEFAULT_START_ROW* initialisiert,

welche auf **3** eingestellt ist. Die Zeilen 0 - 2 sind für das Firmenlogo, eine Menü-Überschrift und eine Leerzeile reserviert. Content kann ab Zeile 3 und bis einschließlich Zeile 7 dargestellt werden.

Der **UP_BUTTON**-Case prüft nun als erstes, ob man sich mindestens in Zeile 3 befindet. Wenn ja, wird der “Arrow“ in der alten Zeile gelöscht, es wird eine Zeile hochgegangen (dekrementieren des Counters) und der “Arrow“ wird in der neuen Zeile gerendert.

Der “Arrow“ ist ein blinkender Zeilenindikator, welcher am rechten Bildschirmrand dargestellt wird. Seine Implementierung erfolgt, ähnlich zum Bau der Menü-Strukturen, im unteren Abschnitt des *main*-Moduls.

In Listing 19, Zeile 8 kann man erkennen, dass der Zeilenindikator, in der aktuell ausgewählten Zeile gerendert wird und dies jeden “Frame“ passiert, nicht nur initial oder bei einem Button-Press. Dies liegt daran, dass der Display-Task mehrere Kontext-Switches hat, in denen er den gesamten Bildschirminhalt löscht. Dabei wird keine Rücksicht auf den Zeilenindikator genommen. Zur Sicherheit stellt der Button-Task diesen deswegen alle 300 ms wieder her.

Der **OK_BUTTON**-Case ist für die Auswahl einer Zeile zuständig. Listing 21 kann der Quellcode dieses Cases entnommen werden.

```

1 case OK_BUTTON:
2
3     // Allocate space for the message
4     memPointer = osPoolAlloc(memPoolID);
5
6     // Set value of pointer to the selected row
7     *memPointer = selectedRow;
8
9     // Send OS-Message with the selected row
10    osMessagePut(msgQueueID, (uint32_t)memPointer, osWaitForever);
11
12    // To debounce
13    osDelay(100);
14
15    pressedButton = DEFAULT;
16    break;

```

Listing 21: Implementierung: OK_BUTTON-Case,
Quellen: [57].

Nach der Allokation eines 32-Bit Speicherblockes, wird der Inhalt des Blockes auf die aktuell ausgewählte Zeile gesetzt. Anschließend wird eine OS-Message mit einem Pointer auf den Speicherbereich verschickt. Der Parameter *osWaitForever* sorgt dafür, dass die Funktion solange wartet, bis die Nachricht zugestellt, also in der Queue platziert werden konnte. Anschließend wird 100 ms gewartet, um ein Überlaufen der Empfängermailbox bei schnell aufeinanderfolgenden Events zu verhindern. Auch hier würde noch Ausbaupotenzial, wie eine größere Button-Debouncing-Lösung, bestehen. Kapitel 5.2 wird dies erneut aufgreifen.

Dies schließt die Implementierung des Button-Tasks ab.

4.6.4 Display-Task

Der Display-Task ist für die Auswertung der Menü-Auswahl und das Darstellen der Menüs verantwortlich. Diese Darstellung erfolgt über das Aufrufen von Prozeduren, welche das Design der konkreten Menüs implementieren und intern den Display-Treiber aufrufen.

Der Display-Task würde der Architektur nach (vergleiche Abbildung 12) mit dem Funk-Task (per OS-Messages) kommunizieren und Funktests initiieren. Die Implementierung dieser Kommunikation erfolgt aus zeitlichen Gründen zu einem späteren Zeitpunkt. Mehr Informationen dazu folgen in Kapitel 5.2. Der Display-Task hält auch Status darüber, ob aktuell ein Funktest durchgeführt wird und stellt in dieser Zeit eine Warteanimation dar.

Listing 22 zeigt die Implementierung des Display-Tasks.

```

1  // Init
2  static osEvent    buttonEvent;
3  static menu_t     selectedMenu = MAIN_MENU;
4  static uint32_t*  retPointer   = NULL;
5  static uint32_t   selectedRow  = 0;
6  static bool       testOngoing = false;
7
8  Display_ClearScreen();
9  PrintMainMenu();
10
11 for (;;)
12 {
13     {...}
14
15     //Get OS-Message or wait 500ms for it
16     buttonEvent = osMessageGet(msgQueueID, 500);
17
18     //Message got in
19     if(buttonEvent.status == osEventMessage)
20     {
21         //Set newly selected row
22         retPointer = buttonEvent.value.p;
23         selectedRow = *retPointer;
24
25         //Switch based on menu
26         switch(selectedMenu)
27         {
28             case MAIN_MENU:
29                 {...}
30
31             case TEST_MENU:
32                 {...}
33
34             case CONNECTION_TEST:
35                 {...}
36
37             case ABOUT_MENU:
38                 {...}
39         }
40
41     {...}
42 }
43 }
```

Listing 22: Implementierung: Display-State-Machine,

Quellen: [57].

Nach einer Initialisierung der im Display-Task lokal benötigten Variablen, wird das Display bereinigt und das Hauptmenü dargestellt. Anschließend wird in die Display-State-Machine übergegangen.

Der Display-Task wartet nun 500 ms auf den Eingang einer OS-Message vom Button-Task. Dies hat den Grund, dass das Display, im Falle eines laufenden Tests, weiter aktualisiert werden muss (Darstellen einer Warteanimation). Ansonsten könnte der Display-Task, ähnlich wie der Button-Task, mit *osWaitForever* solange warten, bis eine Nachricht eingegangen ist.

Bei Eingang einer OS-Message wird sich der Pointer auf den übergebenen Speicherbereich, für eine spätere Freigabe und die ausgewählte Zeile gemerkt.

Anschließend wird basierend auf dem Menü, in dem man sich aktuell befindet, in das entsprechende Switch-Case-Statement gewechselt. Dieses wertet die angewählte Zeile, entsprechend des Layouts und der Regeln des Menüs, aus.

Der *MAIN_MENU*-Case bildet den Startpunkt der State-Machine und bietet (derzeit) zwei Menüpunkte zur Auswahl an. Listing 23 kann der Quellcode dieses Cases entnommen werden.

```

1   case MAIN_MENU:
2
3       // Switch based on row
4       switch(selectedRow)
5       {
6           case 3:
7
8               selectedMenu = TEST_MENU;
9               Display_ClearScreen();
10              PrintTestMenu();
11              break;
12
13           case 4:
14
15               selectedMenu = ABOUT_MENU;
16               Display_ClearScreen();
17               PrintAboutMenu();
18               break;
19
20           default:
21               break;
22       }
23
24       break;

```

Listing 23: Implementierung: MAIN_MENU-Case,
Quellen: [57].

Der *MAIN_MENU*-Case bietet zwei Menüpunkte an, welche sich in Zeile 3 und 4 befinden. Bei der Auswahl einer dieser Zeilen wird das ausgewählte Menü entsprechend neu gesetzt. Anschließend wird der Inhalt des Bildschirms bereinigt und das entsprechende Menü dargestellt.

Das *ABOUT_MENU* ist ein für den aktuellen Prototypen speziell angefertigtes Menü, welches Informationen über das Gerät anzeigt und langfristig nicht vorhanden sein wird. Auf eine genauere Betrachtung wird deshalb verzichtet.

Das *TEST_MENU* ist das Menü, welches eine allgemeine Auswahl der zur Verfügung stehenden Tests anzeigt. Der Quellcode dieses Menüs kann Listing 24 entnommen werden.

```

1   case TEST_MENU:
2
3     // Switch based on row
4     switch(selectedRow)
5     {
6       case 3:
7
8         selectedMenu = CONNECTION_TEST;
9         Display_ClearScreen();
10        PrintConnectionTest();
11        testOngoing = true;
12        break;
13
14       case 7:
15
16         selectedMenu = MAIN_MENU;
17         Display_ClearScreen();
18         PrintMainMenu();
19         break;
20
21       default:
22         break;
23     }
24
25   break;

```

Listing 24: Implementierung: TEST_MENU-Case,
Quellen: [57].

Identisch zu den anderen Cases in der State-Machine führt auch der *TEST_MENU*-Case eine Auswertung auf Basis der ausgewählten Zeile durch. Die einzige Auswahl-Option bietet der *TEST_MENU*-Case derzeit in Zeile 3. Weitere Tests wie bspw. Analysen der Funkqualität, der Fehlerraten oder der RSSI-Werte (*Received Signal Strength Indicator*) sind in Planung und würden hier zur Auswahl angeboten werden. Details dazu folgen in Kapitel 5.3.

Bei der Auswahl von Zeile 3 wird, nach dem bekannten Vorgehen, der aktuelle Kontext auf *CONNECTION_TEST* und die Variable *testOnGoing* auf *true* gesetzt. Dies hat derzeit nur den Effekt, dass auf dem Display eine Warte-Animation dargestellt wird. In Zukunft wird an dieser Stelle zusätzlich eine *Discovery-Message* über das RTOS an den Funk-Task verschickt, welche eine Verbindung mit den Geräten in der Nähe initiiert.

In Zeile 7 befindet sich eine *return*-Option, welche zurück ins *MAIN_MENU* führt. Hier besteht u.U. noch Potenzial zum Ausbau. Wenn sich bspw. herausstellt, dass man stets dieselben Parameter setzt um einen Menü-Wechsel durchzuführen, bietet sich eine Auslagerung in Funktionen an. Die wenigen expliziten Aufrufe bieten sich für diese Arbeit aber von der Darstellung her gut an.

Das *CONNECTION_TEST*-Menü besitzt analog zur Beschreibung oben derzeit keine Funktionalität und hat deswegen nur eine *return*-Option. Auf eine genauere Betrachtung wird deshalb verzichtet.

Dies schließt die Implementierung des Display-Tasks ab. Dieser war die letzte zu implementierende Komponente dieses Projektes.

Die Entwicklungsarbeit, im Rahmen dieser Bachelorarbeit, ist somit abgeschlossen.

5 Fazit und Ausblick

5.1 Zusammenfassung

Die Motivation für diese Arbeit war die Neu-Entwicklung eines Produktes. Ein Handheld-Device, welches in Zukunft Funktests durchführen können soll. Die Aufgabe war es, eine Embedded-Software Architektur für dieses Gerät zu entwerfen und prototypisch zu implementieren.

Zuerst wurde die Situation am Weltmarkt, inklusive der Auswirkungen auf diese Arbeit, erläutert und der verwendete Prototyp vorgestellt.

Anschließend wurde der Stand der Technik präsentiert. In diesem Kapitel wurde die Dipl.-Ing. H. Horstmann GmbH und eines ihrer bekanntesten Produkte, der Kurzschlussanzeiger "Smart Navigator 2.0", vorgestellt. Anschließend folgte eine Erklärung verschiedener Begrifflichkeiten aus der Welt der Mikroprozessoren und Mikrocontroller. Der verwendete Mikrocontroller, ein STM32L471, wurde im Anschluss, mit einigen bekannten und hier im Unternehmen verwendeten Controllern, in zwei Benchmarks verglichen. Dabei fielen die gute Performance pro MHz und die hohe Energieeffizienz des Chips auf. Danach wurde die Vorgehensweise bei der Programmierung von STM32-Controllern bzw. allgemein von ARM-Cortex-Prozessoren erklärt. Die Programmierhierarchien HAL und CMSIS wurden an einem Beispiel präsentiert und erläutert.

Danach folgte die Konzeption des Projektes und der geplanten Software-Architektur. Angefangen wurde mit der Vorstellung der Prototypen-Hardware. Die verbauten Komponenten wurden präsentiert und es wurden alle, auf die Software übergreifenden, Design-Entscheidungen erläutert. Darunter fielen bspw. separate SPI- und Interrupt-Lines, sowie die Entscheidung ein RTOS zu verwenden.

Anschließend wurden die verwendeten Software-Komponenten und die hier im Unternehmen verwendete Programmierrichtlinie vorgestellt. Danach folgte die Konzeption der Tasks und des Display-Treibers. Geplante APIs und Statemachines wurden präsentiert und erläutert. Abgeschlossen wurde das Kapitel mit einer Betrachtung des verwendeten Software- und Debugging-Stacks.

Anschließend folgte das Kapitel der Implementierung. Begonnen wurde mit dem Aufsetzen der Projektstruktur. Parameter wie Bus- und Taktraten wurden erläutert und entsprechend eingestellt. Anschließend folgte ein erster Funktionstest der Hardware in Kombination mit der Software, bei dem ein Fehler auffiel. Die SPI-Kommunikation funktionierte nicht ordnungsgemäß und ein erstes Debugging war erforderlich. Danach wurde mit der Entwicklung der Software begonnen. Angefangen beim Schreiben einer "UART_printf()"-Funktion wurde mit der Implementierung des Display-Treibers weitergemacht. Anhand des Datenblattes wurden die Eigenschaften des Display-Controllers vorgestellt und in Software entsprechend umgesetzt. Danach folgte die Implementierung der Interrupt-Service-Routinen. Abgeschlossen wurde das Kapitel mit der Implementierung der Applikation. Handles und Memory-Buffer wurden erstellt und Keep-Alive-, Button- und Display-Task, der vorgestellten Architektur nach, implementiert.

Das Ziel war ein bedienfähiger Prototyp, welcher die grundlegende Architektur zur Integration des firmeninternen Funk-Treibers besitzt, um in Zukunft Funktests durchzuführen. Das Ziel wurde erreicht.



Abbildung 22: Aktuelles Bild des Prototypen,
Quelle: [58].

Abbildung 22 kann ein aktuelles Bild des Prototypen entnommen werden.

5.2 Ergänzungen

In diesem Kapitel werden Ergänzungen beschrieben, welche, für eine volle Funktionsfähigkeit des Prototypen, noch vorgenommen werden müssen.

Dies umfasst zum Ersten das Thema Funk. Die Architektur des Projektes wurde für eine Kommunikation mit einem Funk-Task konzipiert und die Software entsprechend implementiert. Der Funk-Treiber wurde aus einem alten Projekt extrahiert und rudimentär integriert. Vorgenommen werden muss eine vollständige Integration des Funk-Tasks in die Applikation. Die entsprechenden RTOS-Einstellungen (Queues, Messages, Buffer) müssen entsprechend konfiguriert werden. Anschließend muss ein erster Test, welcher Geräte im Labor anfunkt, geschrieben werden.

Als nächstes müsste man sich einer “ordentlichen“ Entprellung der Buttons widmen. Diese kann wahlweise in Hard- oder Software erfolgen.

Die Entprellung wird, durch das Vorhandensein der beiden Platinen, aber wahrscheinlich in Software erfolgen. Eine Abstimmung mit dem Hardware-Entwickler, welcher die Platinen- und Gehäuse-Planung durchführt, ist hier zwangsläufig erforderlich.

Das Gerät läuft derzeit permanent auf Höchstgeschwindigkeit bei maximaler Display-Helligkeit und ohne Option diese zu regulieren oder das Display komplett abzuschalten.

Weitere Ergänzungen würden eine On/Off-Button-Funktionalität mit einem Wechsel in entsprechende Low-Power-Modi und einige Bug-Fixes beinhalten. So ist während der Entwicklung aufgefallen und dies konnte auch mit einem Oszilloskop verifiziert werden, dass die Up- und Down-Buttons, im Vergleich zu anderen Buttons, sehr lange brauchen um wieder VCC-Level zu erreichen. Dies müsste genauer untersucht werden.

Neben einer On/Off-Button-Funktionalität mit entsprechendem Low-Power-Modi sind auch weiterführende Tests der Energie-Effizienz erforderlich. Nach bisherigem Erkenntnisstand hält der Akku (4000 mAh) das Gerät circa 2-3h aktiv am Laufen. Dies ist für einen vollwertigen Einsatz als Handheld-Device noch zu wenig.

5.3 Ausbau und Erweiterung

In diesem Kapitel werden Ausbau- und Erweiterungsmöglichkeiten beschrieben, die einem in diesem Projekt geboten werden und welche für ein zukünftiges Serienprodukt interessant sein könnten.

Allgemein gesprochen könnte man durch die lose Kopplung der Komponenten/Tasks und die geringe Projektgröße überall recht schnell ansetzen. Bspw. sind auf der unteren Platine ein ungenutzter Speicherkartenslot und etliche ungenutzte EEPROMs vorhanden. Diese könnten für Langzeitanalysen und die persistente Speicherung von Funkqualitäts-Messungen verwendet werden. Auch ein Auslesen der Daten, bspw. über die integrierte UART-Schnittstelle, zur Weiterverarbeitung auf PC-Seite wäre denkbar.

Ebenfalls wäre ein Ausbau des Display-Treibers, um bspw. dynamische Kontrasteinstellungen zur Laufzeit vorzunehmen, möglich.

Auch eine Erweiterung des Funk-Treibers wäre denkbar. Dieser könnte beispielsweise die Funktionalität bekommen, ein “Sniffing“ auf der verschlüsselten Verbindung, welche zwischen den Geräten untereinander besteht, durchzuführen, um so noch detailliertere Informationen zur Kommunikation während der Laufzeit zu erhalten.

Abbildungen

- [1] *Bild des Prototypen.* Eigene Darstellung.
- [2] *Smart Navigator 2.0.* Horstmann GmbH. URL: <https://www.horstmanngmbh.com/produkte/kurz-und-erdschlussanzeiger/fuer-freileitungen/monitoring-zur-netzanalyse/smart-navigator-20> (besucht am 25.10.2021).
- [4] *STM32L471VG Product overview.* STMicroelectronics. 2021. URL: <https://www.st.com/en/microcontrollers-microprocessors/stm32l471vg.html> (besucht am 21.10.2021).
- [7] *Nucleo-64 Board.* STMicroelectronics. 2020. URL: https://www.st.com/resource/en/user_manual/dm00105823-stm32-nucleo-64-boards-mb1136-stmicroelectronics.pdf (besucht am 21.10.2021).
- [21] *Layout des Prototypen.* Eigene Darstellung.
- [25] *SPI Daisy-Chain Konfiguration.* Eigene Darstellung.
- [29] *Software-Komponenten im Projekt.* Eigene Darstellung.
- [34] *Datenflussdiagramm der Anwendung.* Eigene Darstellung.
- [36] *Schnittstellen-Entwurf des Display-Treibers.* Eigene Darstellung.
- [37] *Event Driven Program.* Eigene Darstellung.
- [40] *Interner Aufbau des Button-Tasks.* Eigene Darstellung.
- [41] *Interner Aufbau des Display-Tasks.* Eigene Darstellung.
- [42] *Interner Aufbau des Funk-Tasks.* Eigene Darstellung.
- [45] *Übersicht des Debugging-Stacks.* Eigene Darstellung.
- [48] *Konfiguration der Peripherie in CubeMX.* Eigene Darstellung.
- [49] *Konfiguration der Taktraten in CubeMX.* Eigene Darstellung.
- [53] *ST7565R (Display controller) Datasheet.* Sitronix. 2006. URL: <https://www.lcd-module.de/eng/pdf/zubehoer/st7565r.pdf> (besucht am 21.10.2021).
- [58] *Aktuelles Bild des Prototypen.* Eigene Darstellung.

Literatur

- [3] *Smart Navigator 2.0.* Horstmann GmbH. URL: <https://www.horstmanngmbh.com/produkte/kurz-und-erdschlussanzeiger/fuer-freileitungen/monitoring-zur-netzanalyse/smart-navigator-20> (besucht am 25.10.2021).
- [5] Frédéric Gaillard und Andreas Eieland. „Microprocessor (MPU) or Microcontroller (MCU)? What factors should you consider when selecting the right processing device for your next design“. In: (2013). URL: https://ww1.microchip.com/downloads/en/DeviceDoc/MCU_vs_MPUM_Article.pdf (besucht am 21.10.2021).
- [6] Ralf Jesse. *STM32: ARM-Mikrocontroller programmieren für Embedded Systems. Das umfassende Praxisbuch.* mitp Verlags GmbH & Co. KG, Frechen, 2021.
- [8] *CoreMark™ Scores.* Embedded Microprocessor Benchmark Consortium. 2009-2021. URL: <https://www.eembc.org/coremark/scores.php> (besucht am 26.10.2021).
- [9] *STM32L471VG Product overview.* STMicroelectronics. 2021. URL: <https://www.st.com/en/microcontrollers-microprocessors/stm32l471vg.html> (besucht am 26.10.2021).
- [10] *STM32L476RG Product overview.* STMicroelectronics. 2021. URL: <https://www.st.com/en/microcontrollers-microprocessors/stm32l476rg.html> (besucht am 26.10.2021).

- [11] *ESP32-S2 Datasheet*. Espressif Systems. 2021. URL: https://www.espressif.com/sites/default/files/documentation/esp32-s2_datasheet_en.pdf (besucht am 02.11.2021).
- [12] *MSP430 Datasheet*. Texas Instruments. 2020. URL: <https://www.ti.com/product/MSP430F5529> (besucht am 26.10.2021).
- [13] *ATmega644 Datasheet*. Atmel Corporation. 2012. URL: <https://ww1.microchip.com/downloads/en/DeviceDoc/doc2593.pdf> (besucht am 02.11.2021).
- [14] *ATmega2560 Datasheet*. Atmel Corporation. 2014. URL: https://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf (besucht am 02.11.2021).
- [17] *Linker scripts*. Studiendepartment Informatik HAW Hamburg via Cygnus Solutions Technical Publications. 1994-1997. URL: https://users.informatik.haw-hamburg.de/~krabat/FH-Labor/gnupro/5_GNUPro_Utils/c_Using_LD/ldLinker_scripts.html (besucht am 04.11.2021).
- [18] Sebastian Fischmeister. „Introduction to Programming Embedded Systems“. In: (2015). URL: https://www.cis.upenn.edu/~lee/06cse480/lec-intro_to_prog_embedded_systems.pdf (besucht am 03.11.2021).
- [19] *EA DOGM128-6 Datasheet*. Electronic Assembly. 2009.
- [20] John H. Davies. *MSP430 Microcontroller Basics*. Newnes, 2008.
- [22] *MRF89XAM8A Datasheet*. Microchip. 2010.
- [23] Wikipedia contributors. *FTDI — Wikipedia, The Free Encyclopedia*. [Online; Stand 1. Dez 2021]. 2021. URL: <https://en.wikipedia.org/w/index.php?title=FTDI&oldid=1011612434>.
- [24] Ishtiaque Amin und James Lockridge. „Daisy Chain Implementation for Serial Peripheral Interface“. In: (2018). URL: <https://www.ti.com/lit/an/slvae25a/slvae25a.pdf> (besucht am 04.11.2021).
- [26] Piyu Dhaker. „Introduction to SPI interface“. In: *Analog Dialogue* 52.3 (2018), S. 49–53.
- [27] *ST7565R (Display controller) Datasheet*. Sitronix. 2006. URL: <https://www.lcd-module.de/eng/pdf/zubehoer/st7565r.pdf> (besucht am 21.10.2021).
- [30] Wikipedia contributors. *Bare machine — Wikipedia, The Free Encyclopedia*. [Online; Stand 1. Dez 2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Bare_machine&oldid=1043613167.
- [31] Ann Steffora Mutschler. *Bare Metal Programming*. SEMICONDUCTOR ENGINEERING. 2018. URL: <https://semiengineering.com/bare-metal-programming/> (besucht am 12.11.2021).
- [32] Raj Kamal. *Embedded systems: architecture, programming and design*. Tata McGraw-Hill Education, 2011.
- [33] *FreeRTOS Documentation*. FreeRTOS™. URL: <https://www.freertos.org/implementing-a-FreeRTOS-task.html> (besucht am 05.11.2021).
- [35] Michael Barr. *Embedded C Coding Standard*. BARR group, 2018.
- [38] Nrusingh Prasad Dash u. a. „Event driven programming for embedded systems - a finite state machine based approach“. In: *The Sixth International Conference on Systems* (2011), S. 23–28.
- [39] David Lafreniere. *State Machine Design in C*. 2019. URL: <https://www.codeproject.com/Articles/1275479/State-Machine-Design-in-C> (besucht am 10.11.2021).
- [43] Anselm Busse und Jan Richling. „Fehlersuche auf Embedded-Systemen“. In: *Linux Magazin* 09 (2013). URL: <https://www.linux-magazin.de/ausgaben/2013/09/embedded-debugging/> (besucht am 06.11.2021).
- [44] *SWD*. SEGGER Microcontroller GmbH. 2015. URL: <https://wiki.segger.com/SWD> (besucht am 05.11.2021).

- [46] *J-Link / J-Trace User Guide*. SEGGER Microcontroller GmbH. 2021. URL: https://www.segger.com/downloads/jlink/UM08001_JLink.pdf (besucht am 05.11.2021).
- [47] ybhuangfugui. *In depth master bin, hex, AXF and ELF file formats*. chowdera. 2021. URL: <https://chowdera.com/2021/02/20210219215621853j.html> (besucht am 12.11.2021).
- [50] *STM32L471 Reference Manual*. STMicroelectronics. 2021. URL: <https://www.st.com/en/microcontrollers-microprocessors/stm32l4x1.html#documentation> (besucht am 19.11.2021).
- [52] Wikipedia. *Gerätetreiber — Wikipedia, die freie Enzyklopädie*. [Online; Stand 3. Januar 2022]. 2021. URL: <https://de.wikipedia.org/w/index.php?title=Ger%C3%A4tetreiber&oldid=216981321>.

Programm-Code

- [15] *CMSIS STM32F446xx Device Peripheral Access Layer Header File, BSD 3-Clause license*. STMicroelectronics. URL: <stm32f446xx.h>.
- [16] *CMSIS Cortex-M4 Core Peripheral Access Layer Header File, Apache-2.0 license*. ARM. URL: core_cm4.h.
- [28] *Polling for GPIO-State*. Eigene Quelle. URL: gpio_poll.c.
- [51] *Utility-Funktionen*. Eigene Quelle. URL: <utility.c>.
- [54] *Display-Treiber*. Eigene Quelle. URL: <display.c>.
- [55] *Font-Library*. Eigene Quelle. URL: <font.c>.
- [56] *Peripherie-Initialisierung*. Eigene Quelle. URL: <init.c>.
- [57] *Main-Applikation*. Eigene Quelle. URL: <main.c>.