

Fakultät für Informatik
Studiengang Informatik - Embedded Systems

Evaluation und Entwicklung eines Update-fähigen
Bootloaders für industrielle
Mikrocontrolleranwendungen

Master Thesis

von

Thomas Martin Randl

Datum der Abgabe: 16.07.2021

Erstprüfer: Prof. Dr. Wolfgang Mühlbauer
Zweitprüfer: Prof. Dr. Florian Künzner
Betreuer: Klaus Bachmeier (M.Sc.)

EIGENSTÄNDIGKEITSERKLÄRUNG / DECLARATION OF ORIGINALITY

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Rosenheim, den 16.07.2021



Thomas Martin Randl

Kurzfassung

In der vorliegenden Masterarbeit wird der Entwicklungsprozess eines Update-fähigen Bootloaders im Kontext der eingebetteten Systeme erläutert. Dabei wird auf das mangelhafte Angebot an öffentlich zugänglichen und verwendbaren Implementierungen verwiesen und die daraus resultierende Motivation, einen derartigen Bootloader zu entwickeln. Es wird aufgezeigt, welche Schritte unternommen wurden, um den Bootloader für ein möglichst breites Spektrum an eingebetteten Systemen zugänglich zu machen. Dabei wird auch dessen Limitierung auf die ARM® Cortex-M4® Mikroprozessor-Familie erläutert. Die Arbeit gliedert sich in zwei große Themenbereiche. So wird zunächst zum besseren Verständnis ein theoretischer Einblick in den Themenbereich gegeben. Anhand dieses Wissens wird anschließend im praktischen Teil der Arbeit der Entwicklungsprozess erläutert. Dabei wird zunächst auf die Architektur im Hinblick auf Hardware- und Softwareanforderungen eingegangen. Die technische Umsetzung stellt anschließend den Kern der Arbeit dar. In diesem Schritt wird der Entscheidungsprozess für die Konfigurationsmöglichkeiten und Schnittstellen des Update-Mechanismus dargelegt. Darüber hinaus gibt die Arbeit an dieser Stelle einen tieferen Einblick in die Bereiche Speicherverwaltung und Sprungadressierung eines Bootloaders. Zusätzlich werden die Schritte erläutert, die für ein Upgrade der Bootloader-Software notwendig sind. Darüber hinaus wird die Veröffentlichung unter der 3-Klausel-BSD Lizenz erläutert, bevor abschließend gezeigt wird, anhand welcher Teststrategien die Einsatzbereitschaft des entwickelten Bootloaders sichergestellt wird.

Schlagworte: Bootloader, Eingebettete Systeme, ARM® Cortex-M4®, STMicroelectronics

Inhaltsverzeichnis

Abkürzungsverzeichnis	iii
Abbildungsverzeichnis	v
Tabellenverzeichnis	vii
Codeauszugsverzeichnis	ix
1 Einleitung	1
2 Funktion und Anwendungsbereiche eines Bootloaders	5
2.1 Allgemeine Funktionsweise	5
2.2 Einsatzgebiete	7
2.2.1 Desktop-Betriebssysteme	7
2.2.2 Eingebettete Systeme	10
2.3 Verbreitete Loader-Architekturen	11
2.3.1 Grand Unified Bootloader (GRUB)	12
2.3.2 Universal Bootloader (U-BOOT)	13
3 Bootloader im Kontext eingebetteter Systeme	15
3.1 Anforderungen	15
3.2 Funktionsübersicht	18
3.3 Grundlegende Architekturentscheidungen	20
3.4 Anwendung dieser Konzepte in der STM32-Architektur	23
4 Aufbau und Architektur	25
4.1 Eingebettetes System	26
4.1.1 Auswahl der Hardware	26
4.1.2 ARM® Cortex-M4®-Prozessoreigenschaften	29
4.1.3 Schnittstellen zum Dateitransfer	29
4.2 Trivial File Transfer Protokoll (TFTP)-Server	31
4.2.1 Auswahl der Hardware	31
4.2.2 TFTP-Protokoll	32
5 Implementierung des Bootloaders	37
5.1 STM32CubeIDE	38
5.2 Speicherunterteilung	39
5.2.1 Vollständige Speicherzuweisung für die Hauptanwendung	40
5.2.2 Zusätzliches Backup der Hauptanwendung	41

5.3	Zustandsautomat	44
5.3.1	Bootprozess ohne Backup-Funktion	44
5.3.2	Bootprozess mit Backup-Funktion	46
5.4	Firmware-Update	48
5.4.1	Direktes USB-Update	48
5.4.2	Remote Ethernet-Update	50
5.4.3	Speicherverwaltung	53
5.5	Kontrollmechanismen	54
5.6	Anbindung des optionalen LC-Displays	57
5.7	Sprung in die Hauptanwendung	58
6	Upgrade des Bootloaders	61
6.1	Direktes Upgrade	61
6.2	Indirektes Upgrade	62
6.2.1	Ablauf	63
6.2.2	Implementierung	64
7	Anpassung der Implementierungsdaten	67
7.1	Build-Prozess der Binärdateien	67
7.1.1	Shell-Skript zur Anpassung der Firmware-Updatedatei	67
7.1.2	Shell-Skript zur Anpassung der Sonderfirmware-Updatedatei für das indirekte Bootloaderupgrade	69
7.2	Lizenzierung	70
7.2.1	Überblick Open-Source-Lizenzierung	71
7.2.2	Auswahlprozess	72
8	Durchgeführte Tests am System	75
8.1	Debuggen mittels Entwicklungsumgebung und SWD-Schnittstelle	75
8.2	Durchgeführte Tests an der Release-Version	76
8.3	Überprüfung der TFTP-Schnittstelle	79
8.4	Portierung auf kompatible Hardware	81
9	Zusammenfassung und Ausblick	83
Literaturverzeichnis		xi

Abkürzungsverzeichnis

BIOS	Basic Input/Output System
BOOTP	Bootstrap Protocol
CRC	Cyclic Redundancy Check
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
EFI	Extensible Firmware Interface
EEPROM	Electrically Erasable Programmable Read-Only Memory
ESP	EFI System Partition
FATFS	File Allocation Table Dateisystem
FPU	Floating Point Unit
FTP	File Transfer Protocol
GPIO	General Purpose Input/Output
GRUB	Grand Unified Bootloader
ICER	Interrupt clear-enable Register
IDE	Entwicklungsumgebung
ISPR	Interrupt set-pending Register
ISR	Interrupt Service Routine
IVT	Interrupt Vector Table
JTAG	Joint Test Action Group
LCD	Liquid Crystal Display
LED	Leuchtdioden
MBR	Master Boot Record
MCU	Mikrocontroller-Einheit

MIT	Module Interface Table
MPU	Memory Protection Unit
NVIC	Nested Vectored Interrupt Controller
OSI	Open-Source-Initiative
OSS	Open-Source-Software
PC	Programm-Counter
RAM	Random Access Memory
ROM	Read only Memory
SCB	System Controll Block
SP	Stack-Pointer
SSH	Secure Shell
STM	STMicroelectronics
SWD	Serial Wire Debug
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
UDP	User Datagram Protocol
UEFI	Unified Extensible Firmware Interface

Abbildungsverzeichnis

2.1 Allgemeiner Ablauf eines Bootprozesses	6
2.2 Speicher- und Sprungübersicht eines BIOS Bootprozesses	8
2.3 Speicher- und Sprungübersicht eines UEFI Bootprozesses	10
2.4 GRUB2 Bootmenü für Red Hat Enterprise Linux	12
3.1 Beispielhafter Ablauf eines Bootloaders	18
3.2 Speicherübersicht Backupoption	19
3.3 Speicherzuweisung Flash-Banking	21
3.4 Speicherunterteilung monolithischer Bootloader	22
4.1 Architekturübersicht des Zielsystems	25
4.2 NUCLEO-F429ZI	27
4.3 Trivial File Transfer Protokoll (TFTP) Infrastruktur	31
4.4 TFTP Nachrichtenformate	33
5.1 Einteilung des Programmspeichers ohne Backup-Funktion	40
5.2 Einteilung des Programmspeichers mit Backup-Funktion	42
5.3 Zustandsautomat der Bootloader-Routine ohne Backup-Mechanismus	44
5.4 Zustandsautomat der Bootloader-Routine mit Backup-Mechanismus	46
5.5 Speicherbelegung durch Backup-Mechanismus	47
5.6 LWIP TCP/IP Implementierung	51
5.7 Wireshark-Aufzeichnung einer TFTP-Übertragung	52
5.8 Anzeigeoptionen des LC-Displays	57
6.1 Zustandsautomat des indirekten Bootloaderupgrades	63
7.1 Zusammensetzung der Updatedatei für die Applikation	68
8.1 Speicherbelegung im Produktivsystem	78
8.2 Testaufbau zur Überprüfung des TFTP-Update-Mechanismus	80

Tabellenverzeichnis

4.1 Pakettypen des Trivial File Transfer Protokolls (TFTP)	33
4.2 Fehlernachrichten des TFTP	34
5.1 Konfigurationsoptionen des Bootloaders	55
5.2 Farbbe bedeutung des LED Feedback-Mechanismus	56
7.1 Parameter für den Aufruf des Shell-Skripts der Firmware-Updatedatei	68
7.2 Parameter für den Aufruf des Shell-Skripts der Sonderfirmware des Bootloaderupgrades	69
8.1 Testprotokoll für den Release einer neuen Bootloader-Version	77

Codeauszugsverzeichnis

5.1 Angepasstes Linker-Skript für eine Speicherkonfiguration ohne Backup	41
5.2 Angepasstes Linker-Skript für eine Speicherkonfiguration mit Backup	43
5.3 Blockweises Lesen der Updatedatei über die USB-Schnittstelle	50
5.4 Anpassen des Offsets der Interrupt Vektortabelle	58
5.5 Vorbereitung und Sprung in die Hauptanwendung	59
6.1 Implementierung des Zustandsautomaten für das Bootloaderupgrade	64
7.1 3-Klausel-BSD-Lizenztext für das Bootloader-Projekt	73
8.1 Konfiguration der Datei „/etc/xinet.d/tftp“	80

1 Einleitung

Mit der fortschreitenden Industrialisierung durch Erscheinungen wie der Industrie 4.0 bietet sich Unternehmen in den letzten Jahren ein viel größeres Feld an Möglichkeiten zur Steigerung der Gewinne [1]. Diesbezüglich wird allein in Deutschland bis zum Jahr 2025 eine Steigerung des Wertschöpfungspotenzials von bis zu 78,77 Milliarden Euro für die Branchen Maschinen- und Anlagenbau, Elektrotechnik, Automobilbau, chemische Industrie, Landwirtschaft und Informatik erwartet [1]. Allein der Anteil an Mikrocontrollern und -prozessoren soll bis 2022 weltweit auf einen Umsatz von 78,16 Milliarden US-Dollar ansteigen [2].

Anhand dieser Zahlen lässt sich ein klarer Trend erkennen, welches Potenzial die weltweite Industrie in Bereichen wie der automatisierten Fertigung sieht. Ein weiteres Indiz, das diesen Trend unterstreicht, ist die zunehmende Robotisierung in der verarbeitenden Industrie. So ist allein in Deutschland die Zahl der eingesetzten Roboter im Jahr 2019 um 20.500 auf insgesamt 221.500 angestiegen. Führend sind hier die Länder Südkorea mit 855 Robotern pro 10.000 Beschäftigten sowie auch China. Dieses hatte allein im Jahr 2019 einen Zuwachs von 140.500 neuen Robotern [3].

Da jedes dieser Systeme zu einem bestimmten Zweck eingesetzt wird, werden diese mit entsprechenden Software-Programmen bespielt. Sollte sich jedoch dieser Zweck ändern, muss das alte Programm durch eine angepasste Version ersetzt werden. Dies kann ein aufwendiger Prozess sein, welcher nur durch Fachpersonal durchgeführt werden kann. In Zeiten der Globalisierung ist dies teilweise mit langen Anreisen dieses Personals verbunden und birgt sowohl Zeit- als auch hohe Servicekosten für diese Umstellung des Systems.

Viel praktischer wäre daher der Einsatz eines Mechanismus, der unabhängig von der Hauptanwendung des Systems ohne die Notwendigkeit von Fachpersonal einen reibungslosen Start- und Update-Prozess gewährleistet. Ein solcher Mechanismus trägt den Namen Bootloader und stellt ein eigenständiges Programm im System dar, welches bei Systemstart ausgeführt wird. Dieser kann in vielen Variationen zum Einsatz kommen. Die grundlegende Aufgabe besteht allerdings im Start der Hauptanwendung.

Ein klassisches Einsatzgebiet für Bootloader ist der Bereich der Desktop-Betriebssysteme. Ein solcher wird hier hauptsächlich zur Auswahl und zum Start der im System installierten Betriebssysteme genutzt. Der Update-Prozess des Betriebssystems wird durch einen Update-Manager übernommen. Im Bereich der eingebetteten

1 Einleitung

Systeme ist dies jedoch nicht einheitlich geregelt. Dies ist unter anderem auf begrenzte System-Ressourcen zurückzuführen. Allerdings werden eingebettete Systeme häufig auch individuell für einen Anwendungsfall programmiert und sind im Gegensatz zu Desktop-Betriebssystemen alles andere als einheitlich bezüglich der ausgeführten Anwendung. So werden häufig sehr individuelle Betriebssysteme verwendet. Bei kompakten Systemen kommt darüber hinaus oft gar kein Betriebssystem zum Einsatz.

Das bedeutet jedoch nicht, dass Bootloader in diesem Sektor nicht weit verbreitet sind. Der Einsatz solcher wird individuell von der Entwicklerfirma angestoßen und in der Regel unter Verschluss gehalten. Zusätzlich ist ein Bootloader im Bereich der eingebetteten Systeme stark von der eingesetzten Hardware abhängig. Diese kann jedoch von System zu System unterschiedlich variieren, was eine Wiederverwendung des entwickelten Bootloaders erschwert. Dies stellt auch ein Problem dar, welches nicht ohne Weiteres behoben werden kann. Um einen Bootloader zu entwickeln, welcher für möglichst alle Hardwarekonfigurationen einsetzbar ist, wäre eine sehr umfangreiche Konfigurationsoberfläche notwendig. Dieses Projekt kann daher nur ein langfristiges Ziel für eine Vielzahl von Entwicklern sein.

Zusätzlich ist die Anzahl der frei verfügbaren Bootloader-Implementierungen für kompakte Systeme gering. Diese sind darüber hinaus meist sehr speziell und bieten kaum Konfigurationsmöglichkeiten. Um jedoch eine Möglichkeit für einen besseren Einstieg in das komplexe Thema der Bootloader im Kontext der eingebetteten Systeme zu liefern, wurde im Zuge der vorliegenden Arbeit ein solcher entwickelt. Mit diesem werden die angesprochenen Problematiken in Angriff genommen.

Die Entwicklung der ersten voll funktionsfähigen Version wurde in Kooperation mit der Firma AN Dispensing UG durchgeführt. Die Kernanforderung stellte dabei die Portierbarkeit im Bereich der ARM® Cortex-M4®-Mikroprozessoren dar. Zusätzlich soll der Bootloader über diverse Konfigurationsmöglichkeiten bezüglich des Speicherlayouts und der Update-Schnittstellen verfügen. Auch ein Upgrade-Mechanismus für den Bootloader selbst muss in der Implementierung berücksichtigt werden. Darüber hinaus soll dieser unter einer Open-Source-Lizenz veröffentlicht werden, um zukünftige Erweiterungen durch die Open-Source-Community anzuregen. Dies soll das langfristige Ziel einer einfachen Update-Möglichkeit für die steigende Zahl der eingebetteten Systeme im industriellen Umfeld voranbringen.

Zur Erläuterung dieses Entwicklungsprozesses gliedert sich die Arbeit in mehrere Teilbereiche. Zunächst wird in Kapitel 2 ein theoretischer Einblick in die Thematik Bootloader angestoßen. Dabei liegt der Schwerpunkt auf der allgemeinen Funktionsweise und den verbreitetsten Einsatzgebieten. Zusätzlich werden gängige Bootloader-Implementierungen beispielhaft erläutert. Diese gewonnenen Erkenntnisse werden anschließend in Kapitel 3 in den Kontext der eingebetteten Systeme übertragen. Dabei sollen die resultierenden Problematiken klargestellt werden.

Der praktische Teil dieser Arbeit befasst sich zunächst in Kapitel 4 mit der Planung und Struktur der Bootloader-Implementierung. Im nächsten Schritt wird in Kapitel 5 deren Umsetzung beschrieben. Um die Bootloader-Software selbst zu erneuern, werden daraufhin in Kapitel 6 zwei Möglichkeiten vorgestellt. Als Nächstes wird in Kapitel 7 beschrieben, wie eine Anwendungssoftware angepasst werden muss, um durch den Bootloader verwendbar zu sein. Dabei wird zusätzlich auf die Open-Source-Lizenzierung der entwickelten Software eingegangen. Abschließend wird in Kapitel 8 gezeigt, anhand welcher durchgeföhrter Tests die Stabilität der Software gewährleistet werden kann, bevor ein Fazit gezogen wird.

Zum Zweck des besseren Verständnisses werden im Zuge dieser Arbeit die Bereiche Bootloader und Systemanwendung anhand der Begrifflichkeiten getrennt. So wird die eigentliche Anwendung des Systems mit den Begriffen **Firmware**, **Applikation** und **Hauptanwendung** bezeichnet. Der Bootloader hingegen wird immer als solcher genannt. Im Bezug auf Softwareaktualisierung wird für den Bootloader der Begriff **Upgrade** verwendet. Eine Aktualisierung der Hauptanwendung hingegen wird ausschließlich als **Update** bezeichnet.

2 Funktion und Anwendungsgebiete eines Bootloaders

In diesem Kapitel wird auf die Notwendigkeit für den Einsatz eines Bootloaders eingegangen. Hierfür werden zunächst gängige Begriffe erläutert. Anschließend wird auf die grundlegenden Funktionen eines Bootloaders eingegangen. Darüber hinaus wird ein Überblick über die verbreitetsten Anwendungsfälle den Nutzen eines solchen Bootloaders aufzeigen.

2.1 Allgemeine Funktionsweise

Der Begriff „Boot“ beschreibt einen Prozess, welcher die Ausführung einer Software zur Folge hat [4]. Um dies zu bewerkstelligen, muss der für den Bootprozess zuständige Bootloader das erste Programm sein, welches nach der Hardwareinitialisierung ausgeführt wird [5]. Ein solcher Bootloader stellt dabei selbst eine implementierte Anwendung dar. Diese bildet also die Schnittstelle zwischen Hardwareinitialisierung und Hauptprogramm [6]. Um die Hauptanwendung aus dem Speicher auszuführen, muss das Bootloader-Programm sämtliche Kontrolle über das System abgeben und den Speicherbereich der geladenen Software zum neuen Startpunkt des Systems erklären [7]. Aus diesem Grund bietet es sich an, möglichst wenig Hardware zu initialisieren. Meist genügt es, die Systemuhr, die Interrupt Service Routine (ISR) und den Speicher zu initialisieren [5]. Dies ist abhängig von dem Einsatzgebiet des Bootloaders.

Da ein Bootloader für eine Vielzahl verschiedener Geräte verwendet werden kann, gibt es entsprechend viele Implementierungen für die jeweiligen Rahmenbedingungen. Ein Überblick über die gängigsten Anwendungsgebiete findet sich in Kapitel 2.2. Obwohl die Anwendungsfälle sich häufig in Anforderung und Hardware unterscheiden, gibt es ein Grundkonzept, nach dem alle Bootloader aufgebaut sind [6]. So besteht ein Bootprozess grundlegend aus mindestens drei Komponenten. Diese sind der „Branch“, der Bootloader und eine oder mehrere Anwendungen. Wie in Abbildung 2.1 zu erkennen ist, wird direkt nach dem Start der Branch aufgerufen. Dieser entscheidet, ob ein Bootloader ausgeführt werden soll. Diese Entscheidung wird meist anhand einer Eingabe oder des Zustandes eines General Purpose Input/Output (GPIO) Ports getätigt. Zur Stabilisierung des Bootprozesses wird dieser Branching-Prozess häufig in den Bootloader integriert. Dies bietet den Vorteil, dass eine weitreichendere Entscheidung für den Start der Applikation gefällt werden kann. So ergibt sich beispielsweise die Möglichkeit, den Speicherbereich der Anwendung

2 Funktion und Anwendungsgebiete eines Bootloaders

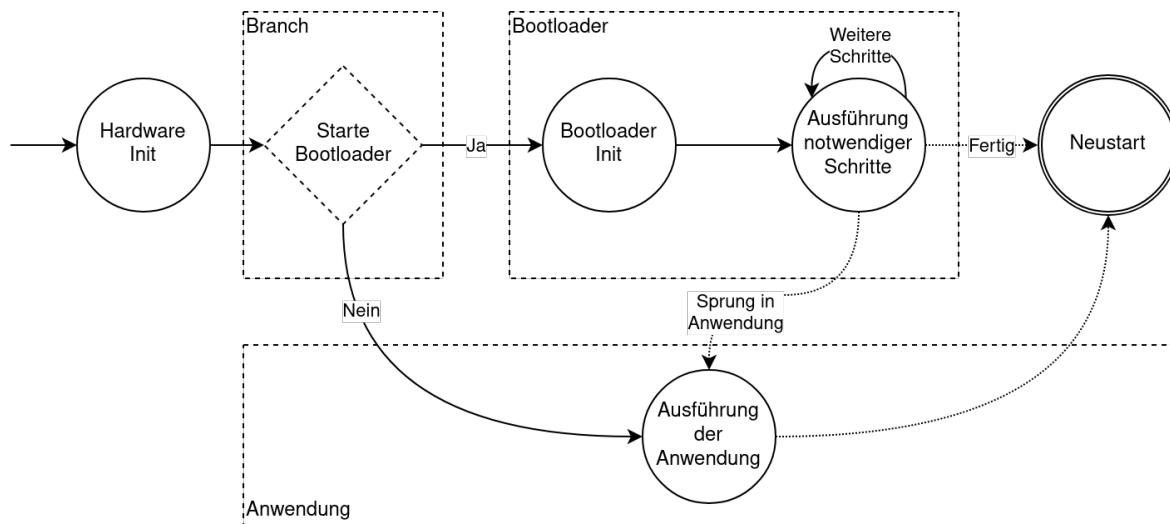


Abbildung 2.1 Allgemeiner Ablauf eines Bootprozesses

Quelle: Eigene Darstellung in Anlehnung an [6]

auf Validität zu prüfen [6]. Darüber hinaus besteht bei einer einfacheren Branching-Lösung die Gefahr, dass eine Signalstörung des GPIO Ports ein unerwünschtes Verhalten hervorruft. Sollte dennoch ein externer Branching-Prozess gewünscht sein, muss eine fehlerresistenterere Lösung gefunden werden [6].

Der Bootloader selbst kann neben dem Start der Anwendung zusätzliche Aufgaben bekommen. So kann ein solcher in der Lage sein, etwaige Updates für diese Applikation durchzuführen und deren Lauffähigkeit zu validieren. Unter anderem wegen dieser beiden Anforderungen gilt eine strikte Trennung von Bootloader und Anwendung im Bezug auf die autarke Ausführung. Diese wird unter anderem durch die strikte Aufteilung des zur Verfügung stehenden Speichers erreicht. Hierfür werden dem Bootloader und der Anwendung feste Speicherbereiche zugewiesen. So weiß der Bootloader, an welcher Adresse im Speicher die Anwendung liegt und kann auf diese Speicherbereiche zugreifen [8].

In Abbildung 2.1 ist der grundlegende Ablauf, den ein Bootloader durchläuft, dargestellt. Zunächst initialisiert dieser die benötigten Komponenten wie Systemuhr, ISR und die verwendeten Schnittstellen. Diese gilt es je nach System auf ein Minimum zu beschränken, da jede weitere Schnittstelle auch mehr Speicherverbrauch der Anwendung bedeutet. Nach der Initialisierung wird dann die Bootloaderroutine durchlaufen [6]. Dies kann entweder automatisch ablaufen, oder mit Hilfe von Nutzereingaben geschehen [5]. Nach Abschluss dieser Routine gibt es, wie in der Abbildung zu erkennen, zwei Möglichkeiten, wie das Hauptprogramm aufgerufen werden kann. Die erste Option ergibt sich durch einen Neustart des Systems. In diesem Fall muss der Branch außerhalb der Bootloaderanwendung liegen und der Bootloader ist wiederum gezwungen, diesem eine entsprechende Information zu hinterlegen um die Anwendung zu starten. Die zweite Möglichkeit besteht darin,

direkt von der Bootloaderanwendung in das Hauptprogramm zu „springen“ [6]. In diesem Fall muss der Bootloader vor dem Sprung sämtliche beanspruchte Ressourcen frei geben [7].

Die Darstellung der Anwendung in Abbildung 2.1 zeigt, dass diese unabhängig vom Programm des Bootloaders eine eigene Applikation ist. Falls es jedoch von Nöten ist, aus dem Hauptprogramm zurück in den Bootloader zu gelangen, muss dies ebenfalls über einen Neustart des Systems und eine entsprechende Information an den Branch erfolgen [6]. Sollte die Anwendung während des Betriebs auf Teile des Bootloaders zugreifen müssen, dann benötigt diese die zugehörigen Speicheradressen. Sollte der Bootloader über keinen Mechanismus verfügen, sich selbst zu aktualisieren, stellt dies kein Problem dar. In diesem Fall bleiben die Startadresse oder etwaige Adressen im Bootloader dauerhaft unverändert. Problematisch wird dies jedoch, wenn der Bootloader selbst upgradefähig ist. In diesem Fall muss die Schnittstelle zwischen Bootloader und Applikation über eine Module Interface Table (MIT) definiert werden. Mit Hilfe dieser MIT können Adresszugriffe auf bestimmte Speicheradressen erfolgen, auch wenn sich diese bei einem Upgrade des Bootloaders verändern [8].

2.2 Einsatzgebiete

Die Einsatzgebiete für einen Bootloader lassen sich fast ausschließlich auf zwei Kategorien eingrenzen. Zum einen den Einsatz im Zusammenhang mit einem oder mehreren Betriebssystemen. Zum anderen den Gebrauch im Bereich der eingebetteten Systeme. Wie die Kategorisierung bereits vermuten lässt, ist die Implementierung eines Bootloaders daher oft auf den entsprechenden Anwendungsfall zugeschnitten und lässt sich, wenn überhaupt, nur mit hohem Aufwand auf ein anderes System portieren [9]. Dieses Kapitel befasst sich mit den allgemeinen Eigenschaften dieser Einsatzgebiete und was diese im Kern unterscheidet. Explizite Beispiele für Bootloaderimplementierungen finden sich in Kapitel 2.3.

2.2.1 Desktop-Betriebssysteme

Ein Bootloader im Kontext mit einem oder mehreren Betriebssystemen hat einen etwas eingeschränkteren Funktionsumfang. In der Regel liegt die Hauptaufgabe darin, das Betriebssystem zu starten. Das heißt, dieses aus dem Systemspeicher in den Arbeitsspeicher zu laden. Sollten auf dem Computer mehrere Betriebssysteme im Einsatz sein, dann regelt ein Bootloader zusätzlich die Auswahl des gewünschten Systems durch Nutzereingabe. Der Update-Mechanismus des Betriebssystems wird auch von diesem übernommen und liegt daher nicht in der Zuständigkeit des Bootloaders [7].

Eine Besonderheit bei Betriebssystemen liegt darin, dass jedes Betriebssystem über einen eigenen Bootloader verfügt. Dieser startet jedoch nicht, wie in Kapitel 2.1 beschrieben, direkt nach der Hardwareinitialisierung. Da auf einem Computer mehrere

2 Funktion und Anwendungsgebiete eines Bootloaders

Betriebssysteme installiert sein können und es die Möglichkeit gibt, nach Systemstart eines davon auszuwählen, wird ein Mechanismus benötigt, um das richtige Betriebssystem mit dem entsprechenden Bootloader zu laden. Dieser Schritt wird meist mit Hilfe eines Basic Input/Output System (BIOS) oder der neueren Version eines Unified Extensible Firmware Interface (UEFI) durchgeführt [4]. Es gibt auch noch andere Softwares für diesen Zweck. Da BIOS und UEFI die verbreitetsten darstellen, werden diese im Folgenden näher betrachtet.

Das BIOS ist eine veraltete Software, die jedoch immer noch Anwendung findet. Es weist einige Limitierungen auf, die nach heutigem Standard eine Einschränkung bezüglich der Verwendung der Hardware beinhalten. So unterstützt ein BIOS lediglich vier primäre Partitionen. Das heißt, auf einem Computer können laut Vorsehung nur vier Betriebssysteme installiert werden. Dies liegt daran, dass der Master Boot Record (MBR) lediglich vier 16-Byte-Adressen beinhalten kann. Es besteht jedoch die Möglichkeit, die vierte Partition in logische Partitionen zu unterteilen, weshalb diese Einschränkung für Betriebssysteme, die derartige Partitionen unterstützen, umgangen werden kann. Diese logischen Partitionen können jedoch nicht durch das BIOS gelesen werden. Zusätzlich darf eine Partition nicht größer als 2,2 TiB sein [4].

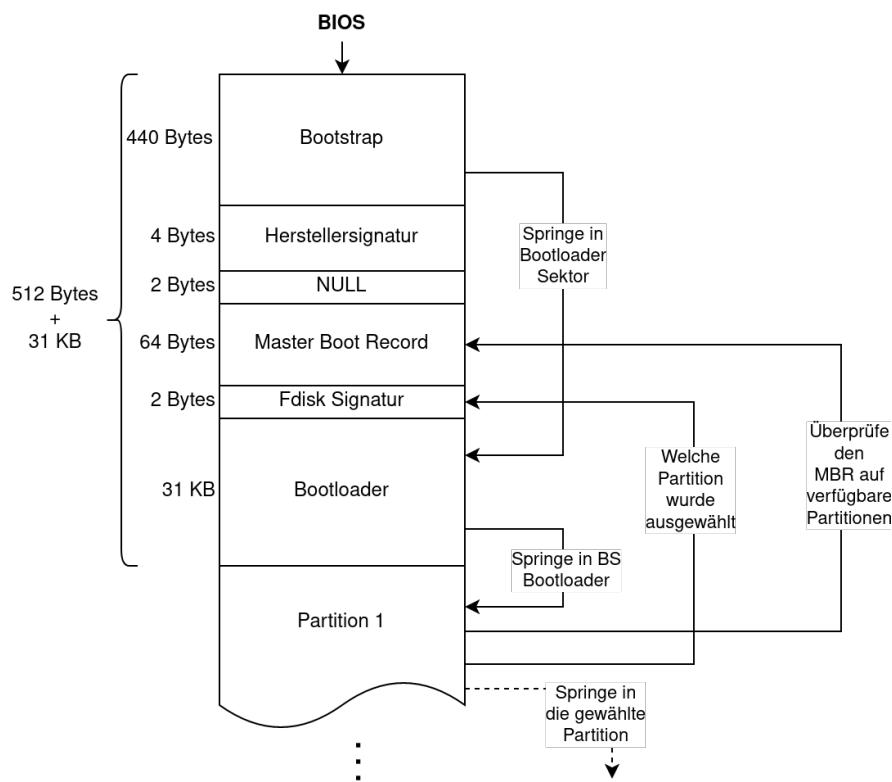


Abbildung 2.2 Speicher- und Sprungübersicht eines BIOS Bootprozesses
Quelle: Eigene Darstellung in Anlehnung an [4]

Das BIOS verwendet selbst auch eine Art Bootloader. Dieser ist in Abbildung 2.2 zu erkennen. Insgesamt wird der Bootloader in drei Teile gespalten. Der erste davon ist der „Bootstrap“. Dieser wird durch das BIOS aufgerufen. Da der Speicher auf 440 Byte beschränkt ist, lautet die primäre Aufgabe, in den zweiten Teil des verwendeten Bootloaders zu springen. Dieser ist mit 31 KiB immer noch recht klein, jedoch ebenfalls in der Lage, einen Sprung durchzuführen. Der dritte Teil liegt im Bereich der ersten primären Betriebssystem-Partition. Dessen Speicherbedarf hat zunächst keine Einschränkungen in der Größe, weshalb nun genügend Speicher für die benötigten Operationen zur Verfügung steht. Dieser überprüft den MBR auf die verfügbaren Betriebssysteme. Anschließend wird die „Fdisk Signatur“ ausgewertet. Diese gibt Auskunft über die zum Boot ausgewählte Partition.

Falls das ausgewählte Betriebssystem tatsächlich in der ersten Partition liegt, wird der Bootloader nun den Kernel des Betriebssystems in den Arbeitsspeicher laden. Sollte jedoch ein anderes Betriebssystem ausgewählt worden sein, so führt dieser einen Sprung zum betriebssystemeigenen Bootloader in die Partition dieses Betriebssystems durch. Diese Aufteilung und die dadurch umständlichen Sprünge liegen darin begründet, dass BIOS eine alte Software ist und zur Entwicklungszeit die 31,5 KiB, die für das Laden des Betriebssystems vorgesehen sind, ausreichend waren. Der dreigeteilte Bootloader, welcher Verwendung findet, ist abhängig von dem Betriebssystem in der ersten Partition. Windows verwendete beispielsweise den „New Technology Loader“ bis einschließlich Windows XP [4]. Seither wird jedoch der sogenannte „Bootmgr“ eingesetzt [10]. In vielen Unix-basierten Betriebssystemen wird hingegen der Grand Unified Bootloader (GRUB) verwendet [4]. Da dieser sehr weit verbreitet ist, wird er exemplarisch für Betriebssystem-Bootloader in Kapitel 2.3.1 genauer betrachtet.

Zur Behebung der Probleme, welche durch BIOS im Bezug auf moderne Hardware entstehen, wurde UEFI eingeführt. Diese Technologie hält keinerlei Beschränkung mehr bezüglich einer maximalen Anzahl an primären Partitionen und unterstützt Partitionen mit einer Größe von bis zu 8 ZiB. Dies liegt daran, dass der MBR durch die GUID Partition Table ersetzt wurde. Zusätzlich sind für den Aufruf einer Partition immer fünf Sprünge notwendig. In Abbildung 2.3 ist ein UEFI Bootprozess abgebildet. Anhand der Pfeile sind die fünf Sprünge ersichtlich, die das UEFI unternimmt, um den korrekten Bootloader auszuwählen und das Betriebssystem zu starten. Zusätzlich verfügt das UEFI über eine eigene Partition. Diese trägt den Namen EFI System Partition (ESP) und hat eine minimale Größe von 256 MiB. Bei der Installation eines Betriebssystems wird nun im Extensible Firmware Interface (EFI) Verzeichnis in der ESP ein Eintrag für dieses erzeugt. Der Bootloader des Betriebssystems wird anschließend in diesem reservierten Bereich gespeichert. Wie in Abbildung 2.3 zu erkennen ist, werden diese Betriebssystemeinträge unabhängig voneinander in die ESP gespeichert. Daraus resultiert die feste Definiertheit des Ladeprozesses des Betriebssystembootloaders. Sobald dieser durch das UEFI aufgerufen wird, lädt er den Kernel des gewünschten Betriebssystems in den Arbeitsspeicher und springt in die entsprechende Partition. Welcher Bootloader dabei zum Start des

2 Funktion und Anwendungsgebiete eines Bootloaders

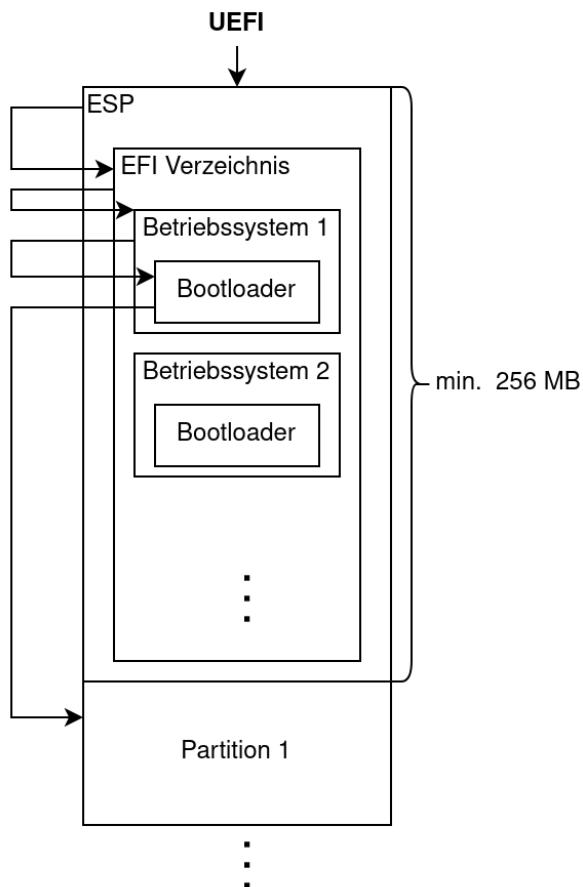


Abbildung 2.3 Speicher- und Sprungübersicht eines UEFI Bootprozesses
Quelle: Eigene Darstellung in Anlehnung an [4]

Systems ausgeführt wird, legt ein Eintrag in der UEFI Bootkonfiguration fest. Dieser Bootloader ist bei einem Multiboot-System in der Lage, alle anderen Systeme zu starten [4].

2.2.2 Eingebettete Systeme

Der Schwerpunkt der vorliegenden Arbeit liegt in der Realisierung eines Bootloaders für den Bereich der eingebetteten Systeme. In Kapitel 3 werden die daraus resultierenden Anforderungen an diesen Bootloader und der Grund für dessen Einsatz genauer erläutert. In diesem Kapitel wird ausschließlich der Unterschied zu einem Bootloader im Bereich der Desktop-Betriebssysteme, die in Kapitel 2.2.1 aufbereitet wurden, deutlich gemacht.

Während ein Bootloader im Kontext der Desktop-Betriebssysteme lediglich die Aufgabe hat, den Kernel des Wunschsystems in den Arbeitsspeicher des Systems zu laden, übernimmt das Betriebssystem selbst den Großteil der ansonsten anfallenden Aufgaben. Es ist beispielsweise in der Lage, selbstständig Updates durchzuführen.

Eine Anwendung, die in einem eingebetteten System betrieben wird, ist dazu meist nicht in der Lage. Dies liegt darin begründet, dass im Gegensatz zum Desktop-Betriebssystem meist Einschränkungen durch die verwendete Hardware zu beachten sind. Ein eingebettetes System verfügt in der Regel über einen eingeschränkten Speicher und geringere Leistung als ein herkömmlicher Computer [7]. Darüber hinaus wird das Programm meist direkt aus dem Flash-Speicher ausgeführt und muss nicht in den Arbeitsspeicher geladen werden. In Kapitel 2.1 wurde bereits die Update-Funktion von Bootloadern angesprochen. Diese findet hauptsächlich Anwendung im Bereich der eingebetteten Systeme [8].

Auf einem herkömmlichen Desktop-Betriebssystem stellt es kein Problem dar, eine komplette Kopie des Betriebssystems abzulegen. Dieser Sachverhalt ist bei einem eingebetteten System nicht gegeben. Die installierte Anwendung ist daher nicht so einfach dazu in der Lage, sich selbst zu aktualisieren. Zusätzlich wird die Applikation nur für den vorgesehenen Zweck implementiert. Häufig ist auf einem eingebetteten System nicht einmal ein Betriebssystem im Einsatz [7]. Zusätzlich befindet sich in der Regel nur eine Anwendung auf dem System. Daher ist es äußerst unüblich, dass ein Bootloader in diesem Kontext mehrere Applikationen verwaltet [7]. Die Anforderung im Bereich der eingebetteten Systeme liegt daher hauptsächlich im Update der Systemanwendung [6]. Entsprechend dieser Tatsache gibt es eine Vielzahl an Implementierungen für verschiedenste Hardware. Wie diese umgesetzt wurden und was diese im Kern gemeinsam haben, wird in Kapitel 3 allgemein erläutert. In diesem finden sich auch die Schlussfolgerungen, welche Ansätze für den im Zuge dieser Arbeit entwickelten Bootloader in Frage kommen und was diese auszeichnet.

2.3 Verbreitete Bootloader-Architekturen

In Kapitel 2.1 wurde auf allgemeine Eigenschaften von Bootloadern eingegangen. Zusätzlich wurde erläutert, welche Schritte notwendig sind, um den Bootloader eines Desktop-Betriebssystems zu starten. An dieser Stelle wurde auch ermittelt, dass es in allen Bereichen eine Vielzahl verschiedener Bootloader-Architekturen gibt. Diese unterscheiden sich in Desktop-Betriebssystemen anhand der Art des Betriebssystems. Im Sektor der eingebetteten Systeme hingegen sind diese oft abhängig von der jeweiligen Hardware und werden daher häufig extra für die individuelle Hardwarekonfiguration entwickelt. Im nachfolgenden Kapitel werden nun exemplarisch die weitverbreiteten Bootloader GRUB und U-Boot vorgestellt. Diese befassen sich mit dem Laden eines ressourcenintensiven Betriebssystems im Umfeld der Desktopanwendungen oder der eingebetteten Systeme. Warum diese Bootloader nicht als Alternative für den in dieser Arbeit entwickelten Bootloader geeignet sind, wird in Kapitel 3 erläutert.

2 Funktion und Anwendungsgebiete eines Bootloaders

2.3.1 Grand Unified Bootloader (GRUB)

GRUB ist ein Bootloader, der sich in einer Vielzahl von Linux-basierten Betriebssystemen findet. Im Jahr 2012 wurde die aktuelle Version 2 veröffentlicht. Im Zuge dieses Updates wurden einige Erneuerungen vorgenommen, welche die vorherige Version grundlegend veränderte. Diese läuft seither unter dem Namen GRUB-Legacy. Die aktuelle GRUB-Version wird auch häufig als GRUB2 bezeichnet [4].

In der Regel wird GRUB2 im Bereich der Desktop-Betriebssysteme verwendet. Dies liegt darin begründet, dass GRUB2 ein Bootloader für Linux-basierte Betriebssysteme ist, welcher sich auf den Einsatz von Multiboot spezialisiert [11]. Falls auf einem System nur ein Betriebssystem installiert ist, führt GRUB2 lediglich die klassischen Bootloader-Funktionen, wie in Kapitel 2.2.1 beschrieben, aus. Im Falle eines BIOS wird das GRUB2 in drei Teilen gebootet. Der Hauptteil des Bootprozesses findet dabei im dritten Teil statt. Bei Verwendung eines UEFI ist die Partition, welche GRUB2 verwendet, in der Bootreihenfolge der UEFI-Konfiguration als Erstes gelistet. Dadurch weiß das System, welcher Bootloader geladen werden muss [4].

Der große Vorteil an GRUB2 ist, dass dieser nicht nur das zugehörige Betriebssystem bootet. Ferner kann im Fall eines Multiboot-Systems nach Ausführung des GRUB2 eine Auswahl des gewünschten Betriebssystems in Form eines durch GRUB2 bereitgestellten Menüs getroffen werden [12]. Um diese Auswahl treffen zu können, kennt GRUB2 die Kernel der installierten Betriebssysteme auf dem System und stellt diese mittels einer Liste, wie sie in Abbildung 2.4 zu sehen ist, im Menü dar. Dabei ist es üblich, dass mehrere Kernel für eines der Betriebssysteme verfügbar sind. Das ist der Fall, wenn ältere Kernel nach einem Update nicht vom System gelöscht wurden. Dies ist jedoch kein fehlerhaftes Verhalten, sondern liefert die Möglichkeit, einen vergangenen Kernel zu verwenden, sollte der aktuelle in irgendeiner Weise beschädigt sein und nicht mehr hochfahren. Darüber hinaus gibt es häufig eine Option zur Rettung und Diagnose einer installierten Distribution [13]. Um einen dieser Kernel nach dem Auswahlprozess zu booten, gibt es zwei Möglichkeiten: das direkte Laden und das Kettenladen [11].



Abbildung 2.4 GRUB2 Bootmenü für Red Hat Enterprise Linux
Quelle: [13]

Das direkte Booten kann für eine Reihe von Betriebssystemen verwendet werden. Dafür müssen diese auf Linux, FreeBSD, NetBSD oder OpenBSD basieren. Diese unterstützen alle den GRUB2 Befehlssatz und können daher auch die Instruktion „boot“ ausführen. Dies ermöglicht es GRUB2, die Kernel der entsprechenden Distributionen in den Arbeitsspeicher des Systems zu laden und deren Ausführung anzustoßen [11].

Das sogenannte Kettenladen wird verwendet, wenn einer der auf dem System installierten Kernel zu keiner Distribution einer der oben genannten Betriebssysteme gehört. Nur mit dessen Hilfe kann dieses Betriebssystem mittels GRUB2 gestartet werden [12]. Der Prozess des Kettenladens beinhaltet im Gegensatz zum direkten Laden einen zusätzlichen Schritt. Im Wesentlichen muss das GRUB2 den Bootloader der Partition in den Arbeitsspeicher laden, die keinen Multiboot unterstützt. Dieser erfüllt dann seine Aufgaben, als wäre GRUB2 nie ausgeführt worden. Das Kettenladen ist jedoch nur auf Systemen verfügbar, die ein BIOS oder UEFI verwenden [11].

Ein klassisches Beispiel liefert hier der Start einer Windows-basierten Distribution. Da dieses Betriebssystem nativ keinen Multiboot unterstützt [11], müssen in diesem Fall einige Dinge beachtet werden. Zunächst muss bei der Installation der verschiedenen Betriebssysteme berücksichtigt werden, dass die Windows-Installation als erstes stattfindet. Dies liegt daran, dass Windows beispielsweise während der Installation auf einem BIOS-System den MBR und den bereits installierten Bootloader überschreibt [12].

2.3.2 Universal Bootloader (U-BOOT)

Das U-Boot, wie es der Entwickler Wolfgang Denk genannt hat, ist ein Bootloader, der für den Einsatz im Bereich der eingebetteten Systeme gedacht ist. Jedoch sind diese Systeme in der Lage, ein richtiges Desktop-Betriebssystem wie beispielsweise eine Linux-basierte Distribution zu betreiben [14]. Hinzu kommt, dass U-Boot selbst sehr eng mit dem Linux Kernel verwandt ist und sogar einige Programmteile beinhaltet [15].

Der Vorteil dieses Bootloaders liegt in seiner Varianz bezüglich der Hardware. Der Bootloader kann aktuell für mehr als ein Dutzend Architekturen auf über 1000 Boards verwendet werden. Darüber hinaus bietet er Möglichkeiten, ein neues Board mit Hilfe der bestehenden Konfigurationen hinzuzufügen [15]. Zu beachten ist jedoch, dass die Anpassung für das jeweilige Zielsystem in jedem Fall mit Konfiguration verbunden ist. Dies kann zuweilen sehr zeitaufwendig und komplex sein. Die bereits bestehenden Board-Konfigurationen finden sich im Git des Projekts im Ordner „./include/configs“ [14]. Diese Erweiterungen werden zusätzlich durch die Open-Source-Gemeinschaft in einem zweimonatigen Release-Zyklus erweitert. Veröffentlicht ist U-Boot unter der GPLv2 Lizenz. [16]. Näheres zur Lizenzierung findet sich in Kapitel 7.2.

2 Funktion und Anwendungsgebiete eines Bootloaders

Entgegen dem in Kapitel 2.3.1 vorgestellten GRUB2 kann ein U-Boot mehr als nur den Kernel eines Betriebssystems in den Arbeitsspeicher des Systems zu laden und auszuführen [15]. Mit U-Boot kann ein eingebettetes System, welches ein Betriebssystem betreibt, mit Hilfe von USB und sogar Ethernet aktualisiert werden. U-Boot bietet dafür die Protokolle Dynamic Host Configuration Protocol (DHCP), Trivial File Transfer Protocol (TFTP) und Bootstrap Protocol (BOOTP) an. Diese ermöglichen den automatischen Zuweisungsprozess einer Ethernet-Verbindung und die spätere Entgegennahme von Dateien. Im weiteren Verlauf des Bootprozesses bietet U-Boot diverse Möglichkeiten, aus denen das System gebootet werden kann. Darunter befinden sich unter anderem RAM boot, NAND boot und NFS boot. Somit kann sowohl aus dem Arbeitsspeicher, dem Flash-Speicher, als auch über Ethernet gebootet werden. Das Einrichten dieser Mechanismen obliegt jedoch dem Entwickler, der U-Boot konfiguriert [14].

U-Boot liefert die Möglichkeit, Linux oder andere Betriebssysteme im Bereich der eingebetteten Systeme zu verwenden und ersetzt dadurch unter anderem auch die Funktionen von BIOS und UEFI. Der Unterschied liegt darin, dass U-Boot nach dem Start des Betriebssystems sämtliche Kontrolle an dieses abgibt [15]. Dies ist der Verwendung auf einem eingebetteten System verschuldet, welches über begrenzte Ressourcen verfügt.

Der Einsatz von U-Boot bietet sich dann an, wenn ein eingebettetes System verwendet wird, um ein Linux oder ähnliches Betriebssystem zu betreiben. Dies ist insbesondere dann der Fall wenn die Möglichkeiten eines GRUB2 nicht die benötigten Kriterien erfüllt. U-Boot ist ein Bootloader, der sich selbstständig updaten kann. Darüber hinaus kann er nicht nur aus dem Hauptspeicher des Systems booten, sondern zusätzlich aus diversen anderen Quellen [14]. Nur wenn weder ein selbstständiges Update des Bootloaders noch ein Booten aus systemfremden Quellen notwendig ist, erfüllt ein GRUB2 auch die benötigten Anforderungen. Dennoch ist zu bedenken, dass der Einsatz eines GRUB2 mit deutlich weniger Aufwand in der Entwicklung verbunden sein kann [15].

3 Bootloader im Kontext eingebetteter Systeme

In Kapitel 2.2.2 wurde herausgearbeitet, was der Unterschied zwischen einem Bootloader im Kontext eines Desktop-Computers und dem eines eingebetteten Systems ist. Im Gegensatz zu einem Desktop-Betriebssystem liegt das Hauptaugenmerk bei einem eingebetteten System im Update der Systemanwendung. Im nachfolgenden Kapitel wird nun ein theoretischer Überblick über die Vielzahl an Möglichkeiten geliefert, welche bei der Entwicklung eines Bootloaders zu beachten sind. Dabei werden zunächst die Anforderungen an ein System geprüft und ein Überblick über gängige Funktionen gegeben. Anschließend werden Grundprinzipien vorgestellt, welche den Programmablauf grundlegend vorschreiben. Abschließend werden die gewonnenen Erkenntnisse auf den konkreten Fall einer STM-Architektur in Verbindung mit einem ARM® Cortex-M4® transferiert. In Kapitel 4 wird anschließend die praktische Anwendung der erläuterten Prozesse für den entwickelten Bootloader aufgezeigt.

3.1 Anforderungen

Wenn es zur Softwareentwicklung bei eingebetteten Systemen kommt, gibt es immer eine grundlegende Frage. Welche Hardware wird verwendet und was hat dies für Konsequenzen bei der Entwicklung. Ein Bootloader soll in der Lage sein, die Anwendung zu starten und gegebenenfalls zu aktualisieren. Die grundlegende Anforderung während des Update-Prozesses ist die Frage, wie die Updatedatei zum Bootloader transportiert wird, also welche Schnittstellen im System zur Verfügung stehen oder stehen sollen [8]. Die folgende Liste gibt einen Überblick über gängige Schnittstellen und Protokolle [6, 8, 17]:

- Seriell (USB, RS232, U(S)ART, ...)
- Ethernet (RJ45)
- Platinenintern (SPI, I2C)
- Drahtlos (Bluetooth, WIFI)
- Debug-Schnittstellen (JTAG, SWD)
- Industrielle Standards (CAN, MODBUS, ...)
- Peripheriegeräte (SD-Karte, SSD, Flash-Speicher, ...)

3 Bootloader im Kontext eingebetteter Systeme

Da es eine Vielzahl an unterschiedlichen Schnittstellen und Protokollen gibt, gilt es zu überprüfen, unter welchen Gegebenheiten das System eingesetzt wird. Sollte das System nicht von außen zugänglich sein, so bietet sich der Einsatz einer drahtlosen Schnittstelle an. Falls das System jedoch unzugänglich ist, aber dennoch die Möglichkeit für eine RJ45-Schnittstelle besteht, kann wiederum ein breiteres Spektrum an Protokollen infrage kommen. Bei einem zugänglichen System sind entsprechend alle gelisteten Optionen möglich [8].

In den Auswahlprozess der Schnittstelle muss auch deren softwareseitiger Aufwand mit einfließen. Während eine serielle Schnittstelle einige recht einfache Protokolle anbietet, unterstützt eine Internetschnittstelle wie beispielsweise Ethernet komplexere Protokolle. Je aufwendiger das Protokoll ist, desto größer ist meist auch der Nutzen. Dieser steht jedoch in direktem Konflikt zur Ressourcenbeanspruchung des Protokolls. Hinzu kommt auch, dass Daten bei seriellen Protokollen streng sequenziell am Bootloader ankommen, wohingegen Netzwerkprotokolle diese sequenzieren müssen, um die Reihenfolge der Daten zu gewährleisten [8]. Hier gilt es demnach, Kosten und Nutzen gegeneinander abzuwägen. Bei der Entscheidung bezüglich der Art und Anzahl der Schnittstelle sollte demnach auch deren Komplexität mit einfließen, da der Bootloader möglichst wenig Speicher benötigen darf [6].

Der Anwendungsspeicher des Systems ist ebenfalls ein wichtiger Aspekt, welcher bei der Entwicklung einer Systemsoftware und des zugehörigen Bootloaders eine Rolle spielt. Meist beinhaltet ein eingebettetes System einen Flash-Speicher. Ein solcher Speicher ist in unbeschriebenem (leeren) Zustand vollständig mit binären Einsen gefüllt. Um eine Speicherzelle zu beschreiben, werden die „0“ Bits beschaltet und entsprechend auf null gesetzt. Dies geschieht blockweise, da ein Flashspeicher nicht bitweise beschrieben werden kann. Dies wiederum hat zur Folge, dass nur blockweise gelöscht werden kann [6, 8].

Um ein Programm aus dem Flash auszuführen, benötigt man darüber hinaus einen sogenannten NOR-Flash. Dieser zeichnet sich durch die Fähigkeit eines direkten Lesezugriffs auf die Speicherzellen des Flash-Speichers aus. Der NAND-Flash hingegen ist nicht dazu in der Lage und kann nur blockweise lesen und schreiben [18]. In diesem Flash-Speicher werden nach der Entwicklung mindestens zwei Programme liegen. Dies ist zum einen die Hauptanwendung des Systems, welcher ein Großteil des verfügbaren Speichers zugestanden wird und zum anderen die Bootloaderanwendung, welche möglichst wenig Speicher verbrauchen darf [8]. In der Regel sind diese Speicherbereiche nicht physisch voneinander getrennt, weshalb dies im Entwicklungsprozess berücksichtigt werden muss. Für beide Bereiche müssen daher feste Größen definiert werden [9].

Darüber hinaus gilt es zu verhindern, dass eines der beiden Programme größer als der reservierte Bereich ist. Hierfür gilt es eine strikte Grenze zwischen Bootloader- und Anwendungsbereich zu ziehen [8]. Um eine solche umzusetzen, gibt es mehrere Möglichkeiten. Eine davon ist die Definition der Speicherbereiche im Linker-Skript

der jeweiligen Anwendungen. Näheres hierzu findet sich in Kapitel 5.2. Darüber hinaus muss sichergestellt werden, dass der Bootloader sowohl Lese- als auch Schreibrechte für den Speicherbereich der Anwendung erhält. Nur so kann dieser ein Update durchführen und die Ausführung anstoßen [6].

Zusätzlich gilt es im Entwicklungsprozess zu berücksichtigen, dass die beiden Programme auf dieselbe Hardware zugreifen können. Daher ist die vollständige Ressourcenfreigabe nach Abschluss der Bootloaderoutine enorm wichtig [7]. Ist dies nicht der Fall, kann dies zu unerwünschtem Verhalten aufseiten der Hauptanwendung führen [8]. Umgehen lässt sich dieses Risiko durch einen Neustart des Systems, welcher in der direkten Ausführung des Hauptprogramms resultiert. Dafür muss jedoch die in Kapitel 2.1 beschriebene Branching-Entscheidung außerhalb des Bootloaders stattfinden [6, S. 9f].

Falls die Notwendigkeit besteht, temporäre Informationen zu speichern, bietet sich die Hardwareerweiterung um einen flüchtigen Speicher wie beispielsweise einen Electrically Erasable Programmable Read-Only Memory (EEPROM) an. Dies kann der Fall sein, wenn, wie in Kapitel 2.1 beschrieben, ein selbstupgradefähiger Bootloader im Einsatz ist. In diesem Fall kann das MIT über den flüchtigen Speicher kommuniziert werden. Solch ein zusätzlicher Speicher kann auch den Update-Prozess des Bootloaders unterstützen, indem gelesene Teile des Updates in eben diesem Speicher für die Verifizierung zwischengespeichert werden [7].

Ein weiterer wichtiger Punkt ist das Dateiformat, welches der Bootloader für die Updatedatei akzeptiert. Diese Datei enthält das kompilierte Programm der aktuellsten Version der Hauptanwendung. Im Flashspeicher muss diese Datei abschließend, im Binärformat abgelegt werden. Daher besteht immer die Möglichkeit eben dieses als Dateiformat zu verwenden. Dies bringt den Vorteil mit sich, dass ein Bootloader keine Veränderung in Form von Parsen an der Datei vornehmen muss. Dies erhöht einerseits die Geschwindigkeit des Update-Vorgangs und bringt andererseits eine Ersparnis von unnötigem Programmcode, der wiederum mehr Programmspeicher bedeuten würde. Nachteil des Binärformats ist jedoch die Unleserlichkeit durch den Menschen.

Aus diesem Grund gibt es einige Alternativen zum Binärformat, welche eine bessere Lesbarkeit für den Menschen gewährleisten. Eine dieser Auswahlmöglichkeiten ist das Intel-HEX-Dateiformat [7]. Dieses Format unterteilt eine Binärdatei in Einträge und erweitert jeden von diesen um Information wie Eintragsnummer, Länge, Adresse, Typ und eine Checksumme [19]. Selbstverständlich gibt es noch weitere spezielle Dateiformate, welche wiederum andere Vorteile und Nachteile mit sich bringen. Manche Hersteller von Mikrocontrollern wie beispielsweise Texas-Instruments haben sogar ein ganz eigenes Format, wie in diesem Fall das TI-TXT Format [7, 20].

3.2 Funktionsübersicht

In Kapitel 2.1 wird auf die Basisfunktionen eines Bootloaders aufmerksam gemacht. Ein solcher muss mindestens dazu in der Lage sein, den Speicher des Systems zu löschen, zu beschreiben und in irgendeiner Weise die geschriebene Anwendung zu starten [6]. Da eine Anforderung an den Bootloader jedoch darin liegt, dass dieser die Updatedatei überprüft, kann dieses Basisset um einige Funktionen erweitert werden.

Natürlich gibt es viele Ansätze und Sonderfälle, welche bei der Entwicklung von Bootloadern entscheidend sind. Aus diesem Grund ist es schwierig, einen festen Ablauf für alle Bootloader zu bestimmen. Im Bezug auf besonders wichtige Funktionen könnte ein typischer Programmfluss eines Bootloaders jedoch in etwa wie in Abbildung 3.1 aussehen. In diesem beispielhaften Ablauf lassen sich bereits zwei Funktionen erkennen, die ein Bootloader zur Fehlererkennung und Behandlung beinhalten sollte. So kann es sehr nützlich sein, den Speicher der Anwendung zu validieren. Mit Hilfe einer Validierung kann nach Start des Bootloaders erkannt werden, ob sich eine Applikation im Speicher befindet. Nur in diesem Fall darf überhaupt ein Sprung zu deren Startadresse stattfinden [6]. Wenn der Branching-Prozess außerhalb des Bootloaders gelagert ist, muss dieser Schritt bereits hier durchgeführt werden. Eine reine Überprüfung eines GPIO-Pins kann dies jedoch nicht erfüllen. Ist dies nicht möglich, bietet es sich an, den Branch in den Bootloader zu integrieren [6].

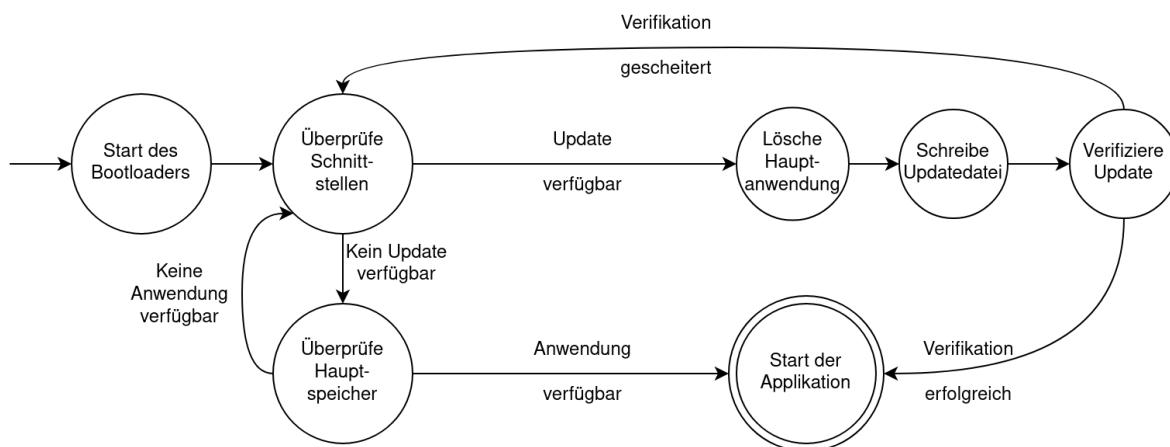


Abbildung 3.1 Beispielhafter Ablauf eines Bootloaders

Quelle: Eigene Darstellung in Anlehnung an [6, 8]

Zusätzlich zur Validierung gilt es, den Speicherinhalt auch auf dessen Fehlerfreiheit zu überprüfen. Diese Verifizierung kann einerseits in Verbindung mit der Validierung des Speichers während des Branching-Prozesses durchgeführt werden. Dadurch lässt sich ein korrupter Speicherinhalt vor einem möglichen Sprung in die Anwendung erkennen. Andererseits muss nach einem Update eine Verifizierung des Geschriebenen erfolgen, um sicherzustellen, dass das Update korrekt installiert wurde [6, 8]. Eine

solche Verifizierung lässt sich entweder mit einem Cyclic Redundancy Check (CRC) oder einer Signatur bewerkstelligen. Ein praktischer Nebeneffekt dieser beiden Methoden ist, dass der Hersteller hier eine gewisse Sicherheit hat, die Integrität der Updatedatei zu gewährleisten. Dies ist möglich, falls dieser einen eigenen Berechnungsalgorithmus für den CRC verwendet oder entsprechend die Signaturbildung nur dem Hersteller bekannt ist [8]. Sollte die Verifizierung scheitern, muss ein entsprechender Zustand für den Bootloader definiert werden. Im konkreten Fall ist dies ein Bereinigen des Anwendungsspeichers und das Warten auf ein gültiges Update. Falls das System über genügend Random Access Memory (RAM) verfügt oder ein EEPROM mit ausreichender Kapazität vorhanden ist, kann eine Verifizierung der Updatedatei sogar vor der Installation stattfinden und die Überschreibung einer funktionalen Software im Speicher somit verhindert werden [8]. Auch wenn diese Optionen nicht verfügbar sind, muss versucht werden, ein Überschreiben von funktioneller durch fehlerhafte Software zu vermeiden.

Ein weiterer wichtiger Aspekt ist die Versionierung der Updatedatei [9]. Dadurch kann die Verwendung veralteter Updates oder gar des falschen Updates verhindert werden, indem der Bootloader die Updatedatei auf deren Version überprüft und diese mit der aktuell im Speicher befindlichen vergleicht. Hier gilt es, die Entscheidung zu fällen, ob nur Upgrades oder auch Downgrades der Software möglich sein sollen. In jedem Fall hilft dies auch bei der Identifizierung der installierten Anwendungsversion und lässt Rückschlüsse über die Notwendigkeit eines Updates zu.

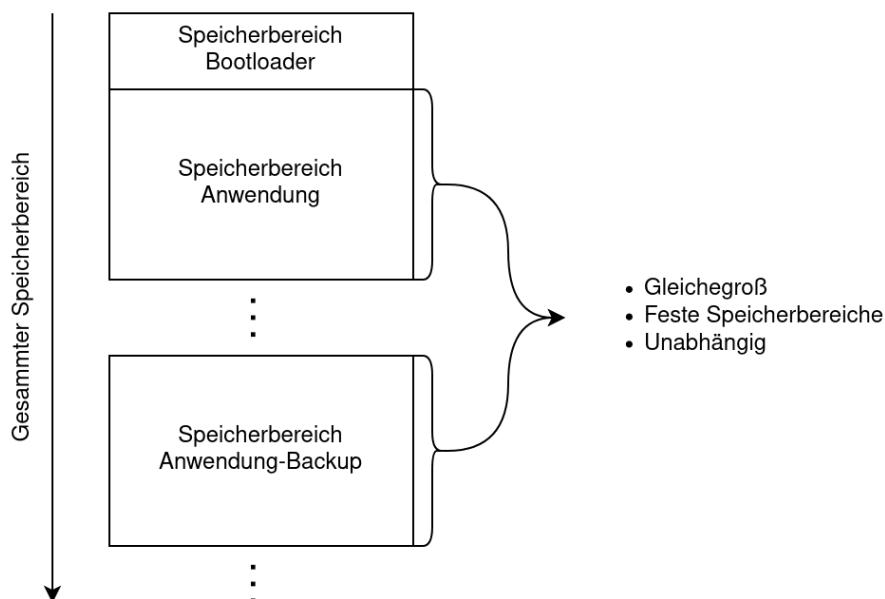


Abbildung 3.2 Speicherübersicht Backupoption
 Quelle: Eigene Darstellung in Anlehnung an [9]

3 Bootloader im Kontext eingebetteter Systeme

Um die funktionelle Sicherheit der Anwendung beim Auftreten eines Fehlers während des Update-Prozesses zu gewährleisten, bietet sich die Verwendung eines Backup-Mechanismus an. In diesem Fall wird eine zusätzliche Kopie der Hauptanwendung im Speicher abgelegt. Abbildung 3.2 zeigt, wie ein derartiger Mechanismus durch eine weitere Unterteilung des Speicherbereichs realisiert werden kann. Eine Grundvoraussetzung dafür ist eine angemessene Speichergröße. Wenn diese gegeben ist, sollte ein zweiter Speicherbereich für das Backup festgelegt werden, welcher der Größe des Anwendungsbereichs entspricht. Auch für diesen Bereich muss eine feste Adresse definiert werden und dieser muss, wie in Kapitel 3.1 für Bootloader und Hauptanwendung beschrieben, strikt von den anderen getrennt werden. Darüber hinaus muss sichergestellt werden, dass auf den Inhalt des Backup-Bereiches nur über den Bootloader zugegriffen wird. Dieser muss ebenfalls verifiziert und validiert werden. Für den Fall, dass die Hauptanwendung fehlerhaft ist, kann das System mit Hilfe des Backups selbst reagieren und die Funktionalität wiederherstellen. Dies führt zu einer Steigerung der Systemzuverlässigkeit [9]. In Kapitel 3.3 wird die Funktion des Flash-Bankings vorgestellt. Diese beinhaltet bereits implizit einen Backup-Mechanismus. Der große Vorteil dieses Mechanismus liegt darin, dass dieses Backup nicht in die Hauptanwendung kopiert werden muss [8].

3.3 Grundlegende Architekturentscheidungen

In den Kapiteln 2.2.1 und 2.2.2 werden die beiden großen Anwendungsbereiche für Bootloader vorgestellt. Die Bootloader, welche in diesen Gebieten Verwendung finden, unterscheiden sich in einigen Punkten. Einer davon ist die zugrunde liegende Architektur. In Kapitel 2.2.1 wird beschrieben, wie der Bootprozess bei Desktop-Betriebssystemen geregelt ist. Dabei wird auf die Option des Multiboots verwiesen, die erlaubt, zwischen mehreren installierten Betriebssystemen zu wählen.

Im Bereich der eingebetteten Systeme stehen hingegen andere Aspekte im Vordergrund. Hier gilt es, die Auswahl der Bootloader-Funktionen anhand der verwendeten Hardware zu treffen. Eine grundlegende Entscheidung spielt dabei, ob das sogenannte Flash-Banking möglich ist oder ob ein monolithischer Bootloader verwendet werden muss [8]. Diese Entscheidung hängt vor allem von dem verwendeten Mikrocontroller [21], dem darin verbauten Prozessor [8] und der Größe des Flashspeichers [9] ab.

Bei manchen Mikrocontrollern findet sich die Unterteilung des Speichers in mehrere sogenannte Bänke. Abbildung 3.3 zeigt, wie eine derartige Speicherunterteilung im Fall eines 2 MiB großen Flash-Speichers aussehen kann. Dabei ist zu erkennen, dass der Speicher in Sektoren unterteilt ist. Diese sind den jeweiligen Bänken zugeordnet und sind in beiden Fällen identisch unterteilt [17]. Die Besonderheit des Flash-Bankings liegt in der Zuweisung der Startadresse für die entsprechende Bank. So sind beide mit der niedrigsten Adresse des Flash-Speichers verlinkt. Die Zuweisung der Reihenfolge, in welcher die Bänke zur Laufzeit im Speicher liegen, wird

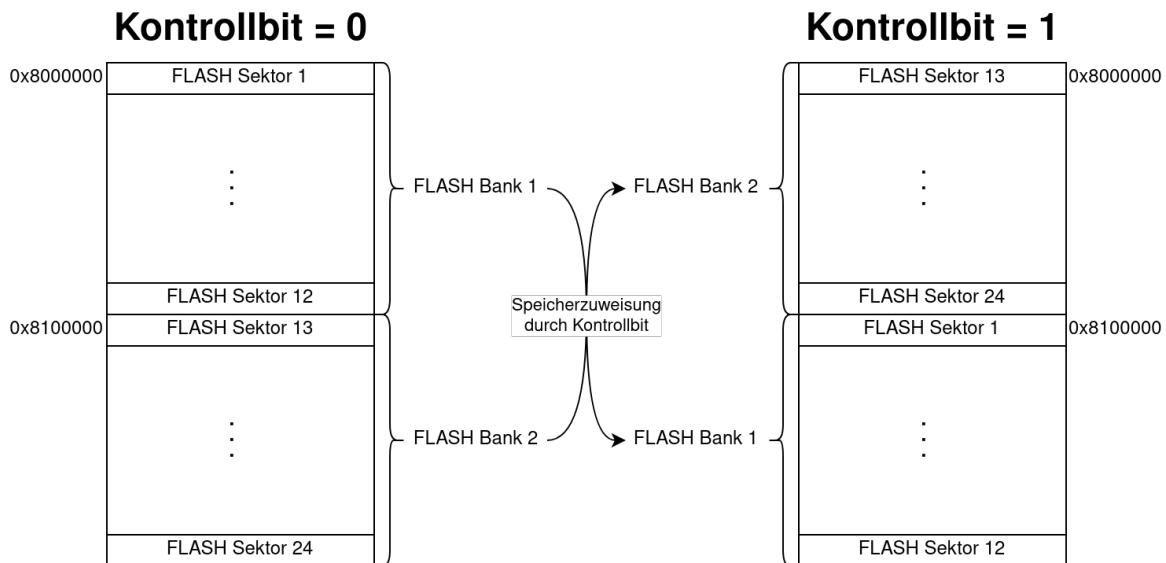


Abbildung 3.3 Speicherzuweisung Flash-Banking
Quelle: Eigene Darstellung in Anlehnung an [21]

über ein Kontrollbit getätigt. Dieses wird zu Systemstart ausgewertet. Die entsprechende Bank wird anschließend auf die Adresse 0x0 assoziiert und ein Start der Anwendung in dieser Bank wird angestoßen. Die zweite Bank wird entsprechend dem freien Speicherbereich zugewiesen. Um die Bank zu wechseln, muss lediglich das Kontrollbit, welches im Nutzercode verändert werden kann, negiert werden. Durch einen Neustart ergibt dessen Auswertung einen Tausch der Speicherzuweisungen für die Bänke [21]. Dieser implizite Backup-Mechanismus bietet den Vorteil, dass ein Backup nicht in den Anwendungsbereich kopiert werden muss. Zusätzlich kann ein Update in den Bereich geladen werden, der das aktuelle Backup enthält. So wird der aktuell funktionale Programmcode nicht angefasst und erst wenn das Update verifiziert ist, werden die Bänke umgeschaltet. Im Fall des Flash-Bankings wird der Update-Mechanismus des Bootloaders in die Controllerfirmware integriert, sodass er nicht im Flash-Speicher liegt [8]. Dies führt allerdings zu Einschränkungen bezüglich der Größe des Bootloaders und daraus resultierend in einem eingeschränkten Funktionsumfang.

Eine Verwendung des Flash-Bankings ist jedoch nicht immer möglich oder die beste Lösung für den Anwendungsfall. Dies ist unter anderem der Fall, wenn die verwendete Hardware nicht über eine derartige Funktion verfügt oder der Funktionsumfang des Bootloaders einen zu großen Speicherbedarf für die Controllerfirmware benötigt. In diesem Fall bietet sich die Implementierung eines monolithischen Bootloaders an [8].

Der monolithische Bootloader wird wie eine herkömmliche Anwendung implementiert. Er wird an den Anfang des Flash-Speichers geladen und direkt nach Systemstart ausgeführt. Für den Flash-Speicher des Mikrocontrollers wird, wie in

3 Bootloader im Kontext eingebetteter Systeme

Abbildung 3.4 zu sehen, eine strikte Unterteilung in mindestens zwei Bereiche vorgenommen. Auf den reservierten Abschnitt des Bootloaders darf nur von diesem im Zuge der Programmausführung zugegriffen werden. Jegliche Updates dürfen ausschließlich in den für die Anwendung vorgesehenen Bereich geschrieben werden. Dies ist wichtig, da ein fehlerhafter Bootloader zu einem inoperablen System führen kann. Eine korrupte Anwendung kann jedoch durch ein erneutes Update durch den Bootloader gerettet werden [8].

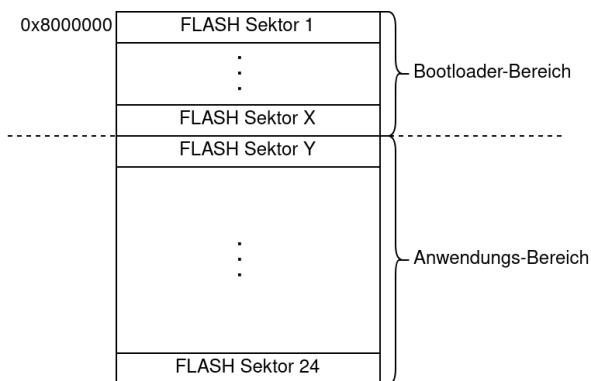


Abbildung 3.4 Speicherunterteilung monolithischer Bootloader

Quelle: Eigene Darstellung in Anlehnung an [8]

Ein monolithischer Bootloader wird häufig als Zustandsautomat implementiert [9]. Da dieser nicht auf Nutzereingaben warten muss, werden die jeweiligen Zustandsänderungen automatisch durchlaufen. Befindet sich keine Hauptanwendung im Speicher, wartet der Bootloader auf eine über eine der konfigurierten Schnittstellen verfügbare Updatedatei [8].

Eine weitere wichtige Architekturentscheidung liegt darin, ob der Bootloader selbst upgradefähig sein muss. Dies ist ein riskanter Vorgang, welcher im Fehlerfall das System korrumpern kann [8]. Da bei der Entwicklung von Software jedoch häufig Fehler auftreten können, welche erst viel später auffällig werden, lässt sich ein derartiger Mechanismus oft nicht vermeiden. So kann man davon ausgehen, dass bei einem Codefragment von 1000 Zeilen bis zu 25 Fehler enthalten sind [22]. Während der Laufzeit eines sich im Einsatz befindlichen Systems besteht daher die Möglichkeit, dass Fehler im Bootloader-Code auffällig werden. Um jedoch einen Rückruf der Produktivsysteme zu verhindern, schafft ein Upgrade-Mechanismus des Bootloaders hier Abhilfe. Für einen solchen müssen entsprechende Absicherungen vorgenommen werden, um auf einen fehlerhaften Upgrade-Prozess reagieren zu können [8]. In Kapitel 6 wird erläutert, wie ein solcher Update-Mechanismus Anwendung findet und welche Überlegungen für dessen Implementierung berücksichtigt wurden.

3.4 Anwendung dieser Konzepte in der STM32-Architektur

Der im Zuge dieser Arbeit entwickelte Bootloader wurde auf Basis eines Mikrocontrollers der Firma STMicroelectronics (STM) entwickelt. Aus diesem Grund wird in folgendem Kapitel ein Überblick über die Umsetzung der Anforderungen (Kapitel 3.1), Funktionen (Kapitel 3.2) und Architekturen (Kapitel 3.3) seitens STM gegeben. Im Kern der Betrachtung stehen dabei Systeme, die für die Verwendung eines Bootloaders ausgelegt sind und einen ARM® Cortex-M4®-Prozessor beinhalten.

Bezüglich der Anforderungen zur Entwicklung eines Bootloaders wurden in Kapitel 3.1 die Punkte Schnittstelle, (flüchtige) Speicher und der Auswahlprozess des Dateiformates betrachtet. STM bietet für die Programmierung und das Debuggen der hier relevanten Systeme die beiden Schnittstellen Joint Test Action Group (JTAG) und Serial Wire Debug (SWD) an [17, 23]. Diese werden dazu verwendet, den kompilierten Code in Form eines binären Formates in einen Speicher des eingebetteten Systems zu laden [24]. Darüber hinaus werden je nach System mehrere der in Kapitel 3.1 genannten Schnittstellen unterstützt [17, 23]. Die Größe des Flash-Speichers und die Verfügbarkeit eines EEPROM ist dabei jedoch systemabhängig und muss bei der Auswahl der Mikrocontroller-Einheit (MCU) berücksichtigt werden.

Die in Kapitel 3.2 beschriebenen Funktionen spielen ebenfalls eine Rolle für den Auswahlprozess. Während sich ein CRC ohne Hardwareunterstützung implementieren lässt, bietet STM dennoch häufig eine CRC-Berechnungseinheit zur Unterstützung an [17, 23]. Diese unterstützt die Verifizierung zur Laufzeit und bringt daher Vorteile gegenüber einer eigenen Implementierung aufgrund der Hardwareunterstützung mit sich [23].

Die Größe des Flash-Speichers gilt es anhand der geplanten Architektur auszuwählen. Im Falle eines monolithischen Bootloaders ohne Backup-Funktion muss dieser mindestens die kombinierte Größe aus Bootloader und Anwendung beinhalten. Für das Flash-Banking bietet STM aktuell drei Serien an. Die STM32G4 und STM32L4 Serien enthalten dabei den ARM® Cortex-M4®-Prozessor. Die dritte Serie STM32L0 mit einem ARM® Cortex-M0®-Prozessor wird ebenfalls angeboten [21]. STM bietet bezüglich der Speichergrößen viele verschiedene Produkte an. Sollte dennoch der Fall einer unzureichenden Speicherkapazität eintreten, bietet sich die Möglichkeit einer externen Speichererweiterung. Daher gibt es auch genügend Möglichkeiten zur Anwendung eines monolithischen Bootloaders mit Backup-Funktion.

4 Aufbau und Architektur

Abbildung 4.1 zeigt den grundlegenden Aufbau des Systems. Dabei werden die verschiedenen Systemkomponenten und -schnittstellen gezeigt. Hierbei handelt es sich um eine Auflistung aller festen und optionalen Bestandteile, welche das System enthalten kann. Die Zusammenstellung der Komponenten für den jeweiligen Anwendungsfall ist Teil des Kompilierungsprozesses. Als fester Bestandteil des Bootloaders ist die serielle Schnittstelle für Aktualisierungen festgelegt. Zusätzlich muss die Planung der Hardware mindestens die dargestellten Leuchtdioden (LED) für das Nutzerfeedback beinhalten. Auch die externe CRC-Recheneinheit und der Flash-Speicher müssen enthalten sein. Nur die Ethernet- und Liquid Crystal Display (LCD)-Schnittstelle sind optional.

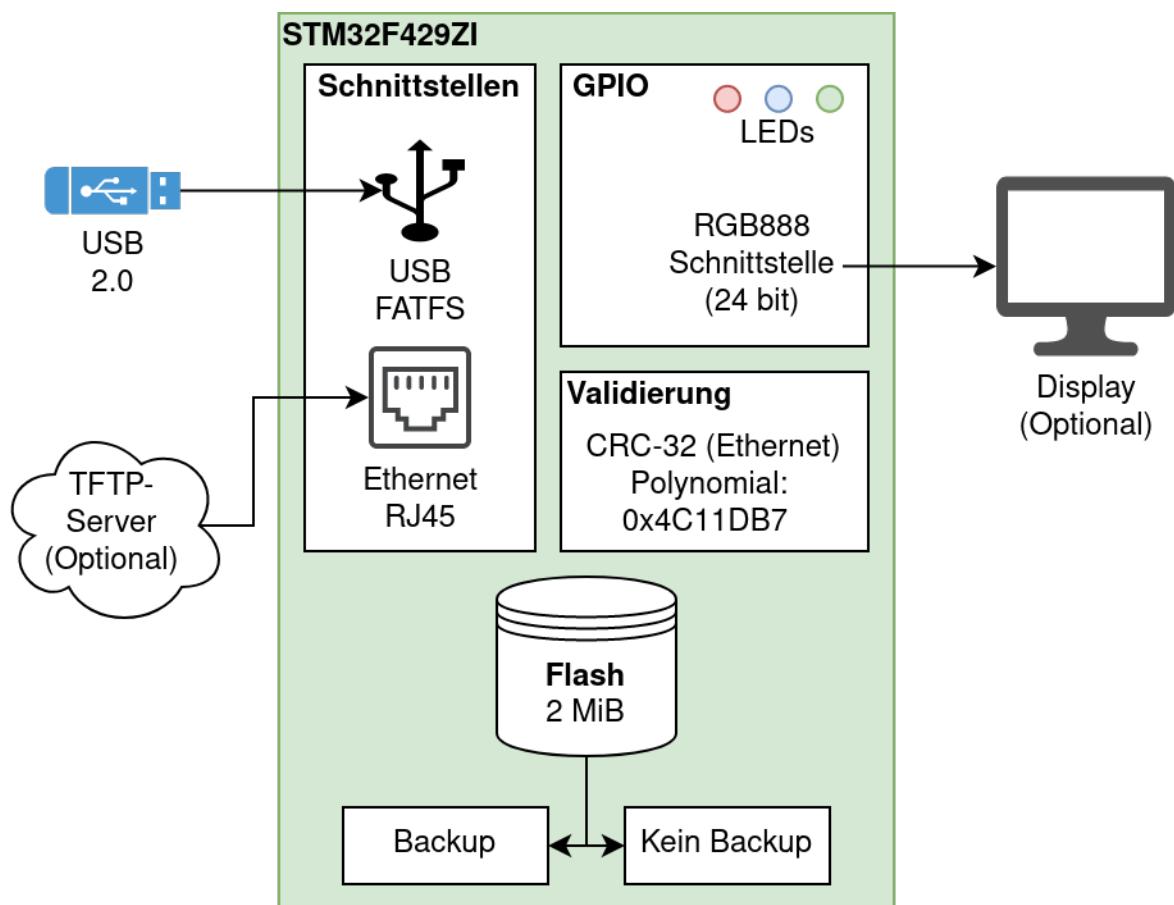


Abbildung 4.1 Architekturübersicht des Zielsystems
Quelle: Eigene Darstellung in Anlehnung an [17]

4 Aufbau und Architektur

Dieses Kapitel liefert einen Überblick über die dargestellten Komponenten. Dabei wird auf die festen und optionalen Systemkonfigurationen eingegangen und der Entscheidungsprozess näher erläutert. Zusätzlich wird auf die zu berücksichtigenden Maßnahmen verwiesen, welche den Auswahlprozess für die Portierung auf ein anderes System beeinflussen.

4.1 Eingebettetes System

Den Kern des Systems stellt das eingebettete System dar. Dieses beinhaltet sowohl den Bootloader als auch die durch diesen zu ladende Anwendung. In dieser Arbeit werden lediglich die notwendigen Komponenten für den Bootloader erläutert. Da dieser in der Lage sein muss, unabhängig von der Applikation diese zu laden, kann kein Schluss auf die Hardwareanforderungen der Hauptanwendung getroffen werden.

Für die Anwendung des Bootloaders werden mehrere Komponenten-Kategorien benötigt. Grundlegend muss die Auswahl des Speichers und dessen Größe beachtet werden. Darüber hinaus werden Schnittstellen benötigt, um die Applikation in den Speicher des Systems zu transferieren. Ein Feedback-Mechanismus zum Status des Updates erleichtert zusätzlich die Nutzerfreundlichkeit der Anwendung. Um die Stabilität und Sicherheit des Update-Mechanismus zu gewährleisten, sollte darüber hinaus auch eine Verifizierung möglich sein.

Die nachfolgenden Kapitel befassen sich mit eben diesem Auswahlprozess. Dabei wird besonders auf die Wahl der Hardware, die unterstützten Schnittstellen und die für den Bootloader notwendigen Komponenten eingegangen.

4.1.1 Auswahl der Hardware

Abbildung 4.1 zeigt die Hardwareübersicht des Bootloaders für die NUCLEO-F429ZI MCU [25]. Dieses System wird aufgrund der Anforderungen durch die eigentliche Anwendung des Systems benötigt. Die Entwicklung des Bootloaders ist daher an diese Hardware gebunden. Da der Bootloader jedoch portierbar sein soll, wird im Folgenden gezeigt, welche Einschränkungen der Bootloader bei der Hardware hat und welche Komponenten verwendet werden. Die dargestellten Informationen entstammen dem offiziellen Datenblatt der Firma STM [25].

STM bietet für die NUCLEO-F429xx Baureihe verschiedene Komponenten zur Auswahl. Das NUCLEO-F429ZI Board wie in Abbildung 4.2 zu sehen ist dabei die Variante mit dem größten Flash-Speicher in Höhe von 2 MiB. Der im System enthaltene ARM® Cortex-M4®-Prozessor ermöglicht die direkte Ausführung von Programmen aus diesem Speicher. Dadurch kann sowohl eine Hauptanwendung als auch der Bootloader aus dem Flash-Speicher betrieben werden. Das System verfügt darüber hinaus über eine Memory Protection Unit (MPU). Diese ermöglicht einen sicheren

4.1 Eingebettetes System

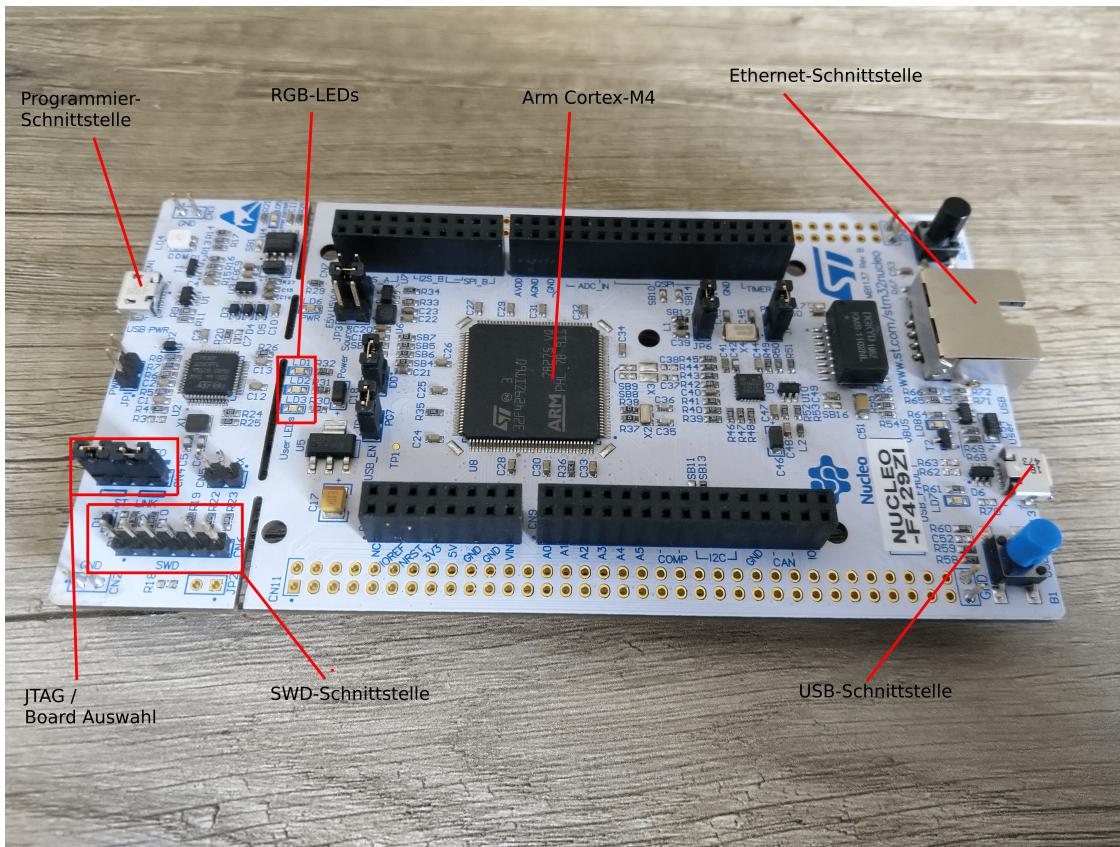


Abbildung 4.2 NUCLEO-F429ZI

Quelle: Eigene Aufnahme

Zugriff auf Speicherinhalte und verhindert deren Korruption. Der Prozessor kann mit einer Taktfrequenz von 168 MHz auch eine schnelle Ausführung der jeweiligen Anwendungen gewährleisten.

In Abbildung 4.2 sind die wichtigsten System-Komponenten hervorgehoben. Hier wird ersichtlich, welche Möglichkeiten das Board für die Bootloader-Implementierung mit sich bringt und welche Vorteile sich daraus ergeben. So verfügt das Board neben dem genannten Prozessor bereits über drei RGB-LEDs, eine Ethernet- und USB-Schnittstelle sowie eine Programmier-Schnittstelle.

Die Programmier-Schnittstelle der MCU umfasst den linken Teil des Boards, welcher durch eine Bruchstelle vom rechten Part getrennt ist. Mit Hilfe von Jumpern kann entweder der rechte Teil mit der Programmier-Schnittstelle verbunden werden oder ein an die SWD-Schnittstelle angeschlossenes System. Sollten die Jumper geschlossen sein, ist der integrierte ST-Link für ein externes System nicht aktiviert und die rechte Boardseite aktiv. Die Programmier-Schnittstelle wird zum Transfer des Bootloader-Programms in den Speicher des Zielsystems genutzt. Des Weiteren unterstützt die Schnittstelle durch JTAG und SWD ein Debuggen dieser Anwendung.

4 Aufbau und Architektur

Das Board verfügt über drei RGB-LEDs. Diese werden im Bootloader verwendet, um ein visuelles Feedback bezüglich des Fortschrittes zu geben. Wie in Abbildung 4.1 ersichtlich, werden für andere Systeme ebenfalls drei RGB-LEDs vorausgesetzt. Näheres zur Fortschrittsanzeige der LEDs findet sich in Kapitel 5.5.

Ein zusätzlicher Feedback-Mechanismus ist durch eine Fortschrittsanzeige auf einem angeschlossenen Display möglich. Dies ist eine optionale Erweiterung für Systeme, die ohne Sicht auf die LEDs fest verbaut sind und über ein Display verfügen. Für die Realisierung wird die 24 Bit parallele digitale RGB-Schnittstelle verwendet. Diese ist für eine Fortschrittsanzeige vollkommen ausreichend, da Visualisierung und Farbwechsel möglich sind. Diese Option lässt sich im Programmcode des Bootloaders vor der Kompilierung aktivieren. Näheres zu den Konfigurationsoptionen findet sich in den Kapiteln 5.5 und 5.6.

Bezüglich der Schnittstellen zum Dateitransfer enthält das System mehrere Optionen. Unter diesen befinden sich Ethernet und USB. Zur Bereitstellung der Anwendungsdatei bieten sich diese beiden Protokolle besonders an, da durch USB eine direkte Schnittstelle am System verfügbar ist. Ethernet hingegen ermöglicht ein Update der Anwendung aus der Ferne. In Abbildung 4.1 ist zu erkennen, dass Ethernet eine optionale Schnittstelle ist. Diese Option lässt sich ebenfalls in der Konfiguration des Bootloaders vor der Kompilierung einstellen. Daraus ergibt sich die Notwendigkeit einer USB-Schnittstelle für die Hardware. Ein RJ45-Ethernet Steckplatz muss jedoch nicht zwingend in das System integriert werden. Eine genauere Beschreibung der Anforderungen für die Schnittstellen findet sich in Kapitel 4.1.3.

Um sowohl den Flash-Speicherinhalt als auch über Schnittstellen erhaltene Dateien auf Korrektheit zu verifizieren, verfügt das System zusätzlich über eine CRC-Berechnungseinheit. Diese ist in der Lage, aus einem Generator-Polynom und einem 32 Bit Speicherinhalt einen CRC-Code zu generieren. Damit kann ein mitgelieferter CRC-Wert verifiziert werden.

Der Auswahlprozess der Hardware ist sehr an die Anforderungen der Hauptanwendung geknüpft. Der Bootloader muss mit den genannten Hardware-Komponenten auskommen, da das Hauptsystem auf diesen aufbaut. Die beschriebene Hardware ist jedoch für einen Bootloader ausreichend, da ein solcher möglichst ressourcenschonend und mit minimalen Anforderungen ausgestattet sein muss. Dennoch liefert die verfügbare Hardware einige Möglichkeiten zur Erweiterung des Bootloaders. Der Speicher mit einer Größe von 2 MiB bietet darüber hinaus die Möglichkeit eines Backups der Hauptanwendung, falls diese eine entsprechende Größe aufweist. Da das System jedoch über keinen Dual-Bank-Modus verfügt, muss hier eine eigene Speicherunterteilung und ein entsprechender Backup-Mechanismus implementiert werden. Falls die Hauptanwendung zu groß ist oder der Speicher des Zielsystems kleiner als 2 MiB ist, bietet sich die Verwendung eines Backup-Mechanismus nicht an. Die Anwendung und Implementierung des Backup-Mechanismus und dessen Auswahlprozess sind in Kapitel 5.2 näher beschrieben.

4.1.2 ARM® Cortex-M4®-Prozessoreigenschaften

Eine zentrale Anforderung an die Hardware-Architektur, welche sich aus der Hauptanwendung des Zielsystems ergibt, ist der ARM® Cortex-M4®-Mikroprozessor. Daraus resultierend muss auch der Bootloader spezifisch für diesen Prozessor entwickelt werden. Dies ist eine enorme Einschränkung bezüglich der Portierbarkeit des Bootloaders auf andere eingebettete Systeme. Die Festlegung auf einen Prozessor ist jedoch ein unvermeidlicher Teil der Architekturstimmung. Da der ARM® Cortex-M4® innerhalb der Cortex-M®-Familie einer der neusten Mikroprozessoren ist, stellt dieser einen soliden Funktionsumfang für industrielle Anwendungen bereit und kann somit für ein breites Spektrum an Anwendungsfällen verwendet werden.

Der Mikroprozessor basiert auf einer 32 Bit-Architektur. Dabei unterstützt dieser 8, 16 und 32 Bit-Daten [26]. Im Programm können auch 64 Bit-Daten verwendet werden. Diese werden jedoch in zwei 32 Bit-Register unterteilt und bedeuten mehr Rechenaufwand bei der Verarbeitung. Der Prozessor unterstützt darüber hinaus die Anbindung von externen Flash-Read only Memory (ROM)-Speichern und statischem RAM [27]. Dies kann für die Bootloader-Entwicklung sehr praktisch sein, da die Hauptanwendung einige Daten wie beispielsweise den Display-Speicher in externe Speichermedien auslagern kann. Zusätzlich verfügt der Mikroprozessor über einen Nested Vectored Interrupt Controller (NVIC). Dieser ermöglicht eine Gesamtzahl von 240 Exceptions und Interrupts. Diese können zusätzlich priorisiert werden, um komplexere Systeme zu erlauben. Da ein Bootloader mit möglichst wenigen Interrupts auskommen muss und diese bei seiner Beendigung darüber hinaus wieder frei gibt [7], ist der ARM® Cortex-M4® hier mehr als ausreichend ausgestattet, um neben dem Bootloader auch noch eine komplexe Anwendung zu betreiben.

Die Einschränkung bezüglich der Portierbarkeit, welche sich durch die ausschließliche Verwendung eines ARM® Cortex-M4®-Mikroprozessors ergibt, ist aus diesen Gründen verkraftbar. Dieser Prozessor bietet ein weites Spektrum an Einsatzszenarien im industriellen Umfeld. Somit kann auch der Bootloader für all diese Anwendungen verwendet werden.

4.1.3 Schnittstellen zum Dateitransfer

Ein wichtiger Aspekt für das Design des Bootloaders liegt in der Definition von Schnittstellen. Diese werden benötigt, um die Daten der Anwendungsdatei zum System zu transportieren. In Kapitel 4.1.1 wurde auf die Verwendung der USB- und Ethernet-Schnittstellen hingewiesen. Diese sind beide am Mikrocontroller aus Abbildung 4.2 enthalten. Da die Hauptanwendungsgebiete für den Bootloader einen industriellen Hintergrund aufweisen, bieten sich diese beiden Schnittstellen auch aus nachfolgenden Gründen sehr gut an.

Ein weiterer Faktor für ein mögliches Update der Hauptanwendung ist die Zugänglichkeit der Hardware. Da es üblich ist, eingebettete Systeme in größere Anlagen zu

4 Aufbau und Architektur

integrieren, kann es vorkommen, dass ein direkter Systemzugriff sehr aufwendig ist. Aus diesem Grund stellt die Bereitstellung von sowohl einer direkten als auch einer remote Schnittstelle eine essenzielle Anforderung dar. Da jedoch jede zusätzliche Schnittstelle zusätzlichen Speicherbedarf für die Bootloaderanwendung bedeutet, gilt es, deren Anzahl möglichst gering zu halten.

Um also die beiden Aspekte Zugänglichkeit und geringe Anwendungsgröße zu vereinen, bietet dieses System die remote Ethernet-Schnittstelle entgegen der USB-Schnittstelle lediglich als Option an. Da ein Ethernet-Protokoll mit sehr viel Overhead wie beispielsweise IP-Adressen in der Kommunikation ausgestattet ist, muss dieses in der Programmierung verarbeitet werden. Da dies in signifikant größerem Programmbedarf resultiert, stellt die Ethernet-Schnittstelle eine aufwendigere Variante dar. Darüber hinaus muss für die Anbindung der Schnittstelle auch deren Hardwarebedarf betrachtet werden. Da ein Ethernet-Port recht klobig und ein Kabel durch dessen Schirmung ebenfalls Platz benötigt, kann die Verwendung dieser Schnittstelle auch zu Platzproblemen innerhalb des Systems oder bereits auf der Platine führen.

Um jedoch den Programmbedarf für die Verwendung der Ethernet-Schnittstelle möglichst gering zu halten, spielt die Auswahl des Kommunikationsprotokolls eine wichtige Rolle, da es hier eine Vielzahl an verschiedenen Ansätzen zur Datenübertragung über das Internet gibt. Da es sich in diesem Anwendungsfall um eine einseitige Datenübertragung handelt, bietet sich die Verwendung eines „File Transfer“ Protokolls an. Hierfür eignen sich namensgebend die Varianten File Transfer Protocol (FTP) und TFTP. Das FTP-Protokoll ist das verbreitetste Protokoll mit dem Zweck des Datentransfers [28]. Dabei bietet es ein breites Spektrum an Kommandos und Funktionen zur sicheren Datenübertragung an. Um dies zu bewerkstelligen, nutzt FTP Transmission Control Protocol (TCP). Neben der Notwendigkeit einer festen Verbindung wird zusätzlich eine Authentifizierung für die Kommunikation benötigt [29]. All diese Aspekte führen zu einer hohen Komplexität des Protokolls, welche wiederum zu einem hohen Programmbedarf im Applikation-Layer der Anwendung führen. Als bessere Alternative bietet sich in diesem Fall das TFTP an. Dieses kommuniziert über das verbindungslose User Datagram Protocol (UDP). Darüber hinaus verfügt es nur über insgesamt fünf Kommandos, um Daten zu transferieren [30]. Diese niedrige Komplexität wiederum ist für den Anwendungsfall hervorragend geeignet, da dadurch die Größe der Bootloaderanwendung im Rahmen der vorgesehenen Speichergröße bleibt.

Im Hinblick auf die genannten Aspekte bietet eine USB-Schnittstelle entsprechende Vorteile. Zum einen ist ein serielles Protokoll für die Kommunikation mit einem USB-Speichermedium einfacher als ein Ethernet-Protokoll. Dies resultiert entsprechend in geringerem Programmaufwand zur Verarbeitung der Daten. Zum anderen kann ein USB-Steckplatz mit weniger Platzbedarf eingeplant werden. Zusätzlich muss dieser nicht dauerhaft mit einem Speichermedium belegt sein, sondern nur während des Update-Prozesses.

Für die Planung des Systems ist dementsprechend immer eine USB-Schnittstelle für den Update-Prozess vorgesehen. Diese kann Binärdateien von einem angeschlossenen Speichermedium lesen. Falls die Notwendigkeit eines Remote-Updates aus diversen Gründen besteht, verfügt die Architektur der Software über eine Ethernet-Unterstützung. Diese muss jedoch bei der Konfiguration der optionalen Systemkomponenten berücksichtigt werden. Darüber hinaus sind über die Ethernet-Schnittstelle lediglich Anwendungsupdates vorgesehen. Näheres zu den Einschränkungen der Schnittstellen findet sich in Kapitel 5.4.

4.2 Trivial File Transfer Protokoll (TFTP)-Server

In Kapitel 4.1.3 wurde der Auswahlprozess für das TFTP-Protokoll erläutert. Um dieses oder andere Ethernet-Protokolle anwenden zu können, wird zusätzliche Hardware und Infrastruktur benötigt, welche nicht direkt am eingebetteten System verbaut ist. Dabei spielen zwei Aspekte eine wichtige Rolle in der Planung. Zum einen die Verbindung mit einem lokalen Netzwerk oder dem Internet. Zum anderen die Serverstruktur, welche die Updatedatei bereitstellt und das TFTP-Protokoll unterstützt.

In den nachfolgenden Kapiteln wird der Auswahlprozess bezüglich der benötigten Hardware näher betrachtet. Dabei liegt ein Schwerpunkt im Bedarf und was bei der Planung des Netzwerkes zu beachten ist. Im weiteren Schritt wird das TFTP-Protokoll erläutert und aufgezeigt, was dieses für das Bootloader-Update auszeichnet.

4.2.1 Auswahl der Hardware

Um ein Update der Anwendung durch den Bootloader durchzuführen, kann eine Ethernet-Schnittstelle verwendet werden. Abbildung 4.3 zeigt, welche infrastrukturellen Anforderungen dies mit sich bringt und die alternativen Zugriffsmethoden auf den Server. Grundlegend werden neben dem eingebetteten System noch zwei weitere

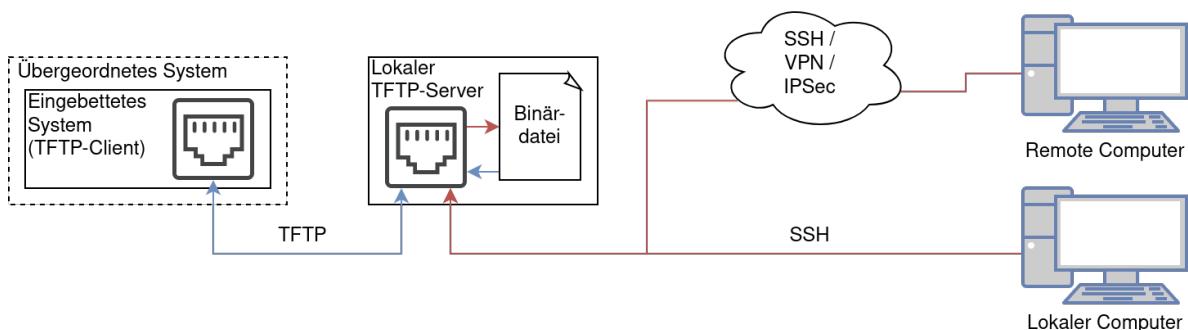


Abbildung 4.3 Trivial File Transfer Protokoll (TFTP) Infrastruktur
Quelle: Eigene Darstellung

4 Aufbau und Architektur

Systeme benötigt. Dieses ist zum einen ein Server, welcher das TFTP-Protokoll unterstützt und zum anderen ein Desktop-Computer der die Updatedatei auf dem Server bereitstellt.

Um die Updatedatei zur Serverinfrastruktur zu befördern, gibt es diverse Möglichkeiten. Da dieser Zugriff nicht via TFTP getätigt wird, sind hier sämtliche Protokolle mit entsprechenden Funktionen verfügbar. An dieser Stelle gibt es nicht einmal die Einschränkung einer kabelgebundenen Verbindung, auch drahtlose Kommunikation ist möglich. Natürlich muss sichergestellt werden, dass die Integrität der Updatedatei gewährleistet wird. Falls ein Zugriff aus einem externen Netzwerk notwendig ist, müssen entsprechende Protokolle wie beispielsweise Secure Shell (SSH), VPN oder IPSec verwendet werden. Diese stellen unter anderem durch eine Authentifizierung und Verschlüsselung der Daten sicher, dass der Zugriff auf den Server rechtens ist [31]. Im lokalen Netzwerk hingegen reicht in der Regel ein SSH-Zugriff auf den Server aus.

Neben einem sicheren Zugriff auf die Serverinfrastruktur muss natürlich auch eine solche bereitgestellt werden. Diese benötigt einen Ethernet-Port für die Kommunikation mit der MCU. Zum Empfang der Updatedatei ist eine solche nicht zwingend notwendig, sofern eine drahtlose Kommunikation möglich ist. Der Server benötigt eine entsprechende Konfiguration, um als TFTP-Server zu fungieren. Dabei liegt die Hauptaufgabe darin, die Updatedatei im Rahmen des TFTP-Protokolls an den TFTP-Client im Bootloader auszuliefern. Besonders wichtig ist die Sicherstellung der aktuellsten Binärdatei. Dies liegt daran, dass der Bootloader bei jedem Systemstart die Datei anfragt und auf deren Version überprüft.

Eine grundlegende Voraussetzung für den Einsatz des TFTP-Protokolls ist jedoch die Verfügbarkeit eines Ethernet-Ports am eingebetteten System. Dieses kommuniziert nur kabelgebunden mit einem Server. Diese Voraussetzung ist notwendig, um zusätzlichen Overhead bei der Implementierung des Bootloaders einzusparen. So hat beispielsweise ein WLAN neben der direkten Verbindung noch weitere Aufgaben wie beispielsweise das Einholen von Informationen über mögliche Geräte in Reichweite [32]. Zusätzlich muss auf Bootloader-Seite ein TFTP-Client implementiert sein. Wie dieses Protokoll aufgebaut ist und was bei der Anforderung der Updatedatei zu beachten ist, findet sich in nachfolgendem Kapitel.

4.2.2 TFTP-Protokoll

Das TFTP, welches für die Kommunikation zwischen Bootloader und TFTP-Server verwendet wird, ist eine sehr vereinfachte Form des FTP. Dies zeigt sich unter anderem dadurch, dass das TFTP weder eine Authentifizierung noch eine verbindungsorientierte TCP Verbindung benötigt [29, 30]. Das TFTP setzt auf eine möglichst einfache, übersichtliche Kommunikation, die ohne großen Overhead einfach nur zum ungesicherten Dateiaustausch verwendet wird [30].

Das TFTP kommuniziert verbindungslos mittels UDP. Die TFTP-Implementierung findet sich wiederum in der übergeordneten Applikationsschicht. Die Verwendung aller Schichten bedeutet zwar einen Overhead bei der Paketgröße. Diese ist allerdings aufgrund des sehr sparsamen TFTP überschaubar und daher die bessere Option gegenüber FTP [29, 30].

Tabelle 4.1 Pakettypen des Trivial File Transfer Protokolls (TFTP) [30]

Opcode	Operation	Kürzel
0x01	Leseanfrage	RRQ
0x02	Schreibanfrage	WRQ
0x03	Datenpacket	DATA
0x04	Acknowledgment	ACK
0x05	Fehler	ERROR

Die Tabelle 4.1 zeigt die fünf Pakettypen, welcher das TFTP unterstützt. Hieraus geht hervor, dass lediglich das Lesen und Schreiben einer Datei möglich ist, was jedoch für den konkreten Fall des Bootloaders ausreicht. Hier muss lediglich die Updatedatei gelesen werden. Die Pakete 0x03 und 0x04 sind für den Datenaustausch zuständig. Im Fehlerfall wird das Paket 0x05 gesendet.

In Abbildung 4.4 sind diese Pakettypen wiederum entsprechend deren Nachrichtenformate dargestellt.

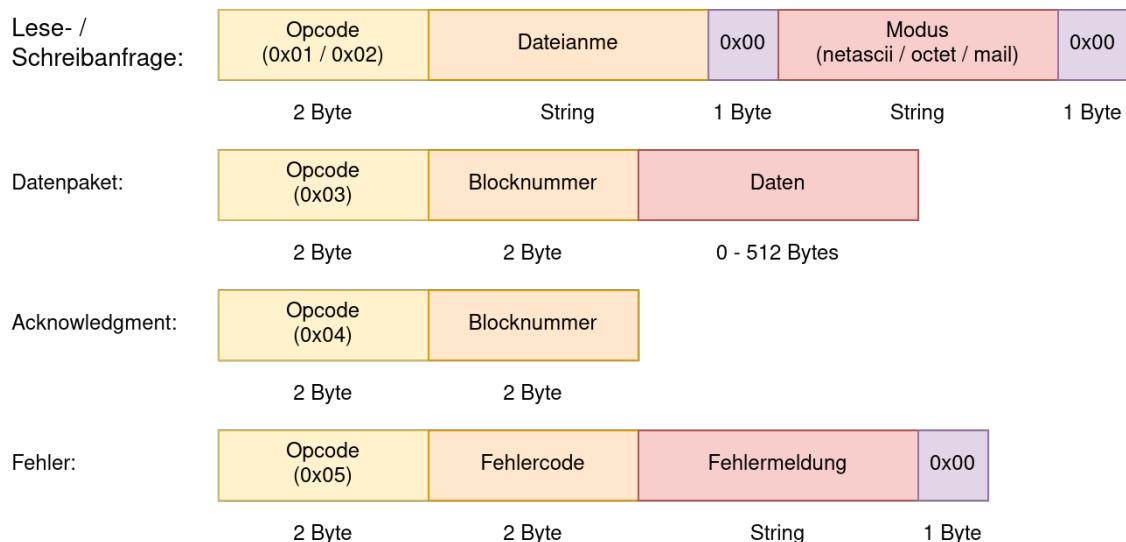


Abbildung 4.4 TFTP Nachrichtenformate
Quelle: Eigene Darstellung in Anlehnung an [30]

4 Aufbau und Architektur

tenformate dargestellt. Um eine Lese- oder Schreibanfrage zu stellen, muss der entsprechende Opcode 0x01 oder 0x02 verwendet werden. Dadurch erfährt der Server das Nachrichtenformat. Als zweiten Schritt muss die Anfrage den genauen Namen der betroffenen Datei beinhalten. Dieser wird durch ein 0x00 Byte vom letzten Teil der Nachricht getrennt. Dieses Zeichen wird verwendet, da es nicht Teil eines Strings sein kann und somit nicht versehentlich ein falscher Dateiname gelesen wird.

Der Modus kann wiederum eines der drei dargestellten Formate definieren. Dieser sagt aus, in welchem Format die Datei vom oder an den Server gesendet wird. Die Unterschiede dieser Modi sind die Folgenden [30]:

- NetAscii: 8 Bit Ascii
- Octet: Binärdaten (8 Bit)
- Mail: Nutzer als Empfänger (Format: NetAscii)

Im Falle einer Leseanfrage antwortet der Server direkt mit dem ersten Datenpaket. Sollte eine Schreibanfrage gestellt worden sein, bestätigt der Server diese mit einem Acknowledgement, woraufhin der Client das erste Datenpaket sendet. Auf jedes Datenpaket antwortet die Empfängerseite mit einem Acknowledgement. Erst nach dessen Erhalt wird das nächste Paket gesendet. Falls keine Bestätigung seitens des Empfängers eingegangen ist, wird das Paket nach Ablauf eines definierten Zeitraums erneut gesendet. Nach mehrfachem Fehlerversuch wird die Übertragung terminiert [30].

Tabelle 4.2 Fehlernachrichten des TFTP [30]

Fehlercode	Fehlernachricht
0x00	Nicht definiert (Eigene Fehlermeldung)
0x01	Datei nicht gefunden
0x02	Zugriff verweigert
0x03	Kein ausreichender Speicherplatz
0x04	Verbotene TFTP-Operation
0x05	Unbekannte Übertragungs-ID
0x06	Datei bereits vorhanden
0x07	Nutzer nicht gefunden

Jedes Datenpaket erhält eine Nummer beginnend bei 1. Da UDP die Paketreihenfolge zwischen Absenden und Empfangen nicht sicherstellt, kann so die richtige Reihenfolge nachempfunden werden. Ein Acknowledgement-Paket enthält dabei die

4.2 Trivial File Transfer Protokoll (TFTP)-Server

Nummer des zuletzt empfangenen Paketes. Die Länge des letzten Datenpakets ist kürzer als 512 Byte. Somit weiß der Empfänger, dass die Übertragung nun beendet ist [30].

Falls ein Fehler während der Kommunikation auftritt, wird eine entsprechende Fehlermeldung gesendet. Insgesamt sind acht Fehlercodes definiert. Diese sind in Tabelle 4.2 aufgelistet [30]. Die Fehlermeldungen mit einem Code zwischen 0x01 und 0x07 sind fest definiert. Hier kann jedoch die Fehlermeldung verwendet werden, um zusätzliche Informationen zum aufgetretenen Problem zu liefern. Sollte ein Fehler auftreten, der nicht abgedeckt ist, so kann mit Code 0x00 ein eigener Fehler gemeldet werden. In diesem Fall gibt ausschließlich die Fehlermeldung Auskunft über die Fehlerquelle.

5 Implementierung des Bootloaders

In Kapitel 4.1 wurde auf die hardwareseitigen Anforderungen an das System eingegangen. Dabei wurde dieses in seine einzelnen Komponenten untergliedert und erläutert, wie und warum diese Verwendung finden. Im Zuge dessen wurde ein umfangreicherer Hardwareeinsatz, als für einen Bootloader benötigt wird, festgestellt. Die verwendete Hardware im System ist jedoch nicht nur für den Bootloader, sondern auch für die Hauptanwendung gedacht und muss daher einen größeren Funktionsumfang mit sich bringen. Festzuhalten ist, dass die Hardware in der Lage ist, alle Funktionen des Bootloaders auszuführen und zusätzlich genügend Spielraum für eine Anwendung bietet.

Nachdem die eingesetzte Hardware nun erläutert und deren Eignung für das Projekt dargelegt ist, wurden mehrere Anforderungen an den Funktionsumfang des Bootloaders definiert. Besonders wichtig ist dabei eine flexible Konfiguration, um den Bootloader ohne viel Aufwand abhängig von Einsatzgebiet und Hauptanwendung anpassen zu können. Zu diesem Zweck wurden folgende zwei Kategorien an Anforderungen definiert.

Zum einen feste Anforderungen, welche nicht konfiguriert werden können:

- Firmware-Update mittels serieller Verbindung
- Individuelle Firmwareerkennung
- Versionserkennung für eine neue Firmware
- Validierung und Verifizierung des Firmware-Updates und -Speichers
- Sprung in die Hauptanwendung am Ende des Bootloader-Zyklus
- Speichereinteilung in drei Bereiche:
 - Bootloader
 - Anwendung
 - Permanente Variablen der Anwendung
- Visuelles Feedback über Bootstatus
- Update-Mechanismus des Bootloaders

5 Implementierung des Bootloaders

Zum anderen flexible Anforderungen, die entsprechend für jede Anwendung anpassbar sind:

- Firmware-Update mittels Ethernet-Verbindung
- Down- und Update-Mechanismus mittels Versionskontrolle
- Backup-Mechanismus der Hauptanwendung
- Visuelles Feedback über angeschlossenes Display

Im nachfolgenden Kapitel wird nun auf diese Anforderungen eingegangen. Dabei wird speziell auf die Umsetzung und etwaige Besonderheiten bei der Implementierung geachtet. Ein weiterer Schwerpunkt dieses Kapitels liegt in der Schaffung der Rahmenbedingungen für die Implementierung. Aus diesem Grund wird nun im folgenden Kapitel die verwendete Entwicklungsumgebung (IDE) näher beleuchtet.

5.1 STM32CubeIDE

Als IDE für einen Mikrocontroller der Firma STM kann zwischen mehreren möglichen Anbietern eine Auswahl getroffen werden. Im Zuge dieser Arbeit wurde die hauseigene IDE von STM verwendet. Dies liegt zunächst an einem erleichterten Einstieg in die Funktionen der STM MCU-Programmierung. So enthält die verwendete STM32CubeIDE bereits die Funktionen der ebenfalls von STM entwickelten STM32CubeMX-Umgebung [33].

Die Firma STM bietet mit deren STM32Cube-Anwendungen Lösungen, um die Produktivität der Entwicklerin oder des Entwicklers zu steigern. Dabei soll vor allem der Einstieg in ein Projekt möglichst komfortabel gestaltet werden. Dies hat das Ziel einer kürzeren Entwicklungszeit und eines geringeren Aufwandes [34].

Die STM32CubeMX-Anwendung dient dem Nutzer dazu, mittels einer grafischen Oberfläche eine erleichterte Auswahl und Konfiguration der Zielhardware vorzunehmen. So bietet das Generationswerkzeug eine Unterstützung bei der Auswahl und Konfiguration der verfügbaren Schnittstellen und internen Systemkomponenten. Zusätzlich wird eine Reihe an Open-Source-Bibliotheken angeboten und bei Bedarf automatisch in das Projekt integriert. Des Weiteren konfiguriert das Programm die Pins entsprechend der getroffenen Auswahl und meldet, falls es zu Konflikten zwischen ausgewählten Komponenten kommt [34].

Durch die Integration der STM32CubeMX in die STM32CubeIDE liefert diese IDE einen einfachen Einstieg in die Entwicklung und unterstützt die Einarbeitung in ein neues unbekanntes System. Die Eclipse-basierte IDE bietet sich darüber hinaus auch während des Programmierens als Basis für die Entwicklung dieses Projektes an. So ist ein GNU C/C++ Compiler für ARM®-basierte MCUs bereits integriert. Vor allem das Debuggen wird mittels eines GDB-Debuggers und diverser Analysefunktionen

des Speichers zur Laufzeit stark gefördert und dient einer schnellen Problemfindung [33].

Durch diesen erleichterten Einstieg und die weitreichende Unterstützung bei der Entwicklung wurde diese IDE als passende Lösung für die Programmierung befunden. Ein weiterer Punkt welcher für die STM32CubeIDE spricht, liegt in der Unterstützung der gängigsten Desktop-Betriebssysteme. So kann die IDE sowohl auf Windows, Linux als auch macOS verwendet werden. Dies und die spezielle Unterstützung von ARM® Cortex Prozessoren steuerte den Entscheidungsprozess maßgeblich und bietet einen passenden Funktionsumfang für die Entwicklung [33].

5.2 Speicherunterteilung

Ein grundlegender Teil der Planung eines Bootloaders liegt in der Einteilung des Speichers. Hier gilt es, feste Adressbereiche für den Bootloader und die Anwendung festzulegen. Die Grenzen zwischen diesen Bereichen wiederum sind fest und lediglich der Bootloader darf auf den Speicherplatz der Applikation zugreifen. Die Anwendung darf nur in dringenden Fällen auf den Speicherbereich des Bootloaders zugreifen [8].

Für die Einteilung gibt es einige Punkte zu beachten. So muss einerseits der gesamt verfügbare Speicher einem der Bereiche zugeordnet werden. Andererseits gilt es, bei der Bestimmung der Bereichsgröße des Bootloaders auf ausreichend Platz für zukünftige Updates zu achten [8].

Da das NUCLEO-F429ZI Discovery Board über einen Flash-Speicher mit einer Größe von 2 MiB verfügt, spielt auch eine dritte Überlegung eine wichtige Rolle. So kann zum Zweck der Datensicherung ein Teil des Speichers für ein Backup der Hauptanwendung verwendet werden. Da dies jedoch den verfügbaren Speicher stark reduziert oder auf anderen Boards gegebenenfalls nicht derartig viel Speicher verfügbar ist, wurde diese Option als konfigurierbar hinzugefügt. So kann vor dem Kompilieren des Bootloader-Codes festgelegt werden, ob eine Backup-Funktion gewünscht ist.

Grundlegend für beide Konfigurationen gilt, dass der Speicher des NUCLEO-F429ZI Discovery Board in zwei Bänke unterteilt ist. Diese weisen die gleiche Struktur auf. Zusätzlich ist jede Bank in zwölf Sektoren unterteilt, welche durch das Programm einzeln gelöscht werden können. Die ersten vier Sektoren jeder Bank haben dabei eine Größe von 16 KiB. Der fünfte Sektor hat eine Größe von 64 KiB und die restlichen Sektoren jeder Bank haben eine Größe von 128 KiB [17]. Wie die beiden Konfigurationsmöglichkeiten unter Berücksichtigung dieser Vorgaben umgesetzt sind, wird nun in den folgenden Kapiteln erläutert.

5.2.1 Vollständige Speicherzuweisung für die Hauptanwendung

Diese Konfigurationsmöglichkeit legt fest, dass kein Backup der Hauptanwendung vorgesehen ist. Dies kann entweder auf Grund eines zu geringen Speichers bestimmt werden oder wegen eines aktuellen oder eventuell zukünftig größeren Speicherbedarfs durch die Hauptanwendung.

Abbildung 5.1 zeigt, wie in diesem Fall der Speicher der MCU unterteilt ist. Dem Bootloader selbst wurden die ersten vier Speichersektoren mit insgesamt 64 KiB zugewiesen. Sektor 5, welcher eine Gesamtgröße von 64 KiB aufweist, wurde als Bereich für permanente Variablen der Hauptanwendung hinterlegt. Der restliche Speicher wurde dem Programmcode der Firmware zugewiesen. Die einzige Besonderheit hierbei sind die ersten 512 Byte von Sektor 6. Diese sind reserviert, um die Version und CRC der aktuell installierten Anwendung für Versionsabgleiche und Verifizierungen zu speichern.

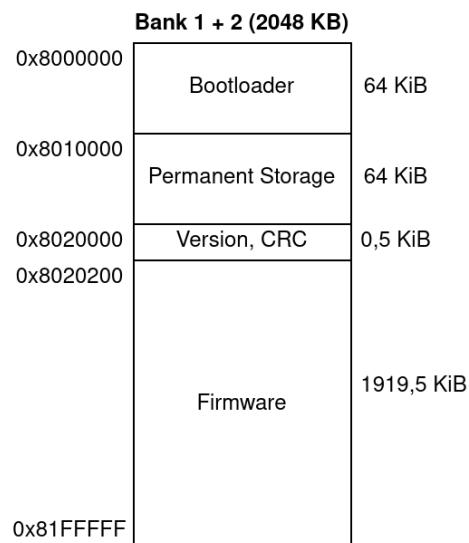


Abbildung 5.1 Einteilung des Programmspeichers ohne Backup-Funktion
Quelle: Eigene Darstellung in Anlehnung an [17]

Der große Vorteil dieser Konfiguration liegt in der Größe des Speicherbereiches der Hauptanwendung. So kann diese insgesamt 1919,5 KiB an Speicher benötigen. Dies ist für eine Anwendung im Bereich der eingebetteten Systeme eine mehr als ausreichende Kapazität für eine Applikation. Dies lässt sich durch die vergleichbaren Angebote an Speichergrößen in der Baureihe der NUCLEO-F429xx Boards erkennen. Das hier vorgestellte Board hat mit 2 MiB den größten Speicher [25] der Reihe. Die nächstkleinere Speichergröße, welche angeboten wird, hat nur noch 1 MiB [17] und ist somit kleiner als der in dieser Speicherkonfiguration verfügbare Platz.

Ein weiterer Vorteil dieser Konfiguration ist der geringere Aufwand des Bootloaders. Dieser muss lediglich die eine Anwendung kontrollieren und keine zusätzlichen Mechanismen pflegen. Näheres zum Programmablauf dieser Konfiguration findet sich in Kapitel 5.3.1.

Der Nachteil, den diese Konfiguration mit sich bringt, liegt in der Fehleranfälligkeit. So kann das System nicht auf Fehler während des Update-Prozesses oder korrumptierte Speicherbereiche im Bereich der Anwendung reagieren. Sollte beispielsweise ein aktives Update abgebrochen werden, ist die zu Teilen installierte Applikation nicht lauffähig. Da jedoch kein Backup verfügbar ist, kann die Funktion nicht durch das System wiederhergestellt werden.

Um eine Anwendung für diese Form der Speicherkonfiguration vorzubereiten, müssen zwei Punkte angepasst werden. Zunächst muss im Linker-Skript der Applikation wie in Codeauszug 5.1 zu sehen der Speicherbereich der Flash-Sektion auf Startadresse 0x8020200 gesetzt werden. Zusätzlich gilt es, die verfügbare Speichergröße auf 1919,5 KiB anzupassen.

```
MEMORY
{
    CCMRAM    (xrw)      : ORIGIN = 0x10000000 , LENGTH = 64K
    RAM        (xrw)      : ORIGIN = 0x20000000 , LENGTH = 192K
    FLASH      (rx)       : ORIGIN = 0x08020200 , LENGTH = 1919K
                           + 0x200
}
```

Codeauszug 5.1 Angepasstes Linker-Skript für eine Speicherkonfiguration ohne Backup

Der zweite Punkt, der für die Implementierung der Hauptanwendung angepasst werden muss, ist der Basis Offset für die Interrupt Vector Table (IVT). Dieser muss als Wert den Abstand 0x20200 zur Basisadresse 0x8000000 des Flash-Speichers enthalten. Dies liegt daran, dass die IVT immer der Startadresse der Anwendung zugeordnet sein muss. Da im Fall der Cortex-M®-Familie diese standardmäßig der Startadresse des Flash-Speichers zugeordnet sind, muss ein Offset um die Verschiebung der Applikation im Speicher angegeben werden [35]. Um Fehler bei diesen Anpassungen zu vermeiden, wurde ein zusätzliches Basisprojekt für diese Konfiguration erstellt und dem Bootloader-Projekt hinzugefügt.

5.2.2 Zusätzliches Backup der Hauptanwendung

Durch diese Konfigurationsmöglichkeit kann die Berücksichtigung eines Backups der Hauptanwendung im Speicher angegeben werden. Dies bietet sich an, wenn die Hauptapplikation voraussichtlich niemals die Größe von 959,5 KiB überschreitet. Abbildung 5.2 zeigt die Unterteilung des Speichers für diese Konfiguration. Für den Bootloader sind die ersten vier Flash-Sektoren zugeteilt. Dadurch kann dieser bis zu

5 Implementierung des Bootloaders

64 KiB Platz benötigen. Die Hauptanwendung, die ausgeführt wird, liegt in den daraufliegenden acht Sektoren. Die ersten 512 Byte enthalten dabei die Informationen zur Versionierung und Verifizierung der installierten Hauptapplikation. Um mit der Firmware dauerhaft Variablen speichern zu können, wurde der gespiegelte Bereich zum Bootloader in der zweiten Flash-Bank reserviert. Dieser bietet gegenüber dem Part für die Konfiguration ohne Backup den Vorteil, dass jeder der vier Sektoren einzeln gelöscht werden kann. In der Konfiguration aus Kapitel 5.2.1 ist dies nicht möglich. Hier muss bei einer Änderung der gesamte Speicherbereich für permanente Variablen gelöscht und neu geschrieben werden. Somit ist in der in diesem Kapitel vorgestellten Konfiguration eine geringere Abnutzung des Flash-Speichers integriert. Der Backup-Bereich dieser Konfiguration ist der gespiegelte Bereich aus Flash-Bank eins. Somit weißt dieser dieselben Eigenschaften bezüglich Sektorunterteilung und Größe auf [17].

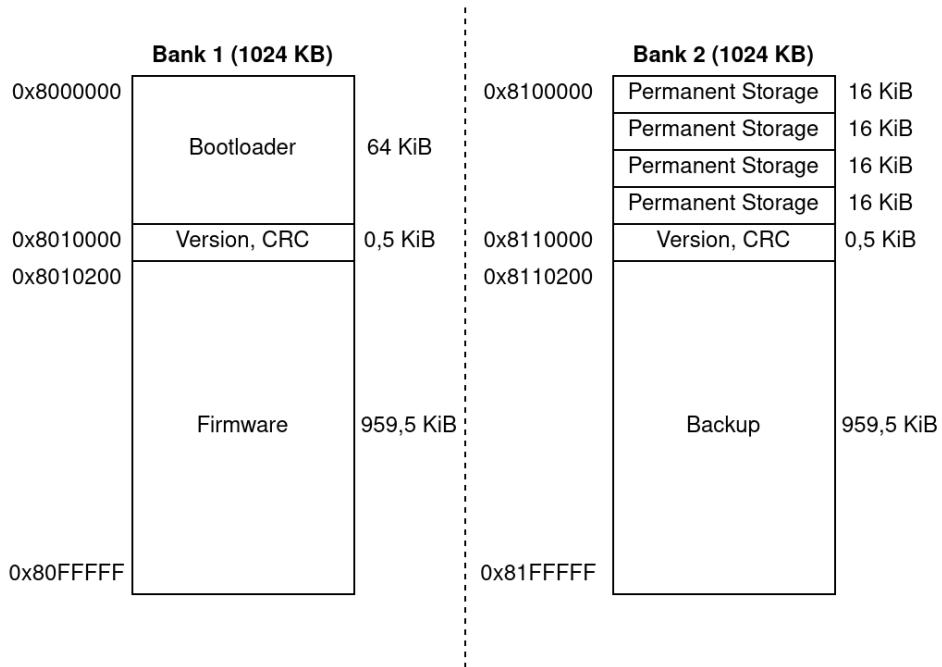


Abbildung 5.2 Einteilung des Programmspeichers mit Backup-Funktion
Quelle: Eigene Darstellung in Anlehnung an [17]

Der große Vorteil, welchen diese Konfiguration bietet, liegt in einer zusätzlichen Sicherheitsstufe. So kann das System auf fehlerhafte Speicherinhalte reagieren. Zusätzlich ist das System dazu in der Lage, Fehler während des Update-Prozesses zu erkennen. In diesem Fall kann das Backup aus der zweiten Flash-Bank in den Speicherbereich der Hauptanwendung kopiert werden. Dies sorgt vor allem im industriellen Umfeld für mehr Sicherheit, da die Anwendung trotz eines schweren Fehlers mit höherer Wahrscheinlichkeit lauffähig ist [9].

Natürlich muss der Bootloader im Zuge dieser Konfiguration mehr Aufgaben erfüllen als ohne Backup-Mechanismus. Deshalb ist der Programmcode des Bootloaders in diesem Fall auch größer und benötigt daher mehr Speicher. Jedoch wurde der Bootloader mit allen speicherintensiven Konfigurationen getestet und erfüllte die Speicherplatz-Kriterien für den Bootloader-Bereich im Flash-Speicher. Eine nähere Beschreibung der Ablauferweiterung des Programms, die diese Konfiguration mit sich bringt, findet sich in Kapitel 5.3.2.

Der große Nachteil dieser Konfiguration ist der geringere Speicherbereich für die Hauptapplikation. Dies gilt es in der Konzeptionsphase der Firmware zu berücksichtigen, da sich die Konfiguration des Bootloaders nach der Inbetriebnahme nicht mehr ändern lässt und daher auch zukünftige Versionen der Hauptanwendung von dieser Einschränkung betroffen sind. Jedoch stellen die verfügbaren 959,5 KiB einen in der Regel ausreichenden Bedarf an Speicher zur Verfügung. Dieser ist fast so groß wie ein Speicherbereich im Fall von Flash-Banking. Da beide Konzepte auf einem Backup-Mechanismus für industrielle Anwendungen basieren, ist ein Vergleich dieser Bereiche hier eine gute Referenz.

Um eine Anwendung für diese Form der Speicherkonfiguration vorzubereiten, müssen wie in Kapitel 5.2.1 erläutert, zwei Punkte angepasst werden. Als Erstes muss im Linker-Skript der Anwendung wie in Codeauszug 5.2 zu sehen, der Speicherbereich der Flash-Sektion auf Startadresse 0x8010200 gesetzt werden. Zusätzlich muss die verfügbare Speichergröße auf 959,5 KiB angepasst werden. Hier wird nur die erste Flash-Bank berücksichtigt. Das Backup in Bank 1 ist daher ausschließlich in Bank 1 lauffähig.

```
MEMORY
{
    CCMRAM    (xrw)      : ORIGIN = 0x10000000 , LENGTH = 64K
    RAM        (xrw)      : ORIGIN = 0x20000000 , LENGTH = 192K
    FLASH      (rx)       : ORIGIN = 0x08010200 , LENGTH = 959K
                           + 0x200
}
```

Codeauszug 5.2 Angepasstes Linker-Skript für eine Speicherkonfiguration mit Backup

Der zweite Punkt, der für die Implementierung der Hauptanwendung angepasst werden muss, ist der Basis Offset für die IVT. Dieser muss, wie in Kapitel 5.2.1 beschrieben, auf die Verschiebung der Startadresse der Anwendung angepasst werden. Dieser muss allerdings den Wert 0x10200 als Abstand zur Basisadresse 0x8000000 des Flash-Speichers enthalten. Um auch hier Fehler bei diesen Anpassungen zu vermeiden, wurde ein zusätzliches Basisprojekt für diese Konfiguration erstellt und dem Bootloader-Projekt hinzugefügt.

5.3 Zustandsautomat

Ein Bootloader kann für unterschiedlichste Einsatzgebiete konzipiert werden. In Kapitel 2 wurde dieser Sachverhalt bereits an der Abhängigkeit des Einsatzgebietes festgemacht. Eine Sache, welche jedoch alle Bootloader gemeinsam haben, ist ein strikter Ablauf. So wird der Bootloader während des Bootprozesses aufgerufen und durchläuft anschließend eine für genau diesen Anwendungsfall passende Routine. Das folgende Kapitel beschreibt nun die zwei Routinen, welche für den im Zuge dieser Arbeit entwickelten Bootloader. Dabei liegt der Unterschied in der Unterstützung eines Backup-Mechanismus, wie in Kapitel 5.2 beschrieben.

5.3.1 Bootprozess ohne Backup-Funktion

Für die Bootloader-Routine des im Zuge dieser Arbeit entwickelten Bootloaders gibt es zwei Optionen. In der hier beschriebenen Variante wird kein Backup der Firmware im Speicher getätigt. Die Gründe für diese Konfiguration werden in Kapitel 5.2.1 näher beleuchtet. Das folgende Kapitel befasst sich nun mit der Routine, welche der Bootloader abarbeitet, um die Anwendung zu starten und gegebenenfalls während dieses Prozesses zu aktualisieren.

Der in diesem Fall grundsätzliche Ablauf der Abarbeitungsreihenfolge des Bootloaders ist in Abbildung 5.3 zu sehen. Hieraus geht hervor, welche Schritte grundlegend

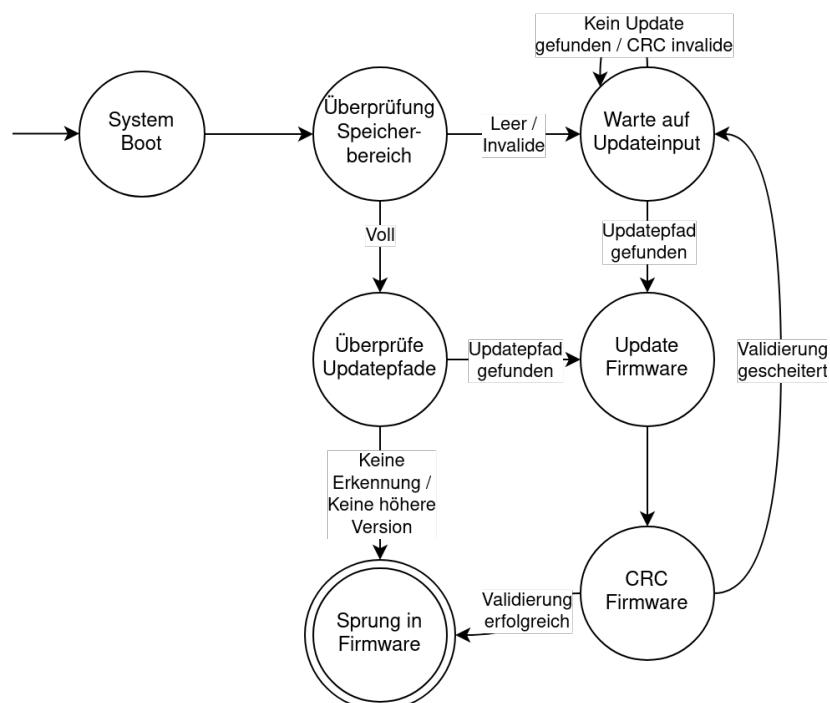


Abbildung 5.3 Zustandsautomat der Bootloader-Routine ohne Backup-Mechanismus
Quelle: Eigene Darstellung

für den Bootloader vorgesehen sind, um seine Aufgaben zu erfüllen. Der hier dargestellte Ablauf ist auch in der in Kapitel 5.3.2 vorgestellten Routine enthalten und stellt daher den grundlegenden Funktionsumfang des Bootloaders dar.

Die erste Aufgabe des Bootloaders ist die Überprüfung des Speichers der Hauptanwendung. Im konkreten Fall wird dabei die Korrektheit des Speicherinhaltes mittels CRC überprüft. Ist keine Anwendung installiert, oder die Validierung fehlerhaft, wird in den Zustand „Warte auf Updateinput“ gewechselt. Dieser Zustand wird so lange aufrecht erhalten, bis eine Updatedatei über USB oder TFTP erkannt wird. Dies bedeutet, dass über diesen Zustand kein direkter Start der Hauptanwendung möglich ist. Dieser kann nur über ein Update erreicht werden.

Wird jedoch bei der Speicherüberprüfung zu Beginn der Routine eine valide Anwendung erkannt, so wird in den Zustand „Überprüfe Updatepfade“ gewechselt. In diesem Fall werden die verfügbaren Schnittstellen einmalig auf eine Updatedatei überprüft. Ist kein Gerät angeschlossen oder keine neuere Firmwareversion bereitgestellt, die an dieses Gerät adressiert ist, wird die Hauptanwendung gestartet. Wenn jedoch eine Updatedatei mit besagten Kriterien verfügbar ist, wird auch in diesem Fall in den Update-Pfad des Bootloaders gewechselt.

Nachdem die Updatedatei auf Gerätezugehörigkeit und Version überprüft wurde, wird im Zustand „Update Firmware“ lediglich der Inhalt dieser Datei in den Speicher der Hauptanwendung kopiert. Dafür wird zunächst ein möglicher alter Softwarestand aus dem Speicher gelöscht. Dies ist notwendig, da Flash-Speicher nicht einfach überschrieben werden können [18]. Anschließend wird die Anwendung, wie in den Kapiteln 5.4.1 und 5.4.2 beschrieben, abhängig von der verwendeten Schnittstelle in den Speicher geschrieben.

Im nächsten Schritt wird über die geschriebene Anwendung ein CRC durchgeführt. Dieser wird mit der CRC-Checksumme aus der Updatedatei verglichen. Nur im Erfolgsfall wird nun die Anwendung gestartet. Scheitert jedoch die Verifizierung, so wird der geschriebene Inhalt gelöscht und die Routine springt zurück in den Zustand „Warte auf Updateinput“. Von hier wird der gesamte Update-Prozess neu gestartet.

Um die Hauptanwendung zu starten, sind einige Schritte seitens des Bootloaders durchzuführen. Eine genaue Beschreibung der Vorgehensweise findet sich in Kapitel 5.7. Dieser Prozess ist unabhängig von der Verwendung einer Backup-Funktion, da in beiden Varianten nur die Hauptanwendung gestartet werden kann und nicht das Backup. Der einzige Unterschied liegt, wie in Kapitel 5.2 beschrieben, in der Startadresse der Firmware.

5.3.2 Bootprozess mit Backup-Funktion

In Kapitel 5.3.1 wurde die Routine des in dieser Arbeit vorgestellten Bootloaders aufgezeigt, falls kein Backup-Mechanismus vorgesehen ist. Dabei wurde ein einfacher Ablauf ersichtlich, welcher sich in seinen Grundzügen auch in der Routine mit Backup-Unterstützung wiederfindet. Eine Übersicht des Ablaufes findet sich anhand des Zustandsautomaten in Abbildung 5.4. Der Backup-Mechanismus erweitert das Grundkonzept aus Kapitel 5.3.1 um fünf weitere Zustände. Diese bieten dem System eine zusätzliche Sicherheit zu den vorhandenen Validierungs- und Verifizierungsmechanismen. Der erweiterte Ablauf und die resultierenden Vorteile werden nun im folgenden Kapitel genauer erläutert.

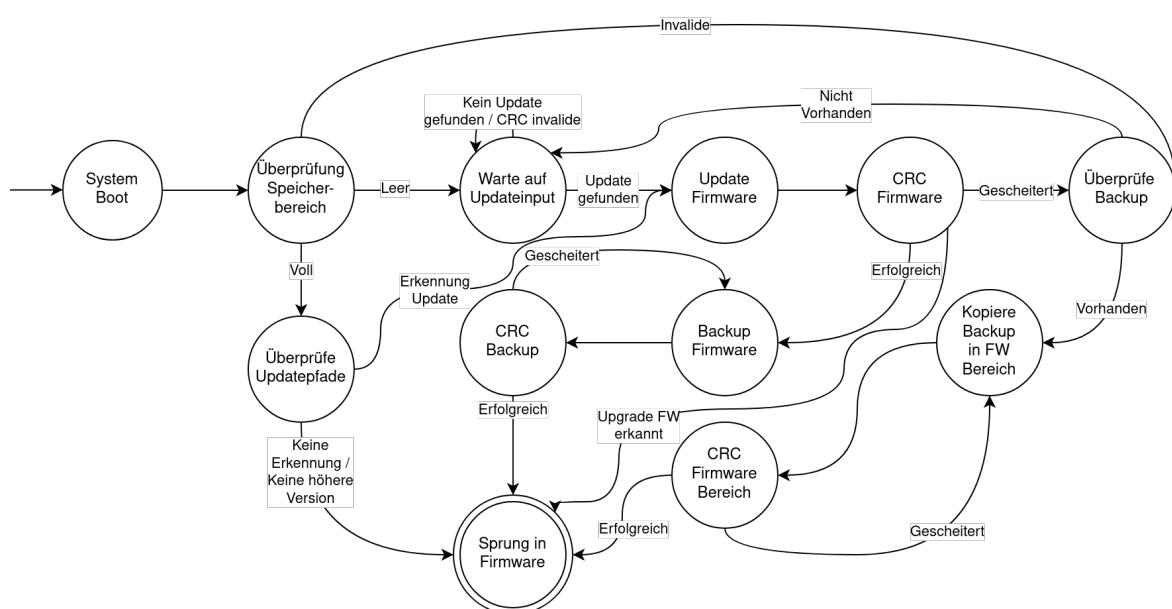


Abbildung 5.4 Zustandsautomat der Bootloader-Routine mit Backup-Mechanismus
Quelle: Eigene Darstellung

Während bei einem abwesenden Backup-Mechanismus zu Beginn der Routine lediglich die Hauptanwendung validiert wird, so kann mit Hilfe dieser Zusatzfunktion ein fehlerhafter Speicherinhalt nicht nur erkannt, sondern auch durch das Backup ersetzt werden. Daraus resultiert eine zusätzliche Sicherheit für das System, welche im Falle einer korrupten Anwendung keine zwingende Interaktion in Form eines Updates erfordert. Das System kann sich selbstständig wieder korrigieren und kann ohne anstehendes Update die wiederhergestellte Applikation starten.

Die Zustände „Überprüfe Updatepfade“, „Warte auf Updateinput“ sowie der direkte Update- und Verifizierungssprozess der Hauptanwendung haben dabei keine Abweichung zum Ablauf aus Kapitel 5.3.1. Der Backup-Mechanismus erweitert anschließend jedoch die Routine um einige wichtige Funktionen.

Zum einen wird direkt im Anschluss an eine erfolgreiche Verifizierung des Updates wie in Abbildung 5.5 dargestellt der Inhalt des Speicherbereiches der Hauptanwendung in den korrespondierenden Backup-Bereich kopiert. Dies ist aufgrund des Zweibanksystems des Flash-Speicherlayouts durch STM möglich. Dieses erlaubt das gleichzeitige Lesen der Inhalte einer Bank und das Schreiben in die andere Bank. STM bezeichnet diesen Mechanismus als „read-while-write“ [25]. Die kopierten Daten werden wie im Fall des Updates zusätzlich im Backup-Bereich verifiziert. Auch dieser Prozess wird im Fehlerfall wiederholt.

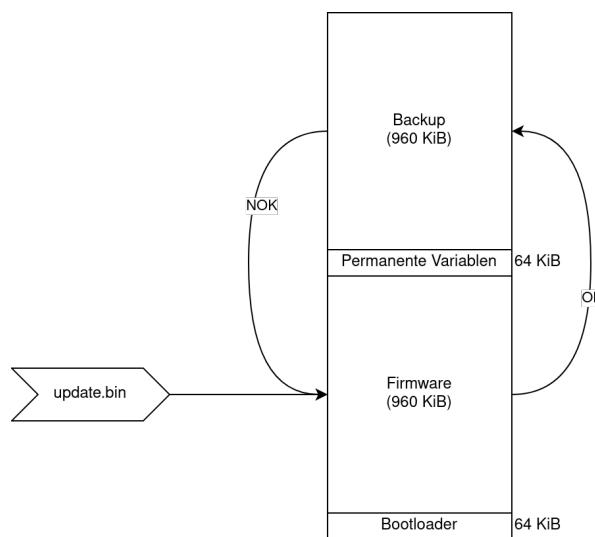


Abbildung 5.5 Speicherbelegung durch Backup-Mechanismus
Quelle: Eigene Darstellung

Zum anderen kann im Fall einer gescheiterten Verifizierung des Updates ein vorhandenes Backup in den Speicherbereich der Hauptanwendung kopiert werden. Dadurch wird das System auf den alten Softwarestand zurückgesetzt. Auch dieser Prozess wird wieder mit CRC verifiziert. Ist kein Backup vorhanden, wird die ursprüngliche Funktionalität ausgeführt und das System wird in einen Wartezustand versetzt. Falls der Fall eintritt, dass eine Updatedatei korrupt ist, kann dieser Backup-Mechanismus verhindern, dass sich das System in einem dauerhaften Update-Prozess entwickelt. Da ein fehlerhaftes Update bei vorhandenem Backup durch dieses ersetzt wird und die Anwendung gestartet wird, kann das System weiterhin betrieben werden. Dennoch gilt es durch den Anwender sicherzustellen, dass die Binärdatei für das Update keine Fehler enthält.

In Kapitel 6.2 wird die Option beschrieben, die Bootloader-Software durch Verwendung einer speziellen Firmware zu aktualisieren. Diese Sonderfirmware stellt einen Sonderfall für den Backup-Mechanismus dar. In Abbildung 5.4 ist zu erkennen, dass in diesem Fall kein Backup der Anwendung gemacht wird. Dies liegt daran, dass das System niemals diese Firmware wiederherstellen darf. Darauf hinaus ist das System unter Verwendung der Backup-Funktion die Anwendungsfirmware

5 Implementierung des Bootloaders

nach einem Bootloader-Update wiederherzustellen, da die Sonderfirmware während der Speicherüberprüfung zu Beginn der Routine erkannt und gelöscht wird. Anschließend kann im Fall des Backup-Mechanismus ein vorhandenes Backup in den Anwendungsspeicher geladen werden.

Nachdem ein Backup in den vorgesehenen Speicherbereich geladen wurde und dessen Verifizierung erfolgreich ist, werden die Werte, welche der Updatedatei wie in Kapitel 5.5 beschrieben, beigefügt sind, in den permanenten 512 Byte großen Speicherbereich direkt vor der Backup-Anwendung geschrieben. Dabei handelt es sich um die CRC-C checksumme. Zusätzlich wird die Dateilänge in Byte hinterlegt. Diese Werte vereinfachen den Kopier- und CRC-Prozess im Wiederherstellungsfall, da die Länge nicht extra durch eine erneute Speicherüberprüfung bestimmt werden muss.

5.4 Firmware-Update

Ein zentraler Aspekt des Bootloaders im Bereich der eingebetteten Systeme liegt im Update der Hauptanwendung des Systems. Zu diesem Zweck wird eine Möglichkeit benötigt, um die Updatedatei in das System zu kopieren. In Kapitel 4.1.3 wurde bereits beschrieben, welche Schnittstellen zu diesem Zweck in das System integriert sind. Dabei handelt es sich sowohl um eine USB-Schnittstelle für das direkte Update am Gerät, als auch eine Ethernet-Schnittstelle, um Updates auch aus größerer Entfernung an das System kommunizieren zu können. Die folgenden beiden Kapitel geben nun einen detaillierten Einblick in die Implementierung dieser beiden Schnittstellen und weisen deren Besonderheiten auf.

5.4.1 Direktes USB-Update

Um eine Datei von einem USB-Stick zu lesen, benötigt das System grundlegend zwei Komponenten. Einerseits eine Hardware-Schnittstelle, um den USB-Stick anzuschließen. Andererseits die softwareseitige Schnittstelle, um auf die Inhalte des USB-Sticks zugreifen zu können. Da das NUCLEO-F429ZI Discovery Board nur mit einer Micro USB-Schnittstelle ausgestattet ist [25], wird an dieser Stelle ein Adapter auf USB-Typ A benötigt. Dies gilt es auch bei der Entwicklung einer eigenen Platine zu beachten.

Softwareseitig gilt es die Schnittstelle so zu implementieren, dass Daten gelesen werden können. Zu diesem Zweck muss zum einen die serielle Schnittstelle aktiviert und zum anderen ein Dateisystem definiert werden. Dieses muss der USB-Stick aufweisen, um für das System lesbar zu sein. Im konkreten Fall wurde für Letzteres das File Allocation Table Dateisystem (FATFS) verwendet. Dieses wurde Ende der Siebzigerjahre von der Firma Microsoft entwickelt und wurde mit der Zeit immer weiter verbessert. Mittlerweile unterstützt FATFS bis zu 32 Bit-Einträge [36] und ist

daher ideal für den Einsatz in Verbindung mit der 32 Bit-Architektur des Cortex-M4® geeignet. Zusätzlich wird FATFS von allen gängigen Desktop-Betriebssystemen unterstützt. Dies und der Fakt, dass, wie in Kapitel 5.1 beschrieben, die IDE auf all diesen Betriebssystemen lauffähig ist, führt zu einem erleichterten Prozess zwischen Entwicklung und Update der neuen Firmware.

Um einen USB-Stick im System zu erkennen, gilt es, die Pin-Konfiguration der MCU anzupassen. Ein Problem, welches dennoch bei der Entwicklung aufgekommen ist, liegt in der Stromversorgung der Hardware-Schnittstelle. Diese wurde zum Zeitpunkt der Entwicklung nicht bei der Codegenerierung durch die STMCubeIDE vorgenommen. Um dies zu korrigieren, muss der Pin G6 zur Stromversorgung während der Hardwareinitialisierung auf digital High gesetzt werden [17]. Nur so wird die USB-Schnittstelle von der Software erkannt, da der Port ansonsten nicht aktiv ist.

Für die Kompatibilität mit dem FATFS wurde für die Entwicklung eine Open-Source-Bibliothek verwendet. Diese wird ebenfalls durch die STM32CubeIDE bereitgestellt und ist speziell für den Einsatz auf kleinen MCUs ausgelegt. In der Lizenzbeschreibung der Bibliothek wird explizit darauf hingewiesen, dass deren Verwendung für alle Arten von Projekten erlaubt ist und der Code dieser Projekte auch kommerziell genutzt werden darf [37]. Die Bibliothek stellt sämtliche Funktionen bereit, um die Schnittstelle zum FATFS des USB-Sticks zu nutzen. Neben den Funktionen, um den USB-Stick zu registrieren, kann das Dateisystem des USB-Sticks durchsucht und dessen Inhalt gelesen und verändert werden. Als Konfiguration wurde im Zuge der Entwicklung die Option des „Long file name“ verwendet. Diese ist seit FATFS Version 0.07 verfügbar und erlaubt Dateinamen mit einer Länge von bis zu 255 Byte [37].

Um eine Updatedatei von einem angeschlossenen USB-Stick zu lesen, werden insgesamt fünf Funktionen der FATFS-Bibliothek verwendet. Chronologisch muss der USB-Stick mit seinem Dateisystem zuerst registriert werden. Anschließend wird die Updatedatei geöffnet. Hierfür muss diese im Wurzelverzeichnis des USB-Sticks liegen und den Namen tragen, welcher in der Konfigurationsdatei des Bootloaders vor dessen Kompilierung festgelegt wurde. Standardmäßig lautet dieser „update.bin“. Nach dem Öffnen wird die Datei insgesamt bis zu zweimal eingelesen. Der erste Leseprozess wird benötigt, um die Validierungs- und Verifizierungsdaten der Datei zu erfassen und zu prüfen. Diese werden in Kapitel 5.5 näher beschrieben. Ist diese Prüfung erfolgreich, wird die Datei ein zweites Mal eingelesen, um die Applikation in den Speicher zu laden. Dieser Prozess ist in Kapitel 5.4.3 genauer erläutert. Abschließend muss die Datei wieder geschlossen und der USB-Stick mit seinem Dateisystem freigegeben werden.

Um diese Funktionen auszuführen, wurde für den Bootloader eine eigene Bibliothek implementiert, welche unter Verwendung der FATFS-Bibliothek Funktionen bereitstellt, um eine Verbindung mit dem USB-Stick einzugehen, die Datei zu finden, zu öffnen, zu lesen und auszuwerten. Da das eingebettete System nur einen

5 Implementierung des Bootloaders

Arbeitsspeicher mit einer Größe von 192 KiB aufweist und eine Binärdatei für die Hauptanwendung bis zu 1919,5 KiB groß sein kann, wird die Datei nicht im Ganzen in den Speicher geladen. Ferner wird diese in 16 KiB-Blöcken eingelesen. Die ersten 12 Byte des ersten Blocks enthalten die Version, CRC-Cheksumme und die Gerätekennung. Diese werden ausgewertet. Die restlichen eingelesenen Daten werden anschließend in den Speicher geschrieben. In Codeauszug 5.3 aus der Datei „usb_update.c“ ist ersichtlich, wie ein derartiger Leseprozess vorstatten geht. Die hier dargestellte Funktion liest die Datei blockweise und dient der Verifizierung des Inhaltes der Datei. In den Arbeitsspeicher werden somit maximal 16 KiB an freiem Speicher benötigt. Selbiger Einlesemechanismus wird auch während des Speichervorgangs angewendet.

```
||| bool usb_update_read_from_usb(FIL *file , uint8_t *rtext , uint32_t length)
||| {
|||     uint32_t current_block_length = length;
|||     uint32_t bytes_read = 0;
|||     bool status = true;
|||     [...]
|||
|||     while((current_block_length > 0) && status)
|||     {
|||         uint16_t write_length = (current_block_length >= PAGE_SIZE)
|||             ? PAGE_SIZE
|||             : current_block_length;
|||         if (f_read(file , rtext , write_length , (void *)&bytes_read)==FR_OK)
|||         {
|||             [...] // Verifizierung der Datei
|||             current_block_length = (current_block_length >= PAGE_SIZE)
|||                 ? current_block_length - PAGE_SIZE
|||                 : 0;
|||         }
|||         else      // Lesen der Datei fehlgeschlagen
|||         {
|||             [...]
|||             status = false;
|||         }
|||     }
|||     return status;
||| }
```

Codeauszug 5.3 Blockweises Lesen der Updatedatei über die USB-Schnittstelle

5.4.2 Remote Ethernet-Update

In Kapitel 4.1.3 wird der Nutzen eines Updates mittels Ethernet-Schnittstelle erläutert. Ist das System schwer zugänglich oder eine direkte Auslieferung der Datei zum System wird anderweitig verhindert, so schafft ein derartiger Zugang Abhilfe. In Kapitel 4.2 wird wiederum der Entscheidungsprozess für das TFTP-Protokoll

erläutert. Dieses ist in der Lage, eine Datei von einem Server mittels UDP an einen Client zu senden. Der Bootloader stellt in diesem Zusammenhang den Client der Verbindung dar. Um diese Rolle auszuführen, wurde entsprechend der Vorgaben des TFTP-Protokolls ein solcher Client in der Anwendungsschicht der Ethernet-Schnittstelle implementiert.

Die restlichen Funktionen für eine erfolgreiche Ethernet-Kommunikation werden durch die Bibliothek „LWIP“ [38] bereitgestellt. Diese implementiert eine vereinfachte Form des TCP/IP-Protokolls und wird aktiv durch die Open-Source-Gemeinschaft verbessert. Die Bibliothek ist mit der BSD-Lizenz veröffentlicht und darf daher auch in kommerziellen Projekten verwendet werden [39]. LWIP unterstützt in seiner vereinfachten Form nur eine Handvoll an Protokollen. Darunter befindet sich sowohl TCP als auch UDP. Für die Anwendungsschicht ist sogar ein TFTP-Server verfügbar. Da der Bootloader jedoch einen TFTP-Client benötigt, muss dieser selbst implementiert werden.

Für diese Implementierung muss zunächst die LWIP-Bibliothek konfiguriert werden. Durch deren Anlehnung an das TCP/IP-Protokoll erklärt sich die Strukturierung nach dessen Muster. In Abbildung 5.6 ist dargestellt, wie diese Anlehnung umgesetzt ist. So gibt es für jede Schicht des Modells eine Klasse, welche die entsprechenden Funktionen ausführt.

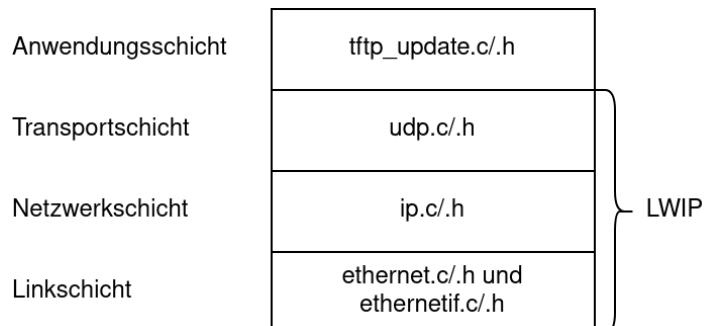


Abbildung 5.6 LWIP TCP/IP Implementierung
Quelle: Eigene Darstellung in Anlehnung an [38, 40]

LWIP bietet insgesamt drei Schnittstellen zur Interaktion mit dem Programmcode. Da der Bootloader aber kein Betriebssystem verwendet, steht lediglich die sogenannte RAW-API zur Auswahl. Diese birgt wiederum den Nachteil, dass sowohl die Netzwerk-Schnittstelle als auch der Nachrichtenempfang durch den eigenen Programmcode des Bootloaders verwaltet werden müssen [39].

Um besagte Netzwerk-Schnittstelle zu konfigurieren, wird zunächst eine IPv4-Adresse für den Bootloader vergeben. Diese wird vor dem Kompilieren in der Datei „tftp_config.h“ festgelegt. Hier wird auch eine Netzmaske, der Gateway und die Ziel IPv4-Adresse angegeben. Mit dieser Adresse wird nun eine neue

5 Implementierung des Bootloaders

Netzwerk-Schnittstelle angelegt und im LWIP hinterlegt. Über diese wird dann die TFTP-Kommunikation ablaufen. Hierfür stellt LWIP eine eigene Schnittstelle mit dem Namen „netif“ zur Verfügung. Diese verfügt über die notwendigen Funktionen. LWIP übernimmt während der Kommunikation auch alle verbindungsorientierten Funktionen wie die Bekanntmachung der Verbindungsdaten im Netzwerk mittels ARP. Diese wird mit den Daten der Netzwerk-Schnittstelle bewerkstelligt [38].

Für den Nachrichtenempfang wird während der Initialisierung des TFTP-Protokolls eine Callback-Funktion eingerichtet. Diese wird bei jedem Durchlauf der Main-Funktion aufgerufen und überprüft, ob eine neue Nachricht angekommen ist [39]. Die Kommunikation zum RJ45-Port der Hardware wird über Direct Memory Access (DMA) an die konfigurierten Pins übermittelt [38].

Um den TFTP-Client zu implementieren, wurde die in Abbildung 5.6 gezeigte Klasse „tftp_update“ angelegt. In dieser werden alle notwendigen Initialisierungsschritte für LWIP getätigt. Zusätzlich wird die Interaktion mit UDP implementiert. Das TFTP-Protokoll, welches in Kapitel 4.2.2 beschrieben ist, verfügt über einen einfachen Befehlssatz. Im Fall des Bootloaders wird darüber hinaus lediglich die Leseanfrage benötigt. Die Schreibanfrage ist daher in dieser Implementierung nicht vorhanden.

Um eine Anfrage über das LWIP senden zu können, wird über den TFTP-Port 69 kommuniziert [30]. Dafür sendet der Bootloader von einem während der Initialisierung zufällig gewählten Port an diesen Port. Dieser Port wird jedoch nur für die initiale Anfrage verwendet. Anschließend sendet der Server von einem anderen Port die Antwort. Über diesen Port läuft auch der Rest der Kommunikation. Um diese Ports in der Implementierung einzubeziehen, liefert LWIP entsprechende Funktionen innerhalb der UDP-Schnittstelle. So kann beispielsweise mit der Funktion „udp_bind“ ein Port für den Bootloader festgelegt werden. Der Port des Servers kann mit der Funktion „udp_connect“ festgehalten werden [38]. Da UDP ein verbindungsloses Protokoll [40] ist, ist dies jedoch keine wirkliche Verbindung sondern lediglich ein Abspeichern der Verbindungsdaten.

326	64.875975041	192.168.178.200	192.168.178.100	TFTP	61 Read Request, File: update.bin, Transfer type: octet
327	64.884540420	192.168.178.100	192.168.178.200	TFTP	558 Data Packet, Block: 1
355	69.884850866	192.168.178.100	192.168.178.200	TFTP	558 Data Packet, Block: 1
367	72.337852110	192.168.178.200	192.168.178.100	TFTP	60 Acknowledgement, Block: 2
368	72.3388082682	192.168.178.100	192.168.178.200	TFTP	558 Data Packet, Block: 2
369	72.345884470	192.168.178.200	192.168.178.100	TFTP	60 Acknowledgement, Block: 2
370	72.346000667	192.168.178.100	192.168.178.200	TFTP	558 Data Packet, Block: 3
371	72.353846101	192.168.178.200	192.168.178.100	TFTP	60 Acknowledgement, Block: 3
372	72.353951465	192.168.178.100	192.168.178.200	TFTP	558 Data Packet, Block: 4
373	72.361834191	192.168.178.200	192.168.178.100	TFTP	60 Acknowledgement, Block: 4

Abbildung 5.7 Wireshark-Aufzeichnung einer TFTP-Übertragung

Quelle: Eigene Aufzeichnung

Das TFTP ist ein reaktives Protokoll. Dies zeichnet sich dadurch aus, dass der Bootloader initial eine Leseanfrage an den Server stellt. Wird diese beantwortet, sendet

der Server nach und nach die Datenblöcke wie in Abbildung 5.7 abgebildet. Die Implementierung des Protokolls ist also darauf ausgelegt, die empfangenen Daten in den Flash-Speicher zu schreiben und ein Acknowledgement zu senden. Um sicherzustellen, dass die Datei auf dem TFTP-Server korrekt und für dieses Gerät gedacht ist, werden deren Parameter nach Erhalt des ersten Paketes überprüft. Dies ist der Grund für die Zeitverzögerung, die in Abbildung 5.7 bei Empfang des ersten Datenpakets zu erkennen ist. Dies löst eine Zeitüberschreitung aufseiten des Servers aus und führt zu einem erneuten Senden der Nachricht. Falls die Prüfung der Updatedatei fehlerhaft ist, so wird eine entsprechende Fehlermeldung an den Server übermittelt und die Kommunikation beendet. Andernfalls wird ein Acknowledgement gesendet und die fortlaufenden Pakete werden anhand ihrer Paketnummer in den Flash-Speicher geschrieben.

Die Implementierung des TFTP-Clients verfügt nicht über alle Fehlermeldungen des Protokolls. Hier wurden nur die Meldungen berücksichtigt, die für einen Client relevant sind. Aus diesem Grund werden ausschließlich zwei Fehlermeldungen implementiert. Der erste Fehler, welchen der Bootloader sendet, wird ausgelöst, wenn eine andere Nachricht als ein Datenpaket eintrifft. Dies liegt daran, dass der Bootloader nur Daten empfängt und daher keine anderen Pakettypen erhalten darf. Dies ist eine Standardfehlermeldung des TFTP und bedarf keiner individuellen Fehlermeldung. Der zweite mögliche Fehler wird bei einer gescheiterten Überprüfung der Updatedatei ausgelöst. In diesem Fall wird dem Server mitgeteilt, dass die Datei nicht durch den Bootloader validiert werden kann. Da dies keine Standardfehlermeldung des TFTP ist, muss diese mit dem Fehlercode „0“ und einer individuellen Fehlernachricht gesendet werden.

5.4.3 Speicherverwaltung

Um den Flash-Speicher der MCU zu verwalten, müssen mehrere Aufgaben erfüllt werden. Zunächst muss ein Mechanismus bereitgestellt werden, der das Schreiben der Anwendung in den Anwendungsspeicher gewährleistet. Diesem müssen Kontrollmechanismen beigegeben werden, um das Geschriebene zu validieren und zu verifizieren. Auch das Löschen des ganzen oder eines bestimmten Speicherbereiches muss durch die Speicherverwaltung abgedeckt werden [8]. Da zusätzlich zwei Konfigurationsmöglichkeiten bestehen, welche eine unterschiedliche Speicherbelegung vorgeben, muss auch dies automatisch berücksichtigt werden.

Wird ein Update über die USB-Schnittstelle getätigt, so gestaltet sich der Speicherprozess linear. Da die Daten in der korrekten Reihenfolge vorliegen, können diese in 16 KiB Blöcken in den Flash-Speicher geladen werden. Diese Aufteilung ist notwendig, da aufgrund der eingeschränkten RAM-Größe der MCU nicht die gesamte Anwendung auf einmal gelesen werden kann.

5 Implementierung des Bootloaders

Falls jedoch ein Update über die Ethernet-Schnittstelle eingespielt wird, muss die korrekte Reihenfolge der Daten anderweitig sichergestellt werden. Dies liegt an der Verwendung von UDP im Zuge des TFTP. UDP kann nicht garantieren, dass gesendete Pakete in dieser Reihenfolge auf Empfängerseite ankommen. Bei TFTP besitzt jedoch jedes Datenpaket eine Blocknummer, welche die korrekte Reihenfolge der Daten angibt. Dieser Mechanismus wird auch für das Speichern der Daten verwendet. Anhand der Blocknummer und der maximalen Paketgröße von 512 Byte pro Datenpaket lässt sich die Position des Blocks in der Nachricht ermitteln. Somit kann jedes empfangene Paket an die entsprechende Stelle im Speicher geschrieben werden.

Um die geschriebenen Einträge zu validieren, wird über den gewünschten Speicherbereich eine CRC-C checksumme gebildet. Der resultierende Wert wird anschließend mit dem hinterlegten Wert der Firmware abgeglichen. Stimmen diese Werte nicht überein, ist der Eintrag im Speicher fehlerhaft und wird direkt gelöscht. Für den Fall einer erfolgreichen Validierung wird dies anhand einer Kennung im Speicher vermerkt, um bei etwaigen Speicherüberprüfungen die Validität der Anwendung sicherzustellen. Dieser Prozess wird teilweise durch eine vom Prozessor ausgelagerte CRC-Recheneinheit durchgeführt. Die Einspeisung der Daten und Handhabung dieser Recheneinheit muss jedoch manuell erfolgen. Dabei gilt es die Endianness der Daten bei der Kommunikation mit der Recheneinheit zu beachten.

Falls die Konfiguration des Bootloaders über einen Backup-Modus verfügt, werden auch die daraus resultierenden notwendigen Schritte in der Speicherverwaltung getätigt. Wie in Kapitel 5.3.2 beschrieben, wird das Backup direkt nach der Installation und erfolgreichen Verifizierung der Firmware als Kopie dieser Anwendung in den vorgesehenen Speicherbereich geladen. Diese Kopie muss anschließend ebenfalls verifiziert werden. Im Erfolgsfall wird die CRC-C checksumme im Speicher hinterlegt, um im Fall einer fehlerhaften Firmware das Backup erneut zu validieren, um dessen Funktionssicherheit zu gewährleisten.

Wurde die Hauptanwendung in irgendeiner Weise beschädigt, so wird diese über die Speicherverwaltung automatisch gelöscht. Anschließend wird das Backup in den Hauptspeicher kopiert und verifiziert. Ist dies erfolgreich, kann die Hauptanwendung so wieder hergestellt werden.

5.5 Kontrollmechanismen

Der im Zuge dieser Arbeit entwickelte Bootloader verfügt über einige individuelle Optionen, um eine möglichst große Flexibilität im Hinblick auf sein Einsatzgebiet zu erreichen. Dieser Aspekt führt zur Notwendigkeit einiger Kontrollmechanismen. Diese Mechanismen sollen dem Nutzer unter anderem eine Konfiguration der gewünschten Systemkomponenten ermöglichen. Darüber hinaus ist die Kernaufgabe des Bootloaders eine Update-Funktion der im System installierten Anwendung. Um deren Sicherheit zu gewährleisten, müssen Vorkehrungen zur Fehlerverhinde-

rung getroffen werden. Das folgende Kapitel befasst sich mit der Umsetzung dieser Mechanismen und zeigt weitere Funktionen auf, welche einen reibungslosen und nutzerfreundlichen Systemablauf gewährleisten.

In Tabelle 5.1 sind die wichtigsten Optionen gelistet, um den Bootloader individuell anzupassen. Dabei ist eine Unterscheidung in drei Kategorien anhand der Dateinamen zu erkennen. In der Datei „bootloader.h“ finden sich die Konfigurationen, welche das Gesamtsystem beeinflussen. Falls ein Update über TFTP gewünscht ist, können die Verbindungseinstellungen in der Datei „tftp_config.h“ getroffen werden. Darüber hinaus kann in der Datei „lcd_conf.h“ ein LCD mit dessen Parametern angelegt werden. Genaueres zu dieser Konfiguration findet sich in Kapitel 5.6.

Tabelle 5.1 Konfigurationsoptionen des Bootloaders

Option	Beschreibung	Datei
DEVICE_SEED	Wert zur Geräteidentifikation	bootloader.h
BINARY_NAME	Name der Updatedatei	bootloader.h
DISPLAY_AVAIL	LC-Display Unterstützung	bootloader.h
TFTP_SUPPORT	TFTP-Update Unterstützung	bootloader.h
BACKUP_OPTION	Backup-Mechanismus Unterstützung	bootloader.h
IP-Einstellungen	Alle IP-Einstellungen für TFTP-Update	tftp_config.h
MAC-Adresse	MAC-Adresse des Gerätes	tftp_config.h
LCD-Werte	Alle Display spezifischen Einstellungen	lcd_conf.h

Der Parameter „DEVICE_SEED“ legt einen Identifikator in Form einer Zahl fest. Dieselbe Zahl muss der Updatedatei an der entsprechenden Stelle angefügt sein. Während der Prüfung der Datei vergleicht der Bootloader seinen Identifikator mit dem der Datei und nur bei Übereinstimmung wird der Prozess weitergeführt. Sind die Werte nicht gleich, gilt die Datei nicht für dieses Gerät. Der Name der Updatedatei wird über den Parameter „BINARY_NAME“ festgelegt. Dieser ist standardmäßig auf „update.bin“ gesetzt.

Wer optionale Funktionen wie einen LCD, Updates über TFTP oder einen Backup-Mechanismus wünscht, kann diese in der Datei „bootloader.h“ aktivieren. Ist dies der Fall, müssen zusätzlich die Verbindungseinstellungen und die Geräte MAC-Adresse in der Datei „tftp_config.h“ angepasst werden. Das Display, dessen Unterstützung aktiviert werden kann, muss mit seinen individuellen Hardwareeinstellungen über die Datei „lcd_conf.h“ eingerichtet werden.

5 Implementierung des Bootloaders

Zur Steuerung eines geregelten Programmablaufs wird ein Timer eingesetzt. Dieser garantiert zu Beginn des Bootloaders genügend Zeit, um die einzelnen Systemkomponenten zu initialisieren. Im späteren Programmablauf wird das visuelle Feedback des LCD über diesen Timer gesteuert. So wird in einem Intervall von 0,5 Sekunden die Bildschirmanzeige erneuert. Auch der Timeoutzähler des TFTP wird mittels dieses Timers weiter gezählt.

Für eine bessere Usability des Systems wird ein Feedback-Mechanismus benötigt. Dieser zeigt dem Nutzer den aktuellen Stand des Bootprozesses visuell auf. Nativ werden dafür drei LEDs verwendet. Diese unterscheiden sich in ihrer Farbe. Die angeschlossenen LEDs haben die Farben Rot, Blau und Grün. Die Bedeutung der jeweiligen Farben ist in Tabelle 5.2 beschrieben. So indiziert eine blaue LED eine aktive Verbindung. Grün wird aktiviert, sobald der Update-Prozess gestartet wurde und Rot ist dann aktiv, wenn ein Fehler in der Hauptanwendung erkannt wurde und diese durch die Backup-Datei ersetzt wird. Jede Firmware-Datei muss mit drei zusätzlichen Parametern versehen werden. Zum einen eine „DEVICE_SEED“, welche wie im vorherigen Verlauf des Kapitels beschrieben ein Identifikator für das Zielgerät ist. Zum anderen eine Versionsnummer und eine CRC-Checksumme für die Datei selbst.

Tabelle 5.2 Farbbe bedeutung des LED Feedback-Mechanismus

Farbe	Bedeutung
Blau	USB- oder TFTP-Verbindung aktiv
Grün	Update-Prozess aktiv
Rot	Anwendungswiederherstellung aus Backup aktiv

Die Versionsnummer gibt dem Bootloader die Information, ob ein Update notwendig ist. Weicht die Versionsnummer der aktuell im System installierten Hauptanwendung ab, so wird ein Update durchgeführt und die neue Version wird im System hinterlegt. Dieser Mechanismus ist aus mehreren Gründen sehr praktisch. Zum einen lässt sich damit jede Softwareversion eindeutig identifizieren, was dem Entwicklungsprozess zugutekommt. Zum anderen wird eine Version, welche an einer der Update-Schnittstellen des Bootloaders verfügbar ist, nur einmal aufgespielt und nicht nach jedem Systemstart erneuert.

Während des Bootprozesses gilt es, mehrmals Daten zu überprüfen. Dies kann sowohl für Daten im Speicher als auch neue Daten aus einer der Update-Schnittstellen notwendig sein. Zu diesem Zweck wurde der Bootloader mit einem Verifizierungsmechanismus in Form eines CRC ausgestattet. Der hier verwendete CRC bildet seine Checksumme über 32 Bit-Blöcke. Welche Folgen diese Form des CRC auf die Erstellung der Updatedatei hat, wird in Kapitel 7.1 näher erläutert. Die Hardware der NUCLEO-F42xxx-Baureihe von STM verfügt für eine schnellere Berechnung

der CRC-Cheksumme über eine zusätzliche Berechnungseinheit. Diese gibt die Limitierung auf 32 Bit-Blöcke vor. Eine derartige Berechnungseinheit gilt es daher bei der Planung einer eigenen Hardware zu berücksichtigen.

Um sowohl die Versionsnummer als auch die CRC-Cheksumme der aktuell installierten Datei für Vergleiche abrufen zu können, werden diese im Flash-Speicher gespeichert. Als Speicherbereich sind 512 Byte vor dem Anwendungsbereich und entsprechend auch vor dem Backup-Bereich reserviert. Die Größe von 512 Byte liegt am Basisoffset für die IVT. Dieser muss für die vorliegende Hardware ein Vielfaches von 512 sein, weshalb dieser Offset für die Startadresse des Anwendungsbereichs berücksichtigt werden muss.

5.6 Anbindung des optionalen LC-Displays

Als zusätzliche Option für die Konfiguration kann ein LCD in das System eingebunden werden. Der Mehrwert darin liegt in einem verbesserten Nutzerfeedback im Vergleich zur nativen Lösung durch LEDs. Das Display übernimmt, wie in Abbildung 5.8 zu sehen ist, die Farbgebung der LEDs. Zusätzlich erhält der Nutzer Feedback zum Status des Systems im Allgemeinen. Dies wird durch ein Durchschalten der Quadratfarbe erreicht. Eines der Quadrate erhält für die entsprechende Farbe jeweils einen dunkleren Ton. In einem halbsekündlichen Intervall wird dieses zur Laufzeit weitergeschaltet, sodass ein Kreislauf entsteht. Der Nutzer kann durch dieses wechselnde Bild erkennen, dass das System arbeitet. Dies kann vor allem während des Update-Prozesses sehr nützlich sein, da das System hier je nach Größe der Anwendung länger beschäftigt sein kann. Der Nutzer kann dadurch trotz des

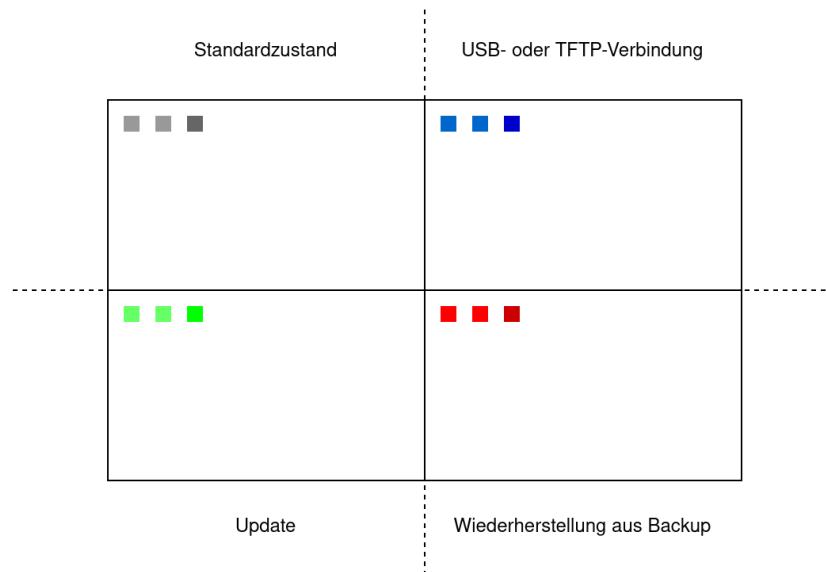


Abbildung 5.8 Anzeigoptionen des LC-Displays
Quelle: Eigene Darstellung

5 Implementierung des Bootloaders

längerem Zeitraums ohne Statusänderung erkennen, dass das System immer noch aktiv ist und kein Fehlerfall eingetreten ist.

Um ein LCD an den Mikrocontroller anzuschließen, ist eine RGB888-Schnittstelle mit 24 Bit verfügbar [25]. Diese kann bei einer eigenen Hardwarekonfiguration für diesen Anwendungsfall berücksichtigt werden. Für die Bootloader-Funktionalität in Form des Feedback-Mechanismus ist eine alleinige Ausgabe ausreichend. Die Konfiguration muss daher keine Eingaben durch den Nutzer berücksichtigen. Für die Programmierung dieser Schnittstelle und die Konfiguration der LCD-Parameter bietet STM einen LCD-TFT Controller an, welcher über Register konfigurierbar ist [17].

Um die Konfiguration eines neuen Displays möglichst einfach zu gestalten, können die displayspezifischen Einstellungen in die Datei „lcd_conf.h“ eingetragen werden. Die Standardeinstellung sieht hier ein Display mit einer Breite von 800 Pixeln und einer Höhe von 480 Pixeln vor. Um Speicher für den Betrieb der Displayausgabe zu sparen, wird nur die linke obere Ecke des Displays benutzt. Der Bereich beschränkt sich auf die notwendige Größe für die Statusausgabe des Bootloaders. Die hardware-spezifischen Einstellungen wie beispielsweise die Synchronisation in horizontale und vertikale Richtung müssen dem Manual des entsprechenden Displays entnommen werden und ebenfalls zur Konfiguration hinzugefügt werden.

5.7 Sprung in die Hauptanwendung

Am Ende der Bootloader-Routine steht, wie in den Abbildungen 5.3 und 5.4 zu sehen ist, der „Sprung“ in die Hauptanwendung des eingebetteten Systems. Dies ist ein sehr heikler Schritt in der Entwicklung, da der Bootloader sämtliche beanspruchte Ressourcen freigeben muss. Zusätzlich muss dieser auch in der Lage sein, die Adressen auf die Hauptapplikation abzuändern. Vor allem das Debuggen gestaltet sich hier kompliziert, da der Programmraum des Bootloaders verlassen wird und der Debugger dies nicht mehr verfolgen kann. Um daher einen „Sprung“ in eine andere Anwendung durchzuführen, sind einige Schritte zu beachten. Dabei spielt sowohl

```
void jump_handler_init_vectors(void)
{
    __disable_irq();
    memmove(vectorTable_RAM, g_pfnVectors, 256 * sizeof(uint32_t));
    SCB->VTOR = APPLICATION_START_ADDRESS;
    __DSB();
    __ISB();
    __enable_irq();
}
```

Codeauszug 5.4 Anpassen des Offsets der Interrupt Vektortabelle

deren Reihenfolge als auch der Systemzustand zum Ausführungszeitraum eine wichtige Rolle. Im Folgenden werden nun diese Schritte erläutert und aufgezeigt, was deren Konsequenzen sind.

Zunächst gilt es, den Offset der Interrupt Vektortabelle von dem des Bootloaders auf den der Hauptanwendung zu ändern [41]. In Codeauszug 5.4 ist zu sehen, wie dieser mittels des System Controll Block (SCB), welcher ein Teil des NVIC ist [42], und des dafür vorgesehenen Registers umgesetzt wird. Wichtig ist es dabei, sicherzustellen, dass zu diesem Zeitpunkt alle Interrupts deaktiviert sind, um diesen Ablauf zu gewährleisten. Die Funktionen „__DSB()“ und „__ISB()“ stellen anschließend sicher, dass alle ausstehenden Speichertransaktionen abgeschlossen sind und dieser Abschluss von allen beteiligten Systemkomponenten erkannt wurde [41]. Erst nach dieser Synchronisation aller Komponenten können die Interrupts wieder aktiviert werden.

Als letzte Funktion des Programmflusses ruft der Bootloader die in Codeauszug 5.5 dargestellte Sprungfunktion auf. In dieser werden zunächst ebenfalls alle Interrupts ein letztes Mal deaktiviert. Anschließend werden alle verwendeten Komponenten freigegeben.

```
void jump_handler_start_app(void)
{
    __disable_irq();
    crc_deinit();
    HAL_RCC_DeInit();
    [ ... ]           // Deinitialisiere alle verwendeten GPIO-Pins
    [ ... ]           // Deaktiviere alle verwendeten Timer
    HAL_DeInit();
    for(uint8_t i = 0; i < 8; i++) NVIC->ICER[ i ] = 0xFFFFFFFF;
    for(uint8_t i = 0; i < 8; i++) NVIC->ICPR[ i ] = 0xFFFFFFFF;
    __set_CONTROL(0);
    __set_MSP(*(_IO uint32_t *)APPLICATION_START_ADDRESS);
    uint32_t jump_address = *((volatile uint32_t *)
                           (APPLICATION_START_ADDRESS + 4));
    __ISB();          // Warten auf Abschluss ausstehender Speichertransaktionen
    __DSB();          // Warten auf Synchronisation aller Beteiligten
    SysTick->CTRL &= ~(SysTick_CTRL_CLKSOURCE_Msk
                        | SysTick_CTRL_TICKINT_Msk
                        | SysTick_CTRL_ENABLE_Msk);
    void (*reset_handler)(void) = (void *)jump_address;
    while(1)
    {
        reset_handler();
    }
}
```

Codeauszug 5.5 Vorbereitung und Sprung in die Hauptanwendung

5 Implementierung des Bootloaders

Besonders wichtig ist die Freigabe aller Interrupts, welche in der Bootloader-Routine verwendet wurden. Um dies zu bewerkstelligen, stellt der NVIC zwei Register zur Verfügung. Das Interrupt clear-enable Register (ICER) beinhaltet 32 Interrupts. Um daher alle 256 möglichen Systeminterrupts zu deaktivieren, enthält der ARM® Cortex-M4® daher acht dieser ICER. Jeder Eintrag in diesen steht dabei mit einem booleschen Ausdruck für einen Interrupt. Um alle zu deaktivieren, müssen alle Einträge der acht ICER auf null gesetzt werden. Selbiger Aufbau gilt auch für die Interrupt set-pending Register (ISPR). Diese Register enthalten alle aktuell bevorstehenden Interrupts. Somit müssen zusätzlich alle ISPR auf null gesetzt werden, um neben der Deaktivierung der Interrupts auch die noch laufende Abarbeitung ausstehender Interrupts zu beenden [41]. Abschließend gilt es, alle Prioritätslevel der Interrupts zurück auf null zu bringen. Dies entspricht dem initialen Zustand nach einem Systemstart [42] und wird durch die Funktion „`__set_CONTROL(0)`“ erreicht.

Nachdem nun fast alle Ressourcen des Bootloaders freigegeben wurden, folgt im nächsten Schritt die Anpassung der Vektortabelle. Diese Tabelle ist der erste Teil des kompilierten Programms und liegt somit auch im Flash-Speicher des Mikrocontrollers auf den ersten Adressen des Anwendungsspeicherbereiches. Der initiale Stack-Pointer (SP) der Hauptanwendung belegt dabei die ersten vier Byte der Hauptanwendung. Der Reset-Vektor liegt auf den darauffolgenden vier Byte [41]. Um eine Anwendung zu starten, muss zunächst der SP der MCU auf den Initialen SP der Anwendung zeigen. Zusätzlich muss der Programm-Counter (PC) auf die nachfolgende Adresse gesetzt werden. Durch das Umsetzen des PC auf den Reset-Handler der Anwendung wird abschließend die Ausführung der Anwendung gestartet. Dies ist möglich, da in diesem die Startadresse der Anwendung gespeichert ist und durch diesen aufgerufen werden kann [6]. Zusätzlich wird der Hauptanwendung mit Hilfe dieses Vorgehens ein Systemreset vorgespielt [8]. Aus diesen beiden Gründen benötigt die Anwendung nur geringe Änderungen, um durch den Bootloader ausführbar zu sein [6, 8].

Bevor der PC jedoch umgesetzt werden kann, muss zunächst noch die Interrupt gesteuerte Systemuhr des Bootloaders deaktiviert werden. Dieser Timer mit dem Namen Systick ist ebenfalls ein Teil des NVIC. Da alle Cortex-M4®-Anwendungen denselben Systick-Timer verwenden [42], muss dieser für den Bootloader extra deaktiviert werden, um Probleme beim Start der Anwendung zu verhindern.

Der Bootloader ist durch diese Schritte dazu in der Lage, jegliche Anwendung im Flash-Speicher zu starten. Für eine solche müssen lediglich drei Änderungen an der Standardkonfiguration geändert werden. Zunächst muss die Startadresse der Anwendung, wie in den Codeauszügen 5.1 und 5.2 gezeigt, angepasst werden. Anschließend muss auch in der Hauptanwendung der Basis Offset der Vektortabelle angepasst werden. Als Letztes muss sichergestellt werden, dass als erste Handlung der Main-Funktion die Funktion „`__enable_irq()`“ aufgerufen wird. Mit dieser wird die Möglichkeit zur Aktivierung von Interrupts wieder hergestellt.

6 Upgrade des Bootloaders

Eine wesentliche Anforderung zur Entwicklung des Bootloaders ist ein Upgrade-Mechanismus für den Bootloader selbst. Dies ist ein sehr heikles Unterfangen, da etwaige Fehler während dieses Prozesses zu einem funktionsunfähigen System führen [8]. Um mögliche Fehlerquellen auszuräumen, wurde daher von einem direkten Upgrade-Mechanismus innerhalb der Bootloader-Software abgesehen. Folglich ergeben sich folgende zwei verbleibende Möglichkeiten, um den Bootloader mit Hilfe eines Software-Updates zu erneuern:

- Direktes Upgrade mittels angeschlossenem Computer
- Indirektes Upgrade mittels eigens entwickelter Firmware

Im nachfolgenden Kapitel werden nun beide Varianten vorgestellt. Ferner werden die individuellen Stärken und Schwächen des jeweiligen Vorgehens erläutert. Des Weiteren wird erklärt, warum beide Varianten notwendig sind und keine strikte Entscheidung für eine dieser Optionen getroffen wurde.

6.1 Direktes Upgrade

Die sicherste Variante, um den Bootloader auf dem laufenden System zu erneuern, ist ein Upgrade mit Hilfe eines angeschlossenen Computers. Dies liegt unter anderem an den folgenden Gründen. Zum einen besteht ein erhöhtes Risiko, dass während eines Upgrades etwas Unvorhersehbares wie beispielsweise ein Reset des Systems durch Einflüsse von außen eintritt [8]. Zum anderen kann ein fehlerhaftes Upgrade des Bootloaders sofort erkannt und direkt durch einen Neustart des Upgrades behoben werden. Tritt also im Verlauf des Upgrades ein Fehler auf, ist das System nicht zwangsläufig handlungsunfähig.

Zur Umsetzung dieser Methode wird jedoch eine direkte Verbindung zwischen einem Computer und dem eingebetteten System benötigt. Diese kann über die JTAG- oder SWD-Schnittstelle der MCU eingerichtet werden [24]. STM stellt zu diesem Zweck zwei kostenfreie Programme zur Verfügung. Zum einen die „ST-Link Utility“ [43], zum anderen den direkten Nachfolger mit dem Namen „STM32 Cube Programer“ [44]. Beide Werkzeuge unterstützen das für das Bootloaderupgrade verwendete Binär-Format und sind dazu in der Lage, gezielte Speicherbereiche des Systems zu löschen, zu beschreiben und zu validieren. Dies kann sowohl in Einzelschritten als auch innerhalb eines Vorgangs namens „ST-LINK Firmware Update“ angestoßen werden [43, 44]. Der „STM32 Cube Programer“ kann darüber hinaus

6 Upgrade des Bootloaders

auch auf den gängigen Betriebssystemen Windows, Linux und macOS installiert werden [44].

Die direkte Variante des Bootloaderupgrades bietet eine sichere Möglichkeit, etwaige Fehler im Bootloader-Code in einem Produktivsystem zu entfernen. Jedoch beinhaltet dies einen erhöhten Zeit- und Hardwareaufwand und es wird Fachpersonal benötigt, um das Upgrade durchzuführen. Ein weiterer Punkt, der gegen den Einsatz eines solchen Upgrade-Mechanismus spricht, ist die Zugangsmöglichkeit des Systems. Ist dieses schwer zugänglich oder im Umfeld der MCU wenig Platz für das benötigte Set-up, ist ein derartiges Update nur mit weit höherem Aufwand möglich. In diesem Fall bietet sich der Einsatz des im nachfolgenden Kapitel vorgestellten Verfahrens an. Dies ist ebenfalls der Grund dafür, warum für das Upgrade der Bootloader-Software zwei Varianten angeboten werden.

6.2 Indirektes Upgrade

In Kapitel 6.1 wird eine sichere Variante des Bootloaderupgrades erläutert. Jedoch werden ebenfalls Szenarien genannt, welche den vorgestellten Mechanismus erschweren oder verhindern. Aus diesem Grund wurde ein weiterer Upgrade-Mechanismus entworfen, welcher zum Einsatz kommt, falls kein direktes Upgrade möglich ist. Dieser kann jedoch nur auf einem System mit bereits installiertem Bootloader verwendet werden. Dieser Umstand ist der extra für diesen Zweck konzipierten Firmware verschuldet, die dieses Upgrade aus dem Speicher der Hauptanwendung durchführt.

Ein großes Problem, welches in Verbindung mit dem Upgrade eines Bootloaders steht, ist die Sicherheit des Prozesses. Führt beispielsweise während dieses Prozesses ein äußerer Einfluss zu einem Abbruch, fällt das System in einen undefinierten Zustand. In diesem ist kein funktionierender Bootloader im System, weshalb auch die Hauptanwendung nicht mehr gestartet werden kann [8].

Dieses Risiko lässt sich softwareseitig jedoch nicht eingrenzen. Um zu verhindern, dass während eines Upgrade-Prozesses äußere Einflüsse das System behindern, müssen vor Ort entsprechende Vorkehrungen getroffen werden. Um jedoch die Notwendigkeit eines Upgrades zu minimieren, wurde für den Prozess des indirekten Upgrades davon abgesehen, diesen Mechanismus in den Bootloader zu integrieren. Dies hätte zu einer erhöhten Komplexität des Codes geführt und zwangsläufig zu einer höheren Fehlerwahrscheinlichkeit [22]. Wie dieser Mechanismus funktioniert und welche Schritte bei der Implementierung zu beachten sind, wird in den beiden nachfolgenden Kapiteln beschrieben.

6.2.1 Ablauf

Abbildung 6.1 zeigt einen Zustandsautomaten, der den Ablauf des Upgrade-Mechanismus darstellt. Hier ist ersichtlich, dass ein solches Upgrade lediglich über die USB-Schnittstelle des eingebetteten Systems möglich ist. Dieser Schritt wurde aufgrund der Verbindungsstabilität unternommen. Der Bootloader unterstützt neben USB eine Ethernet-Schnittstelle mit TFTP. Da dieses Protokoll jedoch UDP-basiert ist und im Zusammenhang mit dem Bootloader keine Fehler während des Upgrades passieren dürfen, wurde diese Schnittstelle für diesen Zweck ausgenommen.

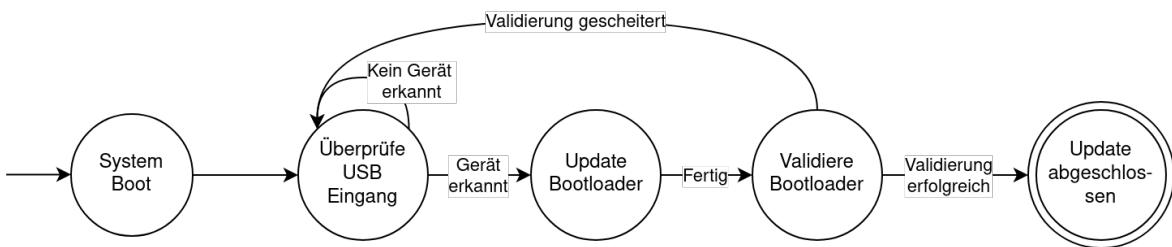


Abbildung 6.1 Zustandsautomat des indirekten Bootloaderupgrades

Quelle: Eigene Darstellung

Der allgemeine Upgrade-Prozess orientiert sich an dem Ablauf, welchen der Bootloader auch für das Firmware-Update anwendet. So wird zunächst die USB-Schnittstelle auf ein neues Gerät überprüft. Anschließend wird der Inhalt des Gerätes auf eine entsprechende Upgrade-Datei mit dem Namen „bootloader.bin“ untersucht. Nur wenn diese Datei gefunden wird, wird der Upgrade-Prozess gestartet. Hierfür wird zunächst der alte Bootloader gelöscht und anschließend der neue Bootloader an dessen Stelle gespeichert. Im nächsten Schritt wird die installierte Software wie im Fall eines Firmware-Updates mittels CRC verifiziert. Scheitert dies, wird der Prozess neu gestartet. In diesem Fall darf auf keinen Fall ein Neustart des Systems angestoßen werden, da aktuell eine fehlerhafte Bootloader-Software im Speicher liegt, die im schlimmsten Fall nicht hochfahren kann. Ist die Verifizierung des Upgrade-Prozesses erfolgreich, gibt das System ein visuelles Feedback und wartet auf einen Neustart des Systems.

Um zu verhindern, dass nach einem Neustart des Systems ein erneutes Upgrade des Bootloaders angestoßen wird, wurde die Upgrade-Firmware mit einer speziellen Versionsnummer versehen. Zu Beginn der Bootloader-Routine wird der Speicher der Anwendung überprüft. Wird diese Versionsnummer erkannt, erfasst der Bootloader die Software und entfernt die Hauptanwendung aus dem Speicher.

Da der Bootloader nun erfolgreich erneuert wurde, kann nun wieder die aktuelle Version der Hauptapplikation installiert werden. Der Flash-Bereich, welcher für permanente Variablen vorgesehen ist, wird während des indirekten Bootloaderupgrades nicht verändert. Daher ist eine nun wieder im System installierte Hauptanwendung in der Lage, entsprechend hinterlegte Einstellungen weiter zu verwenden.

6.2.2 Implementierung

Für die Implementierung der Firmware für das indirekte Bootloaderupgrade wurden zwei Projekte erstellt. Dies liegt an der Konfigurationsoption bezüglich der Backup-Funktion. Diese Implementierungen unterscheiden sich jedoch nur im Hinblick auf die Startadresse der Anwendung im Flash-Speicher und den resultierenden Offset der Vektortabelle zur Basis. Der Bootloader selbst ist in der Lage, diese spezielle Software anhand der Versionsnummer zu erkennen und legt in keinem Fall ein Backup dieser Applikation an. Die im Folgenden vorgestellten Funktionen sind für beide Projekte gleich.

Die Anwendung wurde nach dem Vorbild des Zustandsautomaten aus Abbildung 6.1 konzipiert. Diese Umsetzung ist in Codeauszug 6.1 dargestellt. Hier ist auch der in Kapitel 6.2.1 erläuterte Ablauf zu erkennen.

```
while (1)
{
    MX_USB_HOST_Process();

    switch(bootloader_upgrade . upgrade_state)
    {
        case CHECK_FOR_USB:
            if (usb_update_connect(&FatFs , MAX_ATTEMPT))
            {
                usb_check_update_version ();
            }
            break;
        case UPGRADE:
            usb_update ();
            break;
        case VALIDATE:
            flash_manager_validate_bootloader ();
            break;
        case FINISHED:
            // Upgrade abgeschlossen
            break;
    }
}
```

Codeauszug 6.1 Implementierung des Zustandsautomaten für das Bootloaderupgrade

Für diese Implementierung wurde ein Großteil der Programmteile des Bootloaders im Hinblick auf den USB-Update-Mechanismus wiederverwendet. Neben der Anpassung des Ziel-Speicherbereiches und der Entfernung der Sprungfunktion wurde lediglich der visuelle Feedback-Mechanismus angepasst. Dabei wurde zunächst auf den Einsatz eines LC-Displays für ein besseres Nutzer-Feedback aus Kompatibilitätsgründen verzichtet. Die Firmware muss auf allen möglichen Konfigurationen des Bootloaders lauffähig sein. Daher wird nur auf den Funktionsumfang

6.2 Indirektes Upgrade

der minimalen Konfiguration zurückgegriffen. Als Resultat beschränkt sich der Feedback-Mechanismus auf den Einsatz der drei integrierten LEDs mit den Farben Blau, Grün und Rot.

Angepasst wurde die Funktion der roten LED. Diese indiziert nun den Erfolg des durchgeföhrten Bootloaderupgrades. Die blaue LED steht weiterhin für die erfolgreiche Verbindung mit dem USB-Speichermedium. Die grüne LED wird wiederum während des Speichervorganges für jeden Speicherschritt umgeschaltet.

Leuchten alle drei LEDs, ist das Upgrade erfolgreich abgeschlossen. Nun kann das Speichermedium entfernt werden und ein Neustart des Systems angestoßen werden. Im Zuge dessen wird die Upgrade-Firmware anhand der Versionsnummer im Speicher erkannt und die Firmware gelöscht. Ist der Bootloader mit einem Backup-Mechanismus konfiguriert, wird anschließend direkt das Backup in den Speicher der Hauptanwendung geladen und das System ist wieder in vollem Umfang einsatzbereit. Ist kein derartiger Mechanismus konfiguriert, muss nun ein Update der Firmware mit der neusten Version mit Hilfe des erneuerten Bootloaders durchgeföhrt werden.

7 Anpassung der Implementierungsdaten

Um die Implementierung des Bootloaders in einem Produktivsystem zu verwenden, müssen zwei wichtige Aspekte beachtet werden. Zunächst gilt es, die Binärdateien für Update und Upgrade der entsprechenden Software anzupassen. Die hierfür notwendigen Schritte sind in Kapitel 7.1 beschrieben. Zusätzlich muss geklärt werden, in welchem Umfang und von wem die Software in Zukunft verwendet werden darf. Hierfür wurde der Bootloader mit einer Open-Source-Lizenz ausgestattet. Was hierbei zu beachten ist und welche Lizenz verwendet wurde, findet sich in Kapitel 7.2.

7.1 Build-Prozess der Binärdateien

Bevor eine Datei in den Flash-Speicher des eingebetteten Systems geschrieben werden kann, gilt es, die entsprechende Binärdatei um einige Informationen zu erweitern. Insgesamt handelt es sich dabei um drei Kenngrößen. Diese sind zum einen notwendig, um die jeweilige Software zu versionieren und einem Gerät zuzuweisen. Zum anderen müssen Informationen für die Verifizierung der Software beigelegt werden, um deren Sicherheit und die des Systems gewährleisten zu können. In den folgenden beiden Kapiteln werden nun die Schritte erläutert, welche für die Binärdateien bei Firmwareupdate und der Sonderfirmware für ein indirektes Bootloaderupgrade getroffen werden, um diese für das System verwendbar zu machen.

7.1.1 Shell-Skript zur Anpassung der Firmware-Updatedatei

Um die Binärdatei mit den notwendigen Informationen anzureichern, wurde ein Bash-Skript mit dem Namen „finishBinary.sh“ angelegt, welches diesen Prozess wie, in Abbildung 7.1 zu sehen, automatisiert. Dieses Skript benötigt drei Parameter für den Aufruf. In Tabelle 7.1 sind diese aufgelistet.

Der erste Parameter beinhaltet die Binärdatei mit der neuen Version der Hauptanwendung. Diese findet sich, wie in Abbildung 7.1 ersichtlich, nach dem Kompilieren im entsprechenden Projekt dieser Anwendung. Der Name „update.bin“ ist hier nicht vorgeschrieben und kann auch abgeändert werden. Das Skript ist in der Lage, den Namen der Binärdatei selbst festzustellen und die resultierende Datei entsprechend zu benennen. Wird der Name jedoch geändert, so muss dies in der Datei „bootloader.h“ des Bootloaders vor dessen Kompilierung angepasst werden. Der zweite

7 Anpassung der Implementierungsdaten

Tabelle 7.1 Parameter für den Aufruf des Shell-Skripts der Firmware-Updatedatei

Parameter	Inhalt	Datei	Projekt
\$1	Neue Firmware	update.bin	Hauptanwendung
\$2	DEVICE_SEED	bootloader.h	Bootloader
\$3	Versionsnummer	main.h	Hauptanwendung

Parameter beinhaltet die Gerätekennung mit dem Namen „DEVICE_SEED“. Diese Kennung wird wie in Kapitel 5.5 beschrieben, in der Konfiguration des Bootloaders festgelegt. Um ein Update über das Gerät mit diesem Bootloader installieren zu können, muss die entsprechende Kennung zum Abgleich an die Binärdatei angeknüpft werden. Diese findet sich in der Konfigurationsdatei des Bootloaders mit dem Namen „bootloader.h“. Der dritte Parameter legt die Version der neuen Software fest. Wie aus Abbildung 7.1 hervorgeht, wird diese in der Datei „main.h“ der Hauptanwendung festgelegt. Während des Update-Prozesses wird die Version mit der installierten verglichen und nur bei einem Unterschied wird die Aktualisierung durchgeführt.

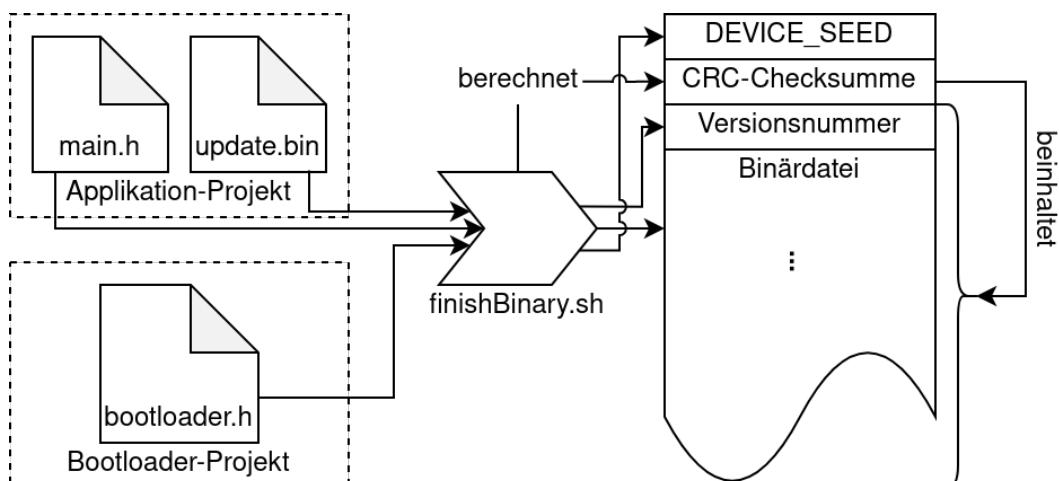


Abbildung 7.1 Zusammensetzung der Updatedatei für die Applikation
Quelle: Eigene Darstellung

Im ersten Schritt des Skriptes wird die Versionsnummer der Anwendung aus der Header-Datei eingelesen. Anschließend wird diese an den Anfang der Binärdatei gesetzt. Falls die Versionsnummer ein leerer String ist, wird dieser als Version null interpretiert.

Der nächste Eintrag, welcher mit der Updatedatei verknüpft wird, ist die CRC-Checksumme der Binärdatei. Diese wird im 32 Bit-Format über den aktuellen Stand der Updatedatei gebildet. Das heißt, die Checksumme beinhaltet sowohl die Versi-

onsnummer als auch die Binärdatei. Die berechnete Summe wird vor der Versionsnummer abgespeichert.

Im letzten Schritt wird die „DEVICE_SEED“ eingelesen und wiederum vor der CRC-Summe an den Anfang der Updatedatei gesetzt. Dieser Parameter wird nicht in der Checksumme berücksichtigt, da die Updatedatei für mehrere Geräte verfügbar sein soll. In diesem Fall kann die fertige Datei so angepasst werden, dass lediglich die Gerätekennung an das entsprechende Gerät angepasst werden muss. Der Rest der Datei ist geräteunabhängig für die aktuelle Version verwendbar. Die drei Datensätze, welche durch das Skript an die Datei angehängt werden, sind jeweils 32 Bit groß. Somit wird die Datei um 12 Byte verlängert. Der Parser des Bootloaders ist nur in diesem Format in der Lage, die Informationen korrekt zu erfassen.

7.1.2 Shell-Skript zur Anpassung der Sonderfirmware-Updatedatei für das indirekte Bootloaderupgrade

In Kapitel 5.5 wurde beschrieben, welche Maßnahmen vor der Kompilierung der Bootloader-Software möglich sind. Dieses Ergebnis kann nun über drei Wege in das eingebettete System geladen werden. So kann die Datei zum Beispiel direkt über die STM32CubeIDE auf das System übertragen werden. Für ein Produktivsystem gibt es darüber hinaus die beiden in Kapitel 6 vorgestellten Methoden.

Ist ein direktes Upgrade der Bootloader-Software möglich, übernimmt die Übertragungsanwendung sämtliche notwendige Schritte um die neue Bootloader-Version in den Speicher zu laden und die Übertragung zu verifizieren. Ist dies jedoch nicht möglich, so muss eine spezielle Firmware an die Stelle der Hauptanwendung geladen werden. Diese übernimmt anschließend das Upgrade des Bootloaders. Um sicherzustellen, dass diese Sonderfirmware korrekt in den Speicher geladen wird, muss deren Binärdatei entsprechend angepasst werden. Dazu ist ein ähnlicher Prozess wie der in Kapitel 7.1.1 vorgestellte notwendig. Da es sich hier jedoch um einen Sonderfall handelt, gibt es einige Unterschiede zur in Kapitel 7.1.1 vorgestellten Variante für die Anwendungsdatei.

Tabelle 7.2 Parameter für den Aufruf des Shell-Skripts der Sonderfirmware des Bootloaderupgrades

Parameter	Inhalt	Datei	Projekt
\$1	Sonderfirmware für Bootloaderupgrade	update.bin	Sonderfirmware
\$2	DEVICE_SEED	bootloader.h	Bootloader

Genau wie für die Hauptanwendungsdatei verfügt auch die Sonderfirmware über ein Bash-Skript. Dieses Skript mit dem Namen „finishBinaryUpgradeBootloader.sh“

7 Anpassung der Implementierungsdaten

stellt die Upgradedatei mit Hilfe der in Tabelle 7.2 dargestellten Parameter fertig. Über den ersten Parameter erhält das Skript die Binärdatei der Sonderfirmware. Hierbei gilt zu beachten, dass abhängig von der Konfiguration des Bootloaders hinsichtlich der Backup-Funktion zwei unterschiedliche Projekte bestehen. Aus diesen gilt es die passende Option zu wählen und deren Binärdatei an das Bash-Skript zu kommunizieren. Der zweite Parameter steht für die Gerätekennung. Diese wird im Bootloader im Zuge der Konfiguration festgelegt. Mit dieser Information kann der Bootloader feststellen, ob die bereitgestellte Firmware für dieses Gerät gedacht ist.

Der Bootloader ist in der Lage, die Sonderfirmware anhand einer gesonderten Versionsnummer zu erkennen. Diese Nummer wird im ersten Schritt des Skriptes an den Anfang der Binärdatei gestellt. Im nächsten Schritt wird über diese beiden Teile eine CRC-C checksumme gebildet. Diese wird vor der Versionsnummer angefügt und dient dem Bootloader zur Verifizierung des Datensatzes. Als Letztes wird noch die Gerätekennung vorne an die Datei angeknüpft. Warum diese nicht in die Checksumme inkludiert ist, wird in Kapitel 7.1.1 erläutert.

Die fertiggestellte Datei kann nun wie eine normale Firmware an das Gerät mit installiertem Bootloader kommuniziert werden. Der weitere Verlauf des Upgrade-Prozesses wird in Kapitel 6.2.1 beschrieben. Hier findet sich auch, wie diese Sonderfirmware wieder von dem System entfernt wird und die Hauptanwendung wiederhergestellt wird.

7.2 Lizenzierung

Eine wichtige Anforderung an das Projekt ist dessen Veröffentlichung unter einer Open-Source-Lizenz. Mit diesem Schritt wird zum einen Entwicklern die Erweiterung und Verbesserung des Projektes für die Zukunft ermöglicht. Zum anderen entsteht die Möglichkeit, das Projekt auch öffentlich in weiteren Projekten zu nutzen. Dadurch soll ein kontinuierlicher Verbesserungs- und Anpassungsprozess an zukünftige und individuelle Gegebenheiten für das Projekt angeregt werden [45]. Im Mittelpunkt dieser Entscheidung steht jedoch der derzeitige Mangel an öffentlichen Bootloader-Implementierungen im Bereich der eingebetteten Systeme.

Neben der Art der Veröffentlichung bietet eine Lizenz auch weiteren Schutz bezüglich des Urheberrechtes und der Verwendung durch Dritte. So ist der Urheber der Software nicht haftbar für etwaige Probleme in der Anwendung dieser Software durch Dritte [46]. Die Lizenz regelt auch die kommerzielle Nutzung der Software und schreibt diesbezüglich klare Richtlinien vor [47].

Da es mehrere mögliche Lizenzen für die Veröffentlichung als Open-Source-Software (OSS) gibt, wird im folgenden Kapitel eine Übersicht über gängige Modelle gegeben, welche für dieses Projekt infrage kommen. In Kapitel 7.2.2 wird anschließend der Entscheidungsprozess dargestellt.

7.2.1 Überblick Open-Source-Lizenzierung

Um festzulegen, ob eine Lizenz die Kriterien für Open-Source erfüllt, wurde die Open-Source-Initiative (OSI) gegründet. Diese befasst sich unter anderem mit der Prüfung, Freigabe und Ablehnung von eingereichten Vorschlägen für ein neues Open-Source-Lizenzzmodell [48]. Zusätzlich stellt die OSI bereits freigegebene Lizenzen öffentlich zur Verfügung und gibt eine ausführliche Beschreibung dieser Lizenzen an [45].

Eine Lizenz kann von der OSI nur als OSS-Lizenz anerkannt werden, sofern diese die folgenden zehn Kriterien erfüllt [49]:

1. Uneingeschränkte Weiterverteilung
2. Beinhaltung von Quellcode im Projekt
3. Erlauben von Änderungen und Weitergabe unter Ursprungslizenz
4. Option auf Verbot der Weitergabe in abgeänderter Form
5. Keine Diskriminierung von Personen oder Gruppen
6. Keine Einschränkung der Nutzbarkeit auf bestimmte Fachgebiete
7. Verpflichtung zur Weiterverwendung der Lizenz
8. Keine spezifische Bindung von Lizenzen an Projekte
9. Keine Einschränkung parallel verwendetener Software
10. Technologische Neutralität

Ein stärkerer Ansatz für OSS findet sich wiederum unter dem Begriff der freien Software wieder. Dieser wird durch das GNU-Projekt definiert und fasst den Rahmen für entsprechende Software-Lizenzen enger. Für freie Software werden die folgenden vier Freiheiten vorausgesetzt, welche das Lizenzierungsmodell des jeweiligen Projektes erfüllen muss [50]:

- Uneingeschränkte Programmausführung
- Programmuntersuchung und Anpassung an eigene Bedürfnisse
- Weitergabe des Programms zur Unterstützung von Mitmenschen
- Verbesserung des Programms und dessen Veröffentlichung

Diese vier Freiheiten sollen verhindern, dass die Verwendung von Software durch Dritte ausgenutzt wird oder eine Verbesserung einer solchen Software behindert wird [51].

In der Regel entsprechen Lizenzen, welche die OSI freigibt, auch dem Konzept einer freien Software-Lizenz. Dennoch muss hier auf die oben gelisteten Kriterien geachtet werden, wenn ein Projekt nicht nur öffentlich einsehbar, sondern zwingend

7 Anpassung der Implementierungsdaten

auch frei verwendbar sein soll [51]. In diesem Zusammenhang spielt der Begriff „Copyleft“ eine wichtige Rolle. Dieser gibt an, inwieweit die Freiheiten einer OSS bei der Einbindung in ein anderes Projekt in dieses übernommen werden müssen [52]. Insgesamt gibt es drei mögliche „Copyleft“-Optionen. Zunächst das strenge „Copyleft“, welches die Einbindung in proprietäre Software verbietet. Auch eine abgeschwächte Form von „Copyleft“ ist möglich. Mit einer solchen ist die Einbindung in proprietären Code erlaubt. Jedoch darf in diesem Fall nur der eigene Code proprietär verwendet werden [53]. Eine dritte Variante ist die Lizenzierung ohne „Copyleft“. In diesem Fall darf die Software ohne Einschränkung in das Projekt eingefügt und verwendet werden [52].

7.2.2 Auswahlprozess

Das Ziel der Lizenzierung für das Bootloader-Projekt entspricht nahezu den durch das GNU-Projekt definierten Kriterien für freie Software. Diese sind in Kapitel 7.2.1 beschrieben. Das Bootloader-Projekt soll von diesen Kriterien profitieren und anhand der Lizenzvorgaben im Rahmen einer freien Software verbessert werden. Dadurch soll ein kontinuierlicher Verbesserungsprozess ermöglicht werden, welcher der Allgemeinheit zur Verfügung steht [50]. Aus diesem Grund beschränkt sich der Auswahlprozess der Lizenz auf die durch GNU geprüften Modelle. Diese sind alle auch durch die OSI als Open-Source lizenziert und erfüllen die Kriterien für freie Software. Innerhalb dieser Auswahl gilt es zu prüfen, welche Form des „Copyleft“ für das Bootloader-Projekt gewünscht ist. In diesem Fall darf die Einbindung in ein anderes Projekt erfolgen, was „Copyleft“ für den Auswahlprozess ausschließt [53].

Die Auswahl fällt daher auf eine BSD-Lizenz. Diese erfüllt die von der OSI festgelegten Open-Source-Kriterien und wurde auch von dieser freigegeben. Darüber hinaus beinhaltet diese Lizenz auch die vier Freiheiten einer freien Software [54]. Die einzige Einschränkung findet sich darin, dass diese Lizenz kein „Copyleft“ enthält [52].

Zu dieser Lizenz sind mehrere Versionen verfügbar. Die neueste Version davon ist die modifizierte BSD-Lizenz. Diese unterscheidet sich von den Vorgängern durch das Hinzufügen einer dritten Klausel. Diese untersagt, wie in Codeauszug 7.1 zu sehen, die Verwendung von Namen der Entwickler ohne schriftliche Erlaubnis. Um diese Lizenz zum Projekt hinzuzufügen, wurde der in Codeauszug 7.1 abgebildete Lizenztext an den Anfang jeder proprietären Datei des Projektes eingefügt. Der Text wurde von der offiziellen Quelle der OSI entnommen und an den vorgesehenen Stellen durch die entsprechenden Informationen zum Lizenzhalter ergänzt [55].

```
/*
 * Copyright (c) 2021 AN Dispensing UG (limited liability)
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above
 * copyright notice, this list of conditions and the following
 * disclaimer in the documentation and/or other materials provided
 * with the distribution.
 * 3. Neither the name of the copyright holder nor the names of its
 * contributors may be used to endorse or promote products derived
 * from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
 * CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
 * TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * Author: Thomas Randl <thomas.randl@andispensing.de>
 */

```

Codeauszug 7.1 3-Klausel-BSD-Lizenztext für das Bootloader-Projekt

8 Durchgeführte Tests am System

Als essenzieller Bestandteil eines Produktivsystems muss die Funktionalität eines Bootloaders sichergestellt werden. Ist ein solcher fehlerhaft, kann dies in unerwünschten Folgen für das System resultieren. Im schlimmsten Fall kann ein solches nicht mehr gestartet werden. Darüber hinaus muss der Bootloader auch in der Lage sein, sekundäre Funktionen fehlerfrei auszuführen. Für den im Zuge dieser Arbeit entwickelten Bootloader muss daher zusätzlich zur Bootfunktion des Systems die Update-Funktionalität sichergestellt werden.

Um die Funktionalität des Bootloaders sicherzustellen, wurden mehrere Schritte unternommen, um etwaige Fehlerquellen während und nach der Entwicklung zu erkennen und diese zu beheben. In den nachfolgenden Kapiteln werden dieses Prozesse erläutert. Darüber hinaus wird erklärt, welche Schritte für eine Portierung auf andere Hardware-Architekturen notwendig sind.

8.1 Debuggen mittels Entwicklungsumgebung und SWD-Schnittstelle

In Kapitel 5.1 wurde auf den Einsatz der STM32CubeIDE verwiesen. Dabei wurden deren Vorteile bezüglich der Entwicklung für eine MCU der Firma STM herausgearbeitet. Einer dieser Vorteile ist der umfangreiche Debugger, welcher durch diese Entwicklungsumgebung bereitgestellt wird.

Für ein Übertragen und Debuggen der Software auf das Zielsystem muss zunächst eine Verbindung zwischen Computer und MCU eingerichtet werden. Diese wird mittels einer SWD-Schnittstelle [56] und der Nutzung eines ST-Link Adapters hergestellt. Der Adapter ist bereits in das verwendete Nucleo-Board integriert, weshalb nur eine USB-Verbindung zwischen Computer und Board benötigt wird. Dieser kann durch eine Anpassung der Hardware-Brücken auch für die Kommunikation mit einem anderen Board verwendet werden. Näheres hierzu findet sich in Kapitel 8.4. Der ST-Link selbst wird in diesem Fall lediglich als Hardwarekomponente für die Kommunikation zwischen USB und SWD genutzt [57].

Der Funktionsumfang, welcher durch den Debugger der STM32CubeIDE bereitgestellt wird, erlaubt eine detaillierte Übersicht über das System zur Laufzeit. So kann die Programmausführung jederzeit unterbrochen werden, um den aktuellen Zustand einzusehen. Darüber hinaus können Inhalte von Variablen während der

8 Durchgeführte Tests am System

Ausführung überwacht werden. Um etwaige Fehlerfälle nachzuvollziehen, kann das Fehler-Analyse-Werkzeug der STM32CubeIDE verwendet werden. Dieses gibt Auskunft über die Register-Zustände zum Zeitpunkt des Fehlers. Ebenfalls kann jederzeit der Zustand der CPU und der Speicherinhalte überprüft werden [33].

Um diesen Funktionsumfang nutzen zu können, muss das Projekt für den Debug-Modus kompiliert werden [56]. Die Binär-Datei ist in diesem Modus im Fall einer maximalen Konfiguration größer als 64 KiB. Dies ist jedoch die umfangreichste Größe für den Bootloader im Speicher (vgl. Kapitel 5.2). Um mittels Debuggen etwaige Fehler zu finden und dadurch die Funktionalität des Bootloaders zu gewährleisten, muss daher für diesen Fall der Speicherbereich des Bootloaders vergrößert werden. Dieser Fall tritt allerdings nur in Verbindung mit der Konfiguration eines Backup-Mechanismus auf. In diesem Fall schließt jedoch der Speicherbereich der Anwendung direkt an den des Bootloaders an.

Aus diesem Grund konnte die maximale Konfiguration nur mit Einschränkungen getestet werden. Zur Fehlerfindung und Überprüfung der einzelnen Konfigurationen wurden diese daher separat getestet, um die Speichergrößen-Anforderungen zu erfüllen. Für den Test, ob die Systemkomponenten auch gemeinsam keine Probleme aufweisen, wurde der Speicherbereich des Bootloaders erweitert. Dadurch konnten alle aufgetretenen Fehler in allen Konfigurationsvarianten gefunden und korrigiert werden. Im Fall des erweiterten Speicherbereiches konnte jedoch nur das Setzen der korrekten Parameter für den in Kapitel 5.7 beschriebenen Sprung überprüft werden. Die korrekte Ausführung des Sprunges in die Hauptanwendung konnte jedoch durch die anderen Konfigurationen und die im folgenden Kapitel beschriebenen Tests am Produktivsystem nachgewiesen werden.

8.2 Durchgeführte Tests an der Release-Version

Offensichtliche Fehler während der Entwicklung einer neuen Bootloader-Software können durch das in Kapitel 8.1 beschriebene Debuggen behoben werden. Der Bootloader folgt jedoch einem strengen Ablaufplan, welcher durch einen der in Kapitel 5.3 abgebildeten Zustandsautomaten definiert wird. Für einen solchen Automaten können viele verschiedene Ablaufszenarien eintreten. Aus diesem Grund wurde zum Ende der Entwicklung ein Testprotokoll entwickelt, welches diese Abläufe abdeckt. Erst nach erfolgreichem Durchlauf dieses Protokolls, welches in Tabelle 8.1 zu sehen ist, kann die Bootloader-Version für den Einsatz freigegeben werden.

Da der Ablaufplan abhängig von der Konfiguration des Backup-Mechanismus ist, muss diese Trennung auch im Protokoll berücksichtigt werden. So wird die neue Version für beide Fälle gesondert kompiliert. Beide Kompilate werden im Anschluss auf Speicherfehler überprüft. Hierfür wurde ein Bash-Skript entwickelt, welches mit Hilfe des Analyseprogramms „Valgrind“ [58] alle eigens entwickelten Header- und Source-Dateien des Bootloaders überprüft. Ist dieser Test ohne Befund, kann der

8.2 Durchgeführte Tests an der Release-Version

Tabelle 8.1 Testprotokoll für den Release einer neuen Bootloader-Version

Schritt	Aufgabe	Bestanden?
01	Kompiliere Software mit maximaler Konfiguration ohne Backup-Mechanismus (NB)	[]
02	Überprüfe Dateien auf Memory-Leaks	[]
03	Kompiliere Software mit maximaler Konfiguration und Backup-Mechanismus (B)	[]
04	Überprüfe Dateien auf Memory-Leaks	[]
05	Komplettiere Binärdateien der Testanwendung mit Bash-Skript	[] []
06	Kopiere Vorgängerversionen der Testanwendung auf USB-Stick und TFTP-Server	[] [] [] []
07	Kopiere neue Versionen der Testanwendung auf USB-Stick und TFTP-Server	[] [] [] []
Gesonderte Tests für Versionen mit und ohne Backup-Mechanismus		
08	Lösche kompletten Flash-Speicher des Systems	NB: [] B: []
09	Lade passende Bootloader-Konfiguration in das System (NB, B)	NB: [] B: []
10	Update der Vorgängerversion der Testanwendung	NB: [] B: []
11	Update der neuen Version der Testanwendung	NB: [] B: []
12	Ausführlicher Funktionstest der Testanwendung	NB: [] B: []
13	Ändere Versionskennung der Testanwendung im Speicher (Neustart: NB: Lösche Anwendung + Update B: Lade Backup)	NB: [] B: []
14	Kompromittiere Testanwendung im Speicher (Neustart: NB: Lösche Anwendung + Update B: Lade Backup)	NB: [] B: []
Erweiterte Tests für Backup-Mechanismus		
15	Kompromittiere Hauptanwendung + Backup	[] []
16	Neustart: Lösche Hauptanwendung und Backup + Update	[]

eigentliche Systemtest des Bootloaders beginnen.

Zu diesem Zweck werden Binärdateien einer Testanwendung abhängig von deren Konfigurationsanforderung bezüglich des Backup-Mechanismus mit dem in Kapitel 7.1.1 vorgestellten Skript fertiggestellt. Diese Binärdateien werden anschließend auf einen USB-Stick und den TFTP-Server geladen. Zum Abschluss der Vorbereitungen werden beide Schnittstellen mit dem eingebetteten System verbunden.

Die Schritte 8 bis 14 des Testprotokolls aus Tabelle 8.1 sind nun jeweils für die beiden Konfigurationen auszuführen. So wird das System zunächst durch das Löschen des Speichers zurückgesetzt. Anschließend wird die zu testende Bootloader-Konfiguration in das System geladen. Nun wird das System gestartet und die alte Version der Testfirmware mit Hilfe des Bootloaders über eine der Schnittstellen in den Speicher geladen. Im nächsten Schritt wird diese durch die neuere Version, welche über die Schnittstellen verfügbar ist, ausgetauscht.

8 Durchgeführte Tests am System

Um sicherzustellen, dass der Bootloader vor dem Sprung in die Anwendung alle Ressourcen freigibt, wird nun ein ausführlicher Test der Hauptanwendung durchgeführt. Zu diesem Zweck wurde eine Testanwendung entwickelt, welche wichtige Systemfunktionen nutzt. Diese wird mit einem FreeBSD-Betriebssystem [59] und Multi-threading betrieben. Dabei wird ein Display angesteuert, welche auf Touch-Eingaben reagiert. Durch die Verwendung dieser Komponenten wird ein breites Spektrum an Funktionen des ARM® Cortex-M4®-Prozessors genutzt. Zusätzlich benötigt die Anwendung mehr als 128 KiB an Speicher, weshalb auch die Speicherlösungs- und Beschreibungsfunktionen des Bootloaders im Zuge des Testprotokolls überprüft werden.

Start des Bootloaders	0x08000000	20030000	08001E19	080016FD	080016FF
	0x08000010	08001701	08001703	08001705	00000000
	0x08000020	00000000	00000000	00000000	08001707
	0x08000030	08001709	00000000	0800170B	0800170D
	[...]				
Version = 100 (1.0)	0x08010000	00000064	FFFFFFFF	FFFFFFFD	FFFFFFFD
	[...]				
Firmware ist Valide	0x08010100	A5A5A5A5	FFFFFFFD	FFFFFFFD	FFFFFFFD
	[...]				
Start der Firmware	0x08010200	10010000	08011AE5	080117F1	08011891
	0x08010210	080118B3	080118B9	080118BF	00000000
	0x08010220	00000000	00000000	00000000	08014C31
	0x08010230	080118C5	00000000	08014E71	080118D3
	[...]				
Permanente Variablen der Firmware	0x08100000	D00FAFF3	00000041	FFFFFFFD	FFFFFFFD
	[...]				
Version des Backups = 100 (1.0)	0x08110000	00000064	FFFFFFFD	FFFFFFFD	FFFFFFFD
	[...]				
CRC-Checksumme des Backups	0x08110100	08ED6447	FFFFFFFD	FFFFFFFD	FFFFFFFD
	[...]				
Länge des Backups in Byte	0x081101B0	0001C21C	FFFFFFFD	FFFFFFFD	FFFFFFFD
	[...]				
Start des Backups	0x08110200	10010000	08011AE5	080117F1	08011891
	0x08110210	080118B3	080118B9	080118BF	00000000
	0x08110220	00000000	00000000	00000000	08014C31
	0x08110230	080118C5	00000000	08014E71	080118D3
	[...]				

Abbildung 8.1 Speicherbelegung im Produktivsystem
Quelle: Eigene Darstellung

In den nächsten drei Schritten werden nun Fehlerfälle geprüft, welche im Bootloader durch eine gescheiterte Verifizierung erkannt werden müssen. So wird durch wiederholte Speichermanipulation der Inhalt oder die Parameter der Hauptanwendung verändert. Im Protokoll sind die erwarteten Ergebnisse dieser Vorgänge hinterlegt. Nur wenn diese eintreten, hat die Bootloader-Version den jeweiligen Test bestanden.

Wird am Ende aller vorgeschriebenen Test kein Befund an der Bootloader-Version erkannt, so kann diese freigegeben werden. In diesem Fall ist der Speicher wie in Abbildung 8.1 ersichtlich belegt. Diese zeigt die Speicherbelegung für den Einsatz der Testanwendung mit Backup-Mechanismus. Im Flash-Speicher des Systems finden sich nach erfolgreicher Verifizierung alle notwendigen Informationen bezüglich der Hauptanwendung. Zusätzlich ist ersichtlich, wo sich die Vektoren für den wie in Kapitel 5.7 beschriebenen Sprung in die Anwendung befinden und was deren Inhalt ist.

Tritt jedoch während des Testprotokolls ein Fehler auf, ist dieser zu dokumentieren. Dennoch sollte der gesamte Test durchgeführt werden, um etwaige weitere Fehler zu erkennen, um diese nicht erst im nächsten Testdurchlauf zu entdecken. Zur Fehleranalyse kann nun ebenfalls der Speicherinhalt herangezogen werden. Anhand dessen lässt sich erkennen, ob beispielsweise der CRC nicht erfolgreich war. In diesem Fall ist die Firmware nicht valide und der entsprechende Speicherinhalt bleibt leer.

8.3 Überprüfung der TFTP-Schnittstelle

Um die Hauptanwendung im Speicher zu aktualisieren, bietet die Bootloader-Implementierung zwei Möglichkeiten. Zum einen die Basisvariante mittels USB-Stick, zum anderen die Möglichkeit eines Updates über eine Ethernet-Verbindung. Für die zweite Option bietet sich, wie in Kapitel 4.2 erläutert, das TFTP an.

Das eingebettete System erfüllt für dieses Protokoll die Rolle des Clients. Um eine Kommunikation zu ermöglichen, wird daher ein Server benötigt, welcher die Update-datei auf Anfrage an den Client kommuniziert. Da die Bootloader-Implementierung unabhängig von der Server-Architektur ist, gibt es keine konkrete Vorgabe für einen derartigen Aufbau.

Um die TFTP-Schnittstelle zu testen und deren Funktionalität zu gewährleisten, wurde im Zuge der Entwicklung die in Abbildung 8.2 gezeigte Architektur verwendet. Hierbei wurde ein Raspberry Pi 3B+ mit einem Raspberry Pi OS [60] aufgesetzt. Dieser dient als Server für die Kommunikation und ist Linux-basiert. Die Kommunikation selbst wurde sowohl direkt zwischen Server und eingebetteten System getestet als auch über einen Router im Heimnetz.

8 Durchgeführte Tests am System

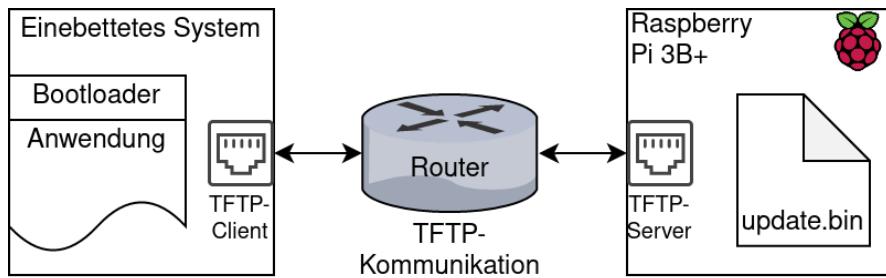


Abbildung 8.2 Testaufbau zur Überprüfung des TFTP-Update-Mechanismus
Quelle: Eigene Darstellung

Um den Server für die TFTP-Kommunikation vorzubereiten, wurden die Pakete „xinetd“ und „tftpd“ mittels des Paketverwaltungssystems „apt“ installiert. Das erste Paket stellt einen Server als Basis der Kommunikation bereit [61]. Diesem wird die Konfiguration aus Codeauszug 8.1 für den TFTP-Server übergeben. Dabei werden die Parameter des TFTP eingetragen und minimale Berechtigungen in Form der Nutzer-ID „nobody“ für den TFTP-Server vergeben. Das zweite Paket stellt wiederum diesen TFTP-Server bereit und wird von „xinetd“ gestartet [62]. In Codeauszug 8.1 ist darüber hinaus der Ordner „/opt/tftpboot“ angeben, in welchem die Updatedatei hinterlegt wird. Dieser ist nun dem Server bekannt und bei etwaigen TFTP-Anfragen wird dieses Verzeichnis nach der angefragten Datei durchsucht.

```
protocol      = udp
port         = 69
flags        = IPv4
socket_type  = dgram
wait         = yes
user         = nobody
server       = /usr/sbin/in.tftpd
server_args  = /opt/tftpboot
disable      = no
```

Codeauszug 8.1 Konfiguration der Datei „/etc/xinet.d/tftp“

Im nächsten Schritt wurde die WIFI-Schnittstelle des Raspberry Pi deaktiviert. Zusätzlich wurde dem Gerät eine statische IPv4-Adresse zugewiesen. Diese wird benötigt, um diese als Zieladresse in der Bootloaderkonfiguration zu hinterlegen. Der Server wiederum muss die IP-Adresse des Clients nicht kennen, da dieser nur den passiven Verbindungspart übernimmt und daher die Adresse des anfragenden Gerätes verwenden kann. Mit Hilfe dieses Servers konnte eine erfolgreiche Kommunikation eingerichtet werden und der Bootloader ist in der Lage, Updates über die TFTP-Schnittstelle durchzuführen. Der korrekte Ablauf dieses Vorgangs wurde mit Hilfe des Programms „Wireshark“ [63] getestet. Ein Ausschnitt eines solchen Tests findet sich in Abbildung 5.7 in Kapitel 5.4.2.

Im Falle eines Verbindungsabbruches während der Übertragung löst der Bootloader einen Timeout aus. Da der Speicherinhalt der Hauptanwendung nun fehlerhaft ist, wird deren Speicherbereich gelöscht. Anschließend wird je nach Konfiguration ein Backup eingespielt. Ist diese Option nicht verfügbar, wird das System, wie in Abbildung 5.3 zu sehen, in einen Wartezustand versetzt. In diesem verweilt der Bootloader, bis eine neue Updatedatei über eine der Schnittstellen verfügbar ist.

8.4 Portierung auf kompatible Hardware

Ein weiterer wichtiger Punkt, den der Bootloader mit sich bringen soll, ist die Portierbarkeit auf andere Systeme. Dies ist ein Prozess, welcher im Bereich der eingebetteten Systeme stark von der eingesetzten Hardware abhängig ist. Aus diesem Grund lassen sich Änderungen an den Hardwareeinstellungen nicht vermeiden und müssen entsprechend auf das jeweilige Zielsystem angepasst werden.

Ein wichtiger Aspekt für die Portierbarkeit des im Zuge dieser Arbeit entwickelten Bootloaders ist die Verfügbarkeit eines minimalen Hardwaresets. So muss zumindest folgende Hardware verfügbar sein, welche für die minimale Konfiguration des Bootloaders benötigt wird:

- ARM® Cortex-M4®-Prozessor
- USB 2.0-Schnittstelle
- Drei getrennte LEDs
- CRC32 Berechnungseinheit
- ≥ 256 KiB Flash-Speicher

Im Zuge dieser Arbeit wurde die Portierbarkeit auf einer Reihe von speziellen Hardware-Platinen getestet, welche sich grob an der Architektur des NUCLEO-F429ZI Boards orientieren. Diese erfüllen mindestens die oben genannten minimalen Anforderungen. Die Tests zeigten, dass eine Portierbarkeit generell möglich ist. Jedoch hängt diese Möglichkeit und der damit verbundene Aufwand von der eingesetzten Hardware ab.

Im konkreten Fall stimmten die Pin-Belegungen der unterschiedlichen Architekturen aufgrund des gleichen Prozessors überein. Dadurch müssen nur wenige Anpassungen vorgenommen werden. Eine derartige Änderung musste im Hinblick auf den eingesetzten externen Quarz des Oszillators des Systems getroffen werden. Dieser ist Teil des Taktgebers [64] und weicht von dem des Entwicklungsbretts ab. Während das NUCLEO-F429ZI Board einen externen Oszillator mit einer Frequenz von 8 MHz verwendet [25], benutzt die Zielarchitektur einen entsprechenden Oszillator mit 12 MHz. Für zukünftige Portierungen kann diese Änderung über die Konfiguration in der Datei „bootloader.h“ des Bootloader-Projektes angepasst werden.

8 Durchgeführte Tests am System

Im Zuge dieser Arbeit wurden keine weiteren Portierungen vorgenommen. Das oben beschriebene Vorgehen beweist als „Proof of Concept“, dass dieser Schritt im Rahmen der genannten Anforderungen tendenziell möglich ist. Über den Aufwand der jeweiligen individuellen Architektur kann jedoch keine Einschätzung getroffen werden.

Bei der Portierung gilt es, die Verwendung des „Hardware Abstraction Layers“ zu beachten. Dies wird durch STM bereitgestellt und dient der vereinfachten Kommunikation mit den Hardwarekomponenten. Aus diesem Grund empfiehlt sich eine Portierung im Bereich der STM32-Architektur. Sollte die Hardware des Zielsystems von der des NUCLEO-F429ZI Boards abweichen, sind größere Änderungen an der Pinbelegung und den Einstellungen der entsprechenden Hardwarekomponenten in der Bootloader-Software notwendig. Es empfiehlt sich daher, die Hardware an die des NUCLEO-F429ZI Boards anzulegen.

9 Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Entwicklung eines Bootloaders zur Anwendung im industriellen Umfeld. Dieser muss zum einen über eine Update-Funktion für eine im System installierte Anwendung verfügen. Zum anderen benötigt dieser einen Upgrade-Mechanismus für den Loader selbst. Ein besonderer Augenmerk wurde dabei auf vielfältige Konfigurationsmöglichkeiten für Einsätze in einem breiten Spektrum an Anwendungsgebieten gelegt. Die Planung und Umsetzung dieser Konfigurationsoptionen bildet daher den Schwerpunkt der Arbeit.

Um dem Leser das Thema Loader zunächst näher zu bringen, befasst sich der erste Teil der Arbeit mit der zugehörigen Theorie und der Vorstellung unterschiedlicher Einsatzgebiete. Dabei wird herausgearbeitet, was ein Loader ist und welche Funktionen ein solcher grundsätzlich beinhaltet. Anschließend werden die unterschiedlichen Einsatzmöglichkeiten anhand der zwei verbreitetsten Einsatzgebiete erläutert. Dabei handelt es sich um das Starten eines Desktop-Betriebssystems und den Einsatz im Umfeld der eingebetteten Systeme. An dieser Stelle werden deren unterschiedliche Anforderungen und der daraus jeweils resultierende Funktionsumfang erläutert. Wie nahe diese beiden Bereiche jedoch zusammenliegen, wird am Beispiel der Loader GRUB2 und U-Boot erläutert. Beide dienen dazu, ein Desktop-Betriebssystem zu starten. GRUB2 ist dabei ein klassisches Beispiel für einen Loader für den Einsatz auf einem Desktop-Computer. U-Boot hingegen wird mit derselben Aufgabe in eingebetteten Systemen verwendet.

Dies umfasst jedoch Architekturen, welche über die notwendigen Ressourcen für ein Desktop-Betriebssystem verfügen. In der Regel ist ein solches System jedoch für spezielle Aufgaben vorgesehen und benötigt kein Desktop-Betriebssystem oder verfügt nicht über die dafür benötigten Ressourcen. Daher wird im nächsten Schritt aufgezeigt, warum für ein solches System kein universeller Loader verfügbar ist und individuelle Implementierungen zum Einsatz kommen. Dafür wird zunächst auf die aus der verwendeten Hardware resultierenden unterschiedlichen Anforderungen für den Loader eingegangen. Anschließend wird der Funktionsumfang eines solchen Loaders beleuchtet und neben dem Anwendungsstart die Kernaufgabe des Updates dieser Anwendung vorgestellt. Zusätzlich wird auf unterschiedliche Architekturtypen für die Entwicklung eingegangen. Diese befassen sich mit der Möglichkeit eines Backups der Anwendung. Dabei wird zwischen dem Flash-Banking und einem monolithischen Ansatz unterschieden und die jeweiligen Vor- und Nachteile beleuchtet. Abschließend werden diese Konzepte auf die STM32-Architektur in Verbindung mit dem ARM® Cortex-M4®-Mikroprozessor transferiert. Diese wurde im Zuge der Entwicklung verwendet.

9 Zusammenfassung und Ausblick

Die Erläuterung der praktischen Umsetzung eines solchen Bootloaders für eingebettete Systeme beginnt mit der Vorstellung von Aufbau und Architektur des Bootloaders. Diese unterteilt sich in den Bereich des eingebetteten Systems und der optionalen externen Architektur für einen TFTP-Server als zusätzliche Update-Schnittstelle. In diesem Kapitel werden die Hardware-Komponenten des Systems vorgestellt. Dabei wird klargestellt, dass diese Komponenten aufgrund der Projektanforderung verwendet werden und zu welchem Zweck diese für den Bootloader Anwendung finden. Besonders der Mikroprozessor der MCU wird dafür betrachtet. Dieser ist die zentrale Komponente und gilt als notwendige Anforderung für Systeme, auf denen der entwickelte Bootloader zum Einsatz kommen soll. Darüber hinaus werden die Schnittstellen für das Anwendungsupdate vorgestellt und der Auswahlprozess für die Ethernet gebundene TFTP-Schnittstelle neben der integrierten USB-Schnittstelle erläutert. Der Auswahlprozess und Funktionsumfang für das verwendete TFTP-Protokoll wird anschließend in einem gesonderten Kapitel behandelt.

Der Hauptteil der Arbeit umfasst die Implementierung des Bootloaders. Dabei werden die einzelnen Komponenten des Systems gesondert vorgestellt. Zunächst werden die beiden Möglichkeiten für das Speicherlayout dargestellt. Diese unterscheiden sich abhängig von der Konfigurationsoption eines Backup-Mechanismus für den monolithischen Bootloader. Diese Entscheidung wirkt sich auf den generellen Ablauf des Bootprozesses aus. Dieser wird daher im Anschluss erläutert und die Auswirkungen des Backup-Mechanismus auf diesen dargestellt. Als Nächstes wird der Ablauf eines Anwendungsupdates erläutert. Dabei wird sowohl auf das serielle Update über USB als auch die TFTP-Variante über Ethernet betrachtet. Diese werden anschließend mit der Implementierung der Speicherverwaltung in Kontext gesetzt. Zusätzlich wird im Zusammenhang mit der Implementierung die Konfiguration des Bootloaders erläutert und die verschiedenen Möglichkeiten bezüglich des optionalen Backup-Mechanismus, der TFTP-Schnittstelle und eines besseren Nutzerfeedbacks mittels LCD-Anzeige aufgezeigt. Abschließend wird der Sprung in die Hauptanwendung beschrieben und welche Punkte dabei zu beachten sind.

Eine wichtige Anforderung an den Bootloader ist dessen Upgrade-Funktion. Dies ist ein kritischer Prozess, der im Fehlerfall in einem nicht bootbaren Gesamtsystem resultieren kann. Aus diesem Grund werden zwei Möglichkeiten vorgestellt, um die Bootloader-Software zu erneuern. Zum einen ein direktes Update mit entsprechender Software über einen seriell verbundenen Computer und zum anderen ein Upgrade mit Hilfe einer extra dafür entwickelten Firmware, die den Bootloader über eine serielle Schnittstelle erneuert. Die erste Variante wird aufgrund der höheren Stabilität in dieser Arbeit empfohlen. Um die Updatedatei für die Verwendung im System vorzubereiten, wurden passende Bash-Skripte entwickelt, um die Binärdateien mit der Version, Gerätekennung und CRC32-C checksumme anzureichern. Nur in Kombination mit diesen Informationen kann der Bootloader ein Update durchführen. Andererseits führt ein fehlerhafter CRC zu einer Ablehnung der Software.

Im nächsten Schritt werden die Tests beschrieben, welche während und nach der Entwicklung am System vollzogen wurden. Um eine neue Bootloader-Version zu veröffentlichen, wurde zu diesem Zweck ein Release-Protokoll angelegt, welches schrittweise vorschreibt, wie das System getestet werden muss. Nur bei Fehlerfreiheit ist eine Veröffentlichung möglich. Zusätzlich wurde die TFTP-Schnittstelle mit Hilfe eines eigens aufgesetzten Servers getestet und die Kommunikation analysiert. Um die Möglichkeit einer Portierung auf ein anderes System zu testen, wurde ein „Proof of Concept“ für eine eigens zusammengestellte Hardwareplatine durchgeführt. Diese konnte mit einigen Anpassungen an der Hardwareinitialisierung erfolgreich realisiert werden.

Der im Zuge dieser Arbeit entwickelte Bootloader liefert eine Implementierung mit einer umfangreichen Konfiguration und minimaler Hardwareanforderung. So wurde bei der Planung und Implementierung darauf geachtet, möglichst wenige Ressourcen neben dem Funktionsumfang des ARM® Cortex-M4® zu benötigen. Dennoch ist es im Bereich der eingebetteten Systeme üblich, spezielle Anwendungsfälle durch eine eigene Hardwarezusammenstellung zu verwirklichen. Diese weicht womöglich von der für diesen Bootloader verwendeten Hardware ab und benötigt daher etwaige Anpassungen. Darüber hinaus gibt es im Bereich der eingebetteten Systeme kaum Implementierungen für Bootloader ohne feste Hardwarezugehörigkeit. Aus diesen Gründen wird dieses Projekt unter einer Open-Source-Lizenz veröffentlicht, um zukünftigen Entwicklern eine Basis zu bieten. Dies soll zu einer langfristigen Erweiterung der Konfigurationsmöglichkeiten des Bootloaders führen und einen besseren Einstieg für zukünftige Projekte liefern. Der in dieser Arbeit vorgestellte Stand der Software wird darüber hinaus bereits im industriellen Umfeld eingesetzt und soll auch in Zukunft durch Anpassungen und Verbesserungen erweitert werden.

Literaturverzeichnis

- [1] W. Bauer, S. Schlund, D. Marrenbach, and O. Ganschar. Volkswirtschaftliches Potenzial für Deutschland. In *Industrie 4.0*. BITKOM, Frauenhofer-Institut IAO, 2014.
- [2] World Semiconductor Trade Statistics. Prognose zum Umsatz der Halbleiterindustrie weltweit im Bereich Mikroprozessoren und Mikrocontroller von 2006 bis 2022. <https://de.statista.com/statistik/daten/studie/249598/umfrage/umsatz-der-halbleiterindustrie-im-komponentenbereich-mikrochips/>, 2021. letzter Aufruf: 15.06.2021.
- [3] M. Brandt. Robotisierung - So viele Roboter kommen auf 10.000 Beschäftigte. <https://de.statista.com/infografik/13676/roboterdichte-in-der-fertigungsindustrie/>, 2020. letzter Aufruf: 15.06.2021.
- [4] Y. Babar. Learn the Boot Process of Linux, Windows, and Unix. In *Hands-on Booting*. Apress, 2020.
- [5] J. Fei and Z. Lian-yu. Development and Implementation of a Bootloader based on the embedded systems. In *Applied Mechanics and Materials*, volume 48-49, pages 419–422, Tianjin (China), 2011. Trans Tech Publications.
- [6] J. Beningo. Bootloader design for microcontrollers in embedded systems. In *Embedded Software Design Techniques*.
- [7] S. Salzwedel. *Entwurf und Implementierung eines Bootloader-Konzepts zur Programmierung eines Embedded Systems auf Basis der Texas Instruments MSP430 Mikrocontroller Familie*. Diplomarbeit, FH Coburg, 2008.
- [8] R.R. Asche. Grundlagen und praktische Umsetzung fuer industrielle Anwendungen. In *Embedded Controller*, Wiesbaden (Deutschland), 2016. Springer Vieweg.
- [9] Y. Kang, J. Chen, and B. Li. Generic bootloader architecture based on automatic update mechanism. In *2018 IEEE 3rd International Conference on Signal and Image Processing (ICSIP)*, pages 586–590, 2018.
- [10] BCD System Store Settings for UEFI. <https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/bcd-system-store-settings-for-uefi>. letzter Aufruf: 04.05.2021.

Literaturverzeichnis

- [11] GNU GRUB Manual 2.04. <https://www.gnu.org/software/grub/manual/grub/grub.html>. letzter Aufruf: 04.05.2021.
- [12] Installation Guide. https://access.redhat.com/documentation/de-de/red_hat_enterprise_linux/5/html/installation_guide/index. letzter Aufruf: 06.05.2021.
- [13] Installation Guide. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/installation_guide/sect-boot-menu-x86. letzter Aufruf: 06.05.2021.
- [14] A. Vaduva, A. Gonzalez, and C. Simmonds. Linux: Embedded Development. Packt Publishing, 2016.
- [15] A. Doug. Linux for Embedded and Real-time Applications. Boston, Massachusetts (USA), 2017. Newnes.
- [16] Das U-Boot – the Universal Boot Loader. <https://www.denx.de/wiki/U-Boot>. letzter Aufruf: 04.05.2021.
- [17] STMicroelectronics. *RM0090 Reference manual*, 2019.
- [18] H. Hidaka. *Embedded Flash Memory for Embedded Systems: Technology, Design for Sub-systems, and Innovations*. Integrated Circuits and Systems. Springer International Publishing, 2017.
- [19] GENERAL: Intel HEX File Format. <https://www.keil.com/support/docs/1584/>. letzter Aufruf: 28.04.2021.
- [20] TI-TXT Hex Format. <http://downloads.ti.com/docs/esd/SPRUI03/ti-txt-hex-format-ti-txt-option-stdz0795656.html>. letzter Aufruf: 28.04.2021.
- [21] STMicroelectronics. *AN4767 Application note*, 2019.
- [22] S. McConnell. Code Complete, Second Edition. Microsoft Press, 2005.
- [23] STMicroelectronics. *DS12288 STM32G474xB STM32G474xC STM32G474xE*, 2020.
- [24] R. Johnson and S. Christie. IEEE 1149.x and Software Debug. In *JTAG 101*. Intel Corporation, 2009.
- [25] STMicroelectronics. *DocID024030 STM32F427xx STM32F429xx*, 2018.
- [26] ARM. *Arm® Cortex-M4® Datasheet*, 2020.
- [27] Y. Bai. In *Practical Microcontroller Engineering with ARM® Technology*, Charlotte, North Carolina (USA), 2016. John Wiley & Sons, Inc.
- [28] Robert T. Braden. Requirements for Internet Hosts - Application and Support. RFC 1123, October 1989.

- [29] File Transfer Protocol. RFC 959, October 1985.
- [30] TFTP Protocol (revision 2). RFC 783, June 1981.
- [31] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, January 2006.
- [32] IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area network–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Preassociation Discovery. *IEEE Std 802.11aq-2018 (Amendment to IEEE Std 802.11-2016 as amended by IEEE Std 802.11ai-2016, IEEE Std 802.11ah-2016, IEEE Std 802.11aj-2018, and IEEE Std 802.11ak-2018)*, pages 1–69, 2018.
- [33] STMicroelectronics. *DB3871: STM32CubeIDE*, 2021.
- [34] STMicroelectronics. *DB2163: STM32CubeMX*, 2020.
- [35] T. Martin. The Designer’s Guide to the Cortex-M® Processor Family. Newnes, 2016.
- [36] Microsoft Extensible Firmware Initiative FAT32 File System Specification, 2000.
- [37] STMicroelectronics. *UM1721User manual: Developing applications on STM32Cube with FatFs*, 2019.
- [38] lwIP - A Lightweight TCP/IP stack - Summary. <http://savannah.nongnu.org/projects/lwip/>. letzter Aufruf: 19.05.2021.
- [39] lwIP Wiki. https://lwip.fandom.com/wiki/LwIP_Wiki. letzter Aufruf: 19.05.2021.
- [40] Andrew S. Tanenbaum and David Wetherall. *Computer Networks*. Prentice Hall, Boston, 5 edition, 2011.
- [41] STMicroelectronics. *PM0214: Programming manual*, 2020.
- [42] J. Yiu. The Definitive Guide to ARM® Cortex-M3® and Cortex-M4® Processors. Newnes, 2013.
- [43] STMicroelectronics. *DocID029852: STSW-LINK004*, 2016.
- [44] STMicroelectronics. *DB3420: STM32CubeProg*, 2019.
- [45] Licenses & Standards. <https://opensource.org/licenses>. letzter Aufruf: 02.06.2021.
- [46] GNU General Public License. <https://www.gnu.org/licenses/gpl-3.0.en.html>. letzter Aufruf: 02.06.2021.

Literaturverzeichnis

- [47] Freie Software verkaufen. <https://www.gnu.org/philosophy/selling.html>. letzter Aufruf: 02.06.2021.
- [48] The License Review Process. <https://opensource.org/approval>. letzter Aufruf: 02.06.2021.
- [49] The Open Source Definition (Annotated). <https://opensource.org/osd-annotated>. letzter Aufruf: 02.06.2021.
- [50] Freie Software. Was ist das? <https://www.gnu.org/philosophy/free-sw.de.html>. letzter Aufruf: 02.06.2021.
- [51] R. Stallman. Warum „Open Source“ das Ziel Freie Software verfehlt. <https://www.gnu.org/philosophy/open-source-misses-the-point.html>. letzter Aufruf: 02.06.2021.
- [52] What is "copyleft"? Is it the same as open source? <https://opensource.org/faq#copyleft>. letzter Aufruf: 02.06.2021.
- [53] Copyleft. Was ist das? <https://www.gnu.org/copyleft/>. letzter Aufruf: 02.06.2021.
- [54] Various Licenses and Comments about Them. <https://www.gnu.org/licenses/license-list.en.html>. letzter Aufruf: 02.06.2021.
- [55] The 3-Clause BSD License. <https://opensource.org/licenses/BSD-3-Clause>. letzter Aufruf: 02.06.2021.
- [56] STMicroelectronics. *AN4989 Application note*, 2021.
- [57] STMicroelectronics. *DB3692 STLINK-V3SET*, 2019.
- [58] Valgrind. <https://valgrind.org/>. letzter Aufruf: 08.06.2021.
- [59] FreeBSD/ARM® Project. <https://www.freebsd.org/platforms/arm/>. letzter Aufruf: 08.06.2021.
- [60] Operating system images. <https://www.raspberrypi.org/software/operating-systems/>. letzter Aufruf: 09.06.2021.
- [61] J. Plötner and S. Wendzel. In *Linux Das umfassende Handbuch*, volume 5. Galileo Computing, 2012.
- [62] Dämon tftpd. <https://www.ibm.com/docs/de/aix/7.2?topic=t-tftpd-daemon>. letzter Aufruf: 09.06.2021.
- [63] About Wireshark. <https://www.wireshark.org/>. letzter Aufruf: 09.06.2021.
- [64] A. Harnau. Schwingquarz oder Quarzoszillator – wie einsetzen? In *Elektronik Industrie*, volume 3. GEYER ELECTRONIC, 2006.