FAKULTÄT FÜR !NFORMATIK

Modular Security Enhanced Bootloader

BAKKALAUREATSARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Bernhard Mathias
Matrikelnummer 0627917

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer/Betreuerin: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Christian Walter MSc.

Wien, 15.04.2010

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Bernhard Mathias, 2124 Oberkreuzstetten

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Wien, 15.04.2010 _____

(Unterschrift)

Kurzfassung

Diese Bakkalaureatsarbeit befasst sich mit dem Design und mit der Entwicklung eines modularen Bootloaders mit kryptographischen Funktionen.

Die modulare Architektur bietet den wesentlichen Vorteil, dass der Bootloader leicht an verschiedene Plattformen angepasst werden kann.

Kryptographische Funktionen sollen verhindern, dass Produkte, welche den Bootloader verwenden, nachgebaut werden können.

Die Bakkalaureatsarbeit beschreibt zunächst den modularen Aufbau des Bootloaders. Es folgt eine Analyse möglicher Verschlüsselungs- bzw. Hashfunktionen für den Bootloader. Des Weiteren wird ein eigenes Kommunikationsprotokoll sowie der Aufbau der durch den Bootloader unterstützten Firmwareimages definiert.
Die Verwendung des Bootloaders auf unterschiedlichen Plattformen wird anhand von Beispielen näher erläutert.

Die Implementierung des Bootloaders erfolgt auf einem AVR32 UC3A1512 Prozessor und wird in den letzten Kapiteln dieser Arbeit näher beschrieben.
Zudem sind ein Verschlüsselungsprogramm (BIN2SEC), womit verschlüsselte Firmwareimages erstellt werden können, sowie eine PC-Software, welche für die Kommunikation mit den Bootloader zuständig ist, notwendig.

Abstract

This bachelor thesis deals with the design and with the implementation of a modular bootloader with cryptographic functions.

The modular design offers the great advantage, that the bootloader can be adapted to different platforms very easily.

Cryptographic functions ensure that products using this bootloader can't be copied.

At the beginning the thesis deals with the modular architecture of the bootloader. After this, an analysis on different encryption- and hashfunctions will be carried out. An own communication protocol and the format of the supported firmware images will be defined in the following chapters.
The usage on different platforms will be shown by means of examples.

The last chapters deal with the implementation of the bootloader on an AVR32 UC3A1512 processor. The full implementation includes the bootloader, a software for generating encrypted firmware images (BIN2SEC) and an additional PC software, which is necessary to communicate with the bootloader.

Inhaltsverzeichnis

1 EINLEITUNG 6

2 ÄHNLICHE PROJEKTE 7

3 ARCHITEKTUR 8

3.1 ANFORDERUNGEN 8
3.1.1 Anforderungen Bootloader (Embedded Target) 8
3.1.2 Anforderungen PC-Software 9
3.1.3 Anforderung Sicherheit 9
3.2 BLOCKDIAGRAMM 10
3.3 ENTWURFSRICHTLINIEN 10
3.3.1 Handhabung unterschiedlicher Blockgrößen 10
3.4 BOOT MODUL 11
3.5 CONTROL MODUL 11
3.6 FLASH MODUL 12
3.7 COMMUNICATION MODUL 15
3.7.1 Blockgrößen im Communication Modul 15
3.7.2 Application Protocol Layer (= APL) 18
3.7.3 Data Link Layer 19
3.7.4 Physical Layer 19
3.8 CRYPTO MODUL 20
3.9 APPLICATION 21

4 FUNKTION DES BOOTLOADERS 22

5 KRYPTOGRAPHIE 23

5.1 GRUNDLAGEN 23
5.1.1 Algorithmen und Schlüssel 23
5.1.2 Symmetrische Algorithmen 24
5.1.3 Public-Key (=asymmetrische) Algorithmen 24
5.1.4 Sicherheit der Algorithmen 24
5.2 ÜBERLEGUNGEN ZUM PROJEKT 25
5.2.1 Mögliche Verschlüsselungen im Überblick 26
5.2.1.1 RSA 26
5.2.1.2 DES, 3DES 27
5.2.1.3 AES 28
5.2.1.4 Blowfish 29
5.2.1.5 Twofish 30
5.2.1.6 XXTEA 31
5.2.1.7 Auswahl des Algorithmus 32
5.3 KRYPTOGRAPHISCHE HASHFUNKTIONEN 33
5.3.1 SHA 34
5.3.2 MD5 35
5.3.3 Fazit 36

6 KOMMUNIKATIONSPROTOKOLL 37

7 FIRMWARE IMAGE 39

7.1 BEARBEITUNG DES FIRMWAREIMAGES IM BOOTLOADER 40

8 BEISPIELE 41

8.1 BEISPIEL FÜR VERWENDUNG AUF 8-BIT AVR MIT RS485 41
8.1.1 Communication Modul 41
8.2 BEISPIEL FÜR VERWENDUNG AUF CORTEX M3 MIT ETHERNET 42

8.2.1 Communication Modul *42*

9 PC-SOFTWARE (UPDATESW) **44**

9.1 FUNKTIONEN 44
9.1.1 Kommunikation mit Sensor bei Firmwareupdate *45*
9.1.2 Verify *46*
9.2 KOMMUNIKATIONSPROTOKOLL 46
9.3 USERINTERFACE 46

10 BIN2SEC **46**

11 IMPLEMENTIERUNG DES BOOTLOADERS **47**

11.1 BOOTLOADER 47
11.1.1 Allgemeines *47*
11.1.2 Communication Modul *47*
11.1.2.1 APL 47
11.1.2.2 Data Link Layer 47
11.1.2.3 Physical Layer 47
11.1.2.4 Frameformat 47
11.1.3 Crypto Modul *47*
11.2 PC-SOFTWARE (UPDATESW) 48
11.3 BIN2SEC 48

12 BENUTZERHANDBUCH **48**

12.1 PC-SOFTWARE (UPDATESW) 48
12.2 BIN2SEC 50

13 ZUSAMMENFASSUNG **51**

14 QUELLENVERZEICHNIS **52**

15 ABBILDUNGSVERZEICHNIS **53**

16 TABELLENVERZEICHNIS **53**

1 Einleitung

Ein Bootloader ist ein von der Hauptapplikation unabhängiges Programm, das dazu verwendet wird, um die Applikation in einen internen oder auch in einen externen Speicher eines Mikroprozessors zu laden oder eine bereits programmierte Applikation durch eine neuere Version zu ersetzen.

Bei einem Reset des Mikrocontrollers wird der Bootloader gestartet, welcher daraufhin, je nach Implementierung, z.B. eine vorgegebene Zeit auf einen externen Befehl wartet, um sich mit einem Wartungsgerät zu verbinden. Dieses kann dann weitere Befehle an den Bootloader schicken. Hierbei kann es sich beispielsweise um ein einfaches Verify, also einen Befehl für die Überprüfung der programmierten Software, handeln. Die Hauptaufgabe wird allerdings darin bestehen, ein Update der Firmware durchzuführen.

Wird ein Update-Kommando empfangen, so wird die bestehende Applikation durch die neue Firmware ersetzt. Wird kein Kommando empfangen, so wird die bereits programmierte Applikation gestartet. Sollte keine Applikation gespeichert sein, so wird weiterhin auf ein Kommando gewartet.

Während bei einer herkömmlichen Programmierung ohne Bootloader der Flash-Speicher über spezielle Programmer (z.B. JTAG) gelöscht und neu programmiert werden muss, können mithilfe des Bootloaders Standard-Kommunikationsschnittstellen (z.B. RS 485) initialisiert und für die Programmierung verwendet werden.

Ein Bootloader bietet also den wesentlichen Vorteil, dass ein Update im Feld, d.h., bei dem Kunden selber durchgeführt werden kann. Die Programmierung ist zudem mit einem geringeren Hardwareaufwand möglich, da keine speziellen Programmer notwendig sind. Auch eine schnellere Übertragungsrate bei der Programmierung des Speichers wird mithilfe des Bootloaders erreicht, da Schnittstellen mit höheren Übertragungsraten verwendet werden können.

Zurzeit werden allerdings für unterschiedliche Architekturen meist unterschiedliche Bootloader verwendet. Dies ist auch ein Nachteil für die Firma S::can Messtechnik, für Sensoren und Sonden werden größtenteils nachfolgende, unterschiedliche Plattformen Verwendung finden:

- AVR
- AVR32
- ARM Cortex M3
- Motorola CPU32
- Texas Instruments MSP430

Jede Plattform hat ihren eigenen Bootloader mit einer eigenen zugehörigen PC-Software. Dadurch entsteht ein hoher Wartungsaufwand für die Software und damit verbunden auch für die zugehörigen Handbücher.

Ziel dieser Bakkalaureatsarbeit ist das Design und die Implementierung eines modulareren Bootloaders, welcher leicht an die verschiedenen Plattformen angepasst werden kann.

Ein weiteres Problem entsteht für die Firma S::can beim Export in Länder wie z.B. China wo Produktpiraterie ein großes Problem ist. Daher soll der Bootloader ein Crypto Modul besitzen, welches es erlaubt, verschlüsselte Firmwareimages auf den Prozessor zu programmieren.

Ein weiteres Kriterium für die Entwicklung des modularen Bootloaders ist, dass der Bootloader selbst geschützt sein muss. Einerseits darf der Bootloaderbereich nicht überschrieben werden können - der Speicherbereich muss also durch die Software entsprechend geschützt werden. Des Weiteren darf der Speicherbereich nicht gelesen werden können.

Es muss auch darauf geachtet werden, dass keine unzulässigen Applikationen auf dem Mikroprozessor gestartet werden können.

Auch auf der Hardwareseite muss ein Schutz vor fremden Zugriff gewährleistet sein (Tamper Protection: Tamper Resistance, Tamper Detection, Tamper Response). Möglich wäre z.B. ein Löschen der Applikation als Tamper Response.

2 Ähnliche Projekte

Es gibt bereits ähnliche Projekte, welche ebenfalls einen modularen Bootloader zur Verfügung stellen, wie z.B. das Open Source Projekt U-boot (universal bootloader) [1]. Dieser Bootloader unterstützt unter anderem folgende Architekturen:

- PPC
- ARM
- MIPS
- x86
- m68k
- NIOS
- Microblaze

Jedoch hat U-boot einen relativ hohen Ressourcenverbrauch. Mit einer minimalen aber dennoch nutzbaren Konfiguration benötigt U-boot immer noch 128 kByte im Flash-Speicher, was kleinere Prozessoren schon ziemlich auslastet. Des Weiteren werden von S::can verwendete Architekturen wie z.B. AVR-Prozessoren nicht durch diesen Bootloader unterstützt.

Ein weiteres Projekt ist der Flash-Bootloader von der Firma Vector [2], welcher kommerziell vertrieben wird. Dieser ist allerdings für den Automobil-Bereich für das Programmieren von Motorsteuergeräten entwickelt worden und unterstützt Architekturen wie zum Beispiel:

- Fujitsu 16FX
- Infineon C16x
- Microchip PIC18
- Texas Instruments TMS320
- Toshiba TLCS900

Dieser Bootloader würde sogar ein Security Modul für eine RSA und eine H-MAC Verschlüsselung anbieten. Durch die Spezialisierung auf den Automobilbereich werden aber auch hier sehr viele von S::can verwendete Architekturen nicht unterstützt. Zudem werden nur die in der Automobilindustrie üblichen Kommunikationsprotokolle wie CAN, LIN und FlexRay unterstützt, welche bei S::can nur eine untergeordnete Rolle spielen.

Auch für AVR-Prozessoren gibt es einen Open Source Bootloader mit der Bezeichnung „AVR Universal Bootloader“, entwickelt von Shaoziyang [3]. Dieser lässt sich durch Makrodefinitionen an die verschiedenen AVR-Prozessoren anpassen. Es wird aber keine Verschlüsselung unterstützt und der Bootloader ist nur auf AVR-Plattformen verwendbar.

3 Architektur

3.1 Anforderungen

3.1.1 Anforderungen Bootloader (Embedded Target)

- Die Vervielfachung der Software ist nicht das Problem von S::can (alle Kunden haben einen kostenlosen Anspruch auf die aktuelle Softwareversion). Es soll allerdings der Nachbau der Sonden und Sensoren mithilfe des Bootloaders verhindert werden.
- Es ist auf minimalen Ressourcen-Verbrauch zu achten. Hierbei sind folgende Abstriche zulässig:
  - Als physikalische Transportschicht nur RS485/RS232 (vorzugsweise auch Ethernet)
  - Nur einfache Verschlüsselung
- Da jeder Mikrocontroller unterschiedliche Flash-Typen hat, muss ein Flash-Zugriffslayer definiert werden. Dieser muss mindestens folgende Funktionen unterstützen:
  - Initialisierung
  - Ermitteln der Page-Größe für Lese- und Schreibzugriff
  - Löschen

  - Schreiben
  - Lesen
  - Freigabe der verwendeten Ressourcen
- Es muss ein Interface für ein Crypto Modul vorhanden sein. Das Crypto Modul dient zur Verschlüsselung der Software und wird für alle sicherheitsrelevanten Daten verwendet. Als sicherheitsrelevant gelten die Firmware sowie sämtliche Applikationskonfigurationen (= Dateien, welche nur auf einen Sensor kopiert werden und später von der Applikation gebraucht werden). Das Crypto Modul unterstützt entweder ein sehr effizientes Verfahren, welches auf kleinen 8-Bit Mikrocontrollern funktioniert oder muss die Möglichkeit bieten, in verschiedenen Ausführungen vorhanden zu sein.
- Ein Interface für das Booten in der Applikation muss vorhanden sein, da das Booten prozessorspezifisch ist.
- Geschriebene Bootloader-Bereiche sollen durch eine Checksumme geschützt werden, wobei hier überlegt werden muss, wo diese gespeichert wird. Eventuell auch nur optional.
- Es muss ein Protokoll für die Kommunikation zwischen Bootloader- und PC-Software definiert werden. Dieses ist von der physikalischen Schnittstelle unabhängig zu realisieren.
- Es muss eine physikalische Transportschicht als Modul eingeführt werden. Diese empfängt und sendet Datenframes, wie bereits definiert. Zusätzlich werden noch Header und Trailer angefügt.
- Wünschenswert sind eine hohe Updategeschwindigkeit und eine hohe Ausfallssicherheit beim Updatevorgang.

3.1.2 Anforderungen PC-Software

- Erlaubt Bedienung von der Kommandozeile.
- Hat eine integrierte Hilfe in der Kommandozeile (-h).
- Liefert Feedback über den aktuell laufenden Vorgang.
- Läuft unter Windows XP, Windows Vista und Windows 7. Vorzugsweise auch unter Windows NT und Linux.
- Erlaubt das Schreiben und Verifizieren der jeweiligen Flash-Speicherblöcke.
- Soll aus Gründen der Wartbarkeit das gleiche Kommunikationsprotokoll (=gleicher Code) wie der Mikrocontroller verwenden.
- Wünschenswert ist ein komfortables User Interface.

3.1.3 Anforderung Sicherheit

- Software darf nicht disassembliert werden können.
- Software darf durch PC-Virenscanner nicht als Virus erkannt werden.
- Applikationsdateien dürfen nicht dekodiert werden können (z.B. ein Textfile, welches über den Bootloader übertragen wird).

3.2 Blockdiagramm

Die einzelnen Module des Bootloaders sind in Abbildung 1 ersichtlich.

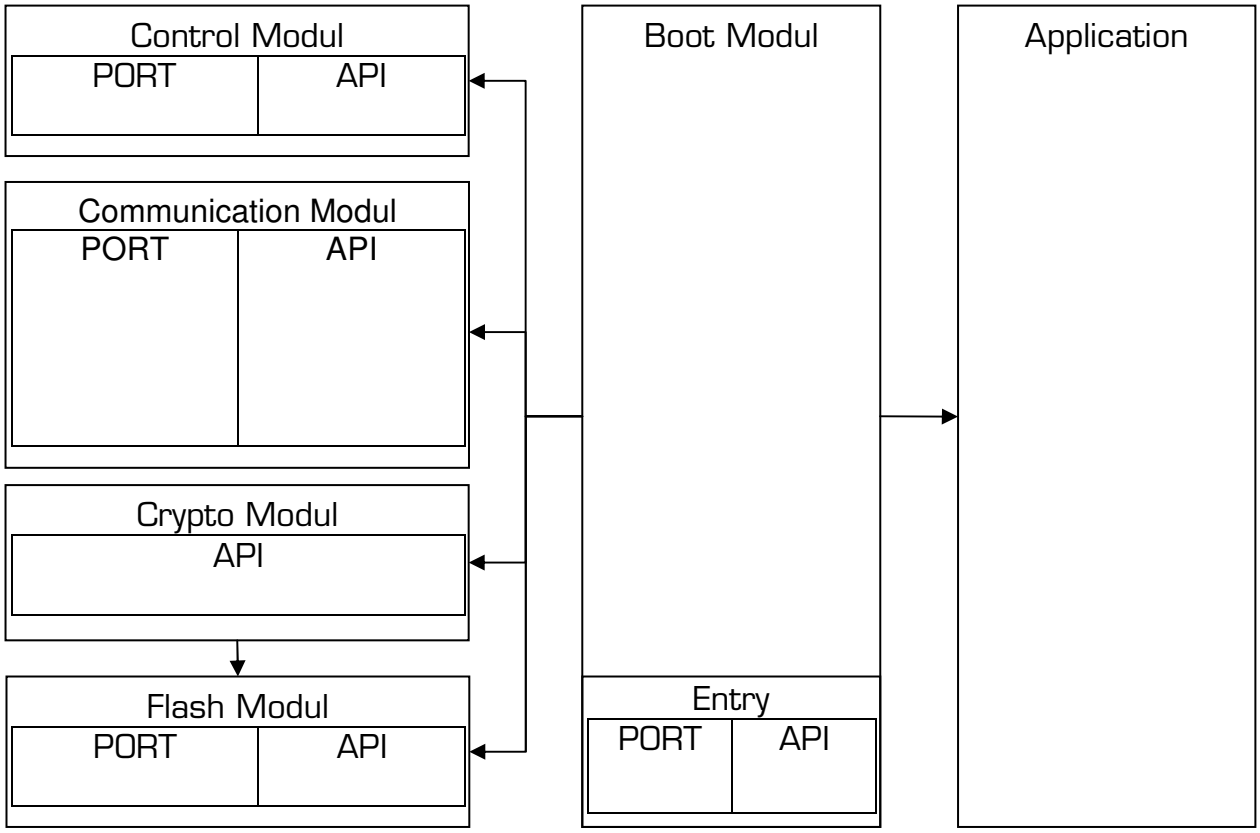

Abbildung 1: Modularer Bootloader

Alle Module bestehen aus einer oder mehreren generischen API's, welche unabhängig vom verwendeten Prozessortyp sind.
Wie aus der Grafik ersichtlich, bestehen einige Module aus einer API, andere Module bestehen aus 2 Teilen: Port und API. API bezeichnet die bereits erwähnten generischen API's, PORT bezeichnet eine prozessorspezifische Implementierung.

3.3 Entwurfsrichtlinien

3.3.1 Handhabung unterschiedlicher Blockgrößen

In unterschiedlichen Modulen werden jeweils unterschiedliche Blockgrößen verarbeitet:

- Im Flash Modul wird mit bestimmten Blockgrößen auf den Flash-Speicher zugegriffen.
- Das Communication Modul verwendet verschiedene Blockgrößen
  - im Application Protocol Layer
  - im Data Link Layer

  - im Physical Layer
- Das Crypto Modul arbeitet mit einzelnen Datenblöcken bzw. mit einzelnen Bytes, falls ein Stream Cipher implementiert werden sollte.

Aus diesem Grund müssen die Datenblöcke jeweils zwischengepuffert bzw. in einzelne Blöcke aufgeteilt werden. Hierfür beinhaltet jedes Modul, welches mit Datenblöcken arbeitet, eine Funktion, welche die benötigte Blockgröße retourniert. Die Blockbearbeitung ist jeweils die Aufgabe des in der Hierarchie weiter oben liegenden Layers. Will z.B. das Crypto Modul bei einem Verify über das Flash Modul auf den Flash-Speicher zugreifen, so ist das Crypto Modul dafür verantwortlich, dass das Flash Modul die Daten in der korrekten Blockgröße bekommt.

Bsp.: Unterschiedliche Blockgrößen bei einem Firmwareupdate

Crypto Modul: Blöcke zu 256 Bytes
Communication Modul: Blöcke zu 128 Bytes
Flash Modul: 512 Bytes

- Die PC-Software schickt das Firmwareimage in Blöcken zu 128 Bytes über das Communication Modul an das Boot Modul.
- Sobald 2 Blöcke empfangen wurden, können diese zu einem Datenblock zusammengefasst und als 256 Byte Block an das Crypto Modul übergeben werden.
- Das Crypto Modul entschlüsselt den Datenblock und gibt diesen wieder an das Boot Modul zurück.
- Dieser Datenblock muss in den Flash-Speicher geschrieben werden. Da das Flash Modul jedoch nur Blöcke von 512 Bytes akzeptiert, wird der Datenblock vorerst zwischengepuffert.
- Sobald der nächste 256 Byte Block vom Crypto Modul erhalten wird, können beide Datenblöcke wieder zusammengefasst und in den Flash-Speicher geschrieben werden.

Bei diesem Beispiel wurden aus Übersichtlichkeitsgründen Blockgrößen zu 2-er Potenzen gewählt. Jedoch soll der Bootloader jede beliebige Blockgröße wie z.B. 51 Bytes unterstützen.

3.4 Boot Modul

Dieses Modul beinhaltet den eigentlichen Bootloader, welcher nach dem Reset des Prozessors gestartet wird.
Das Boot Modul koordiniert sämtliche Vorgänge nach dem Reset und ist für die korrekte Ansteuerung der restlichen Module verantwortlich.

3.5 Control Modul

Dieses Modul stellt grundlegende Funktionen für die Verwendung des Bootloaders zur Verfügung, welche abhängig vom Prozessortyp sind.

Die API stellt folgende Funktionen zur Verfügung:

- void vBLInitPlatform (void)
  Hier wird die für den Bootloader nötige Hardware initialisiert.
- void vBLClosePlatform (void)
  Diese Funktion macht sämtliche durch vBLInitPlatform vorgenommenen Einstellungen rückgängig.
- void vBLClearWatchdog (void)
  Setzt den Watchdog-Timer zurück.
- void vBLBoot (void)
  Startet den Controller neu.
- void vBLStartApp (uint64_t uiAddress)
  Startet die an der durch uiAddress definierten Position im Flash-Speicher gespeicherte Applikation.

3.6 Flash Modul

Dieses Modul ist für den Flashzugriff verantwortlich. Es sollen über die API Datenblöcke gelesen, geschrieben und gelöscht werden können. Hierbei ist allerdings zu beachten, dass die meisten Prozessoren immer nur das Lesen bzw. Schreiben einzelner Pages unterstützen. Die Größe der Pages kann jedoch bei den einzelnen Prozessoren variieren. Daher muss es auch eine Funktion geben, welche die Pagegröße des Flash-Speichers für den jeweiligen Prozessor ermittelt.

Die von S::can verwendeten und für dieses Projekt relevanten Controller verwenden folgende Pagegrößen:

- AVR (8-Bit)
  - ATMEGA324P: 256 Pages zu je 64 Words (=128 Bytes)
- AVR32
  - UCR3A1512: 1024 Pages zu je 128 Words (=512 Bytes)
- ARM Cortex M3:
  - ST Microelectronics STM32F10xxx: Je nach Modell Pagegrößen zu 1 bzw. 2 kBytes
- Texas Instruments MSP430
  - MSP430F2619: Single Byte oder Single Wordzugriff

Wie für die angeführten Prozessoren kann auch für jeden weiteren beliebigen Prozessor eine Pagegröße ermittelt werden. Es gibt die folgenden theoretischen Extrembeispiele:

- Werden immer nur einzelne Bytes geschrieben, so ist die Pagegröße gleich 1.
- Wird der gesamte Flash-Speicher auf einmal geschrieben, so ist die Pagegröße gleich der Flash-Größe.

Des Weiteren wurde überprüft, ob die verwendeten Controller die Möglichkeit bieten, den Flash-Speicher vor einem Lesezugriff zu schützen. Dies ist notwendig, da ansonsten jegliche Verschlüsselung sinnlos wäre. Der Flash-Speicher könnte einfach ausgelesen werden und man hätte den Hex-Code zur weiteren Bearbeitung zur Verfügung:

- ATMEGA324P (8-Bit): Es gibt entsprechende Fuse-Bits.
- UCR3A1512: Es gibt entsprechende Fuse-Bits.
- ST Microelectronics STM32F10xxx: Der Speicher kann durch ein Option Byte geschützt werden.
- MSP430F2619: Es gibt eine Protection fuse, wobei aber beachtet werden muss, dass der Prozessor nach dem Durchbrennen der Fuse nicht mehr programmiert werden kann (auch nicht über JTAG).

Weitere Probleme beim Flashzugriff:

- **Größe des Firmwareimages ist kein Vielfaches der Pagegröße**
  In diesem Fall muss die zuletzt geschriebene Page mit einzelnen Bits mit logisch 1 aufgefüllt werden.
- **Aufteilung des Flash-Speichers**
  Es gibt 3 relevante Bereiche im Flash-Speicher:
  - **Bootloader-Section:** Hier wird der Bootloader gespeichert. Dieser Bereich wird durch Fuse-Bits vor Schreib- und Lesezugriffen geschützt.
  - **Firmwareinformationsbereich:** Dies ist ein weiterer Bereich außerhalb der Bootloader-Section, wo der Bootloader Daten ablegen kann (dies ist aufgrund der kryptographischen Anforderungen des Bootloaders notwendig, siehe Kapitel 7.1). Hier muss beachtet werden, dass dieser Bereich für den Bootloader schreibbar bleibt, es dürfen also keine Fuse-Bits gesetzt werden, welche dies verhindern würden.
  - **Applikationsbereich:** Dies ist der Speicherbereich für die zu programmierende Applikation. Auch dieser Bereich muss für den Bootloader schreibbar bleiben. Die Firmware wird also nicht direkt an den Anfang des Flash-Speichers geschrieben, sondern muss entsprechend gelinkt werden.

Die API berücksichtigt auch, dass ein Prozessor Zugriff auf mehrere Flash-Speicher (intern+extern) haben kann. Es kann also spezifiziert werden, auf welchen Speicher zugegriffen wird. Es kann allerdings nur eine Applikation gebootet werden, welche sich im selben Speicher wie der Bootloader befindet.

Die API stellt die folgenden Funktionen für den Flash-Zugriff zur Verfügung.

- err_t eBLFlashInit (void)
  Initialisiert das Flash Modul soweit, dass alle weiteren Funktionen innerhalb dieses Moduls verwendet werden können.
- err_t eBLFlashGetPageSize (uint8_t uiChipID, uint16_t * puiPageSize)

Gibt über puiPageSize die Größe der einzelnen Pages zurück, auf welche in einem Lese- bzw. Schreibbefehl zugegriffen werden kann. uiChipID spezifiziert hierbei den zu verwendeten Flash-Speicher.

- err_t eBLFlashWriteFirmware (uint8_t uiChipID, uint32_t uiPageNr, uint64_t uiFlashOffset, uint8_t * puiData)
  Schreibt die Daten puiData der Größe einer Page (entspricht dem Rückgabewert von uiBLFlashGetPageSize) an die durch uiPageNr angegebene Position in dem durch uiChipID spezifizierten Flash-Speicher. Die erste Page des Firmwareimages wird mit dem Index uiPageNr = 0 an die durch uiFlashOffset angegebene Position geschrieben.
- err_t eBLFlashCompareFirmware (uint8_t uiChipID, uint32_t uiPageNr, uint64_t uiFlashOffset, uint8_t * puiData)
  Liest die Daten der Größe einer Page (entspricht dem Rückgabewert von uiBLFlashGetPageSize) an der durch uiPageNr und uiFlashOffset angegebenen Position in dem durch uiChipID spezifizierten Flash-Speicher und vergleicht sie mit dem Datenpuffer puiData.
- err_t eBLFlashRead (uint32_t uiPageNr, uint64_t uiFlashOffset, uint8_t * puiData)
  Liest die Daten der Größe einer Page (entspricht dem Rückgabewert von uiBLFlashGetPageSize) an der durch uiPageNr und uiFlashOffset angegebenen Position im Flash-Speicher und schreibt sie in den Datenpuffer puiData. Eine Flash-ID ist hier nicht notwendig, da diese Funktion nur zur Berechnung der Hash-Signatur verwendet wird (diese wird nur für eine bootbare Applikation kontrolliert, welche sich immer im selben Flash-Speicher wie auch der Bootloader befindet).
- err_t eBLWriteFirmwareInfo (uint8_t uiFirmwareVersion, uint8_t uiBootloaderVersion, uint8_t uiCpuType, uint64_t uiLength, uint64_t uiAddressOffset, uint8_t * puiHash)
  Diese Funktion wird nach einem Update-Vorgang verwendet, um alle notwendigen Informationen des Firmware-Images an einer geeigneten Stelle im Flash-Speicher abzulegen. Dieser Block muss auch durch einen CRC geschützt werden.
- err_t eBLLoadFirmwareInfo (uint8_t * puiFirmwareVersion, uint8_t * puiBootloaderVersion, uint8_t * puiCpuType, uint64_t * puiLength, uint64_t * puiAddressOffset, uint8_t * puiHash)
  Diese Funktion wird verwendet, um die gespeicherten Informationen eines Firmwareimages wieder aus dem Flash-Speicher zu laden. Ist der CRC nicht korrekt, so muss ein Fehler gemeldet werden.
- err_t eBLCheckCRC (void)
  Überprüft, ob der Firmwareinformationsbereich gültig ist.
- void vBLGetFlashStartAddress (uint64_t ** puiStartAddress)
  Gibt über puiStartAddress einen Pointer auf die Start-Adresse im Flash-Speicher zurück.
- err_t eBLFlashErase (uint8_t uiChipID)

Löscht den durch uiChipID spezifizierten Flash-Speicher. Hier muss allerdings darauf geachtet werden, dass der Bootloader-Bereich nicht gelöscht werden kann.

- err_t eBLFlashClose (void)
  Hier werden alle verwendeten Ressourcen freigegeben.

Die Implementierung der einzelnen Funktionen ist in diesem Modul prozessorspezifisch.

3.7 Communication Modul

Dieses Modul ist für die Kommunikation mit der PC-Software zuständig. Tabelle 1 zeigt den Aufbau und die einzelnen Layer des Communication Moduls. RS485 und Ethernet sind nur Beispiele, es können auch andere Schnittstellen verwendet werden.

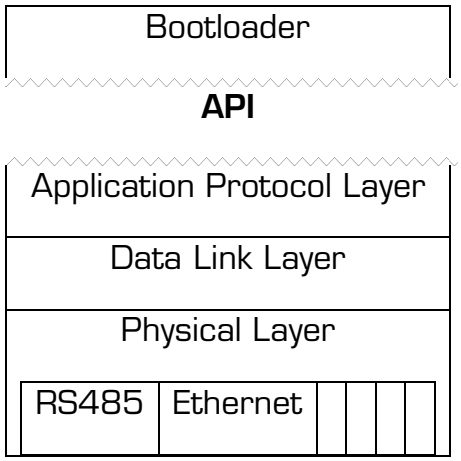

Tabelle 1: Aufbau Communication Modul

Es werden 3 Layer (Application Protocol Layer, Data Link Layer, Physical Layer) in einzelnen Teilmodulen implementiert. Über die API kann das Boot Modul auf den obersten Layer, also den Application Protocol Layer zugreifen. Die Implementierung des Data Link Layers und des Physical Layers ist prozessorspezifisch.

3.7.1 Blockgrößen im Communication Modul

Der Application Protocol Layer, der Data Link Layer und der Physical Layer spezifizieren jeweils unterschiedliche Protokolle, welche auch Datenübertragungen mit unterschiedlichen Blockgrößen unterstützen können.

Das Protokoll im APL, welches in Kapitel 6 näher beschrieben wird, versendet Datenblöcke mit bis zu 258 Bytes (256 Bytes Payload + 2 Bytes Protokolldaten). Es ist nicht vorgegeben, welches Protokoll im Data Link Layer bzw. im Physical Layer verwendet wird.

Prinzipiell soll eine Segmentierung der Datenframes während der Kommunikation vermieden werden. Dies hat den Vorteil einer einfacheren Implementierung und vermeidet eine potentielle Fehlerquelle im Communication Modul. Des Weiteren soll nur ein Datenpuffer für alle Layer verwendet werden, somit ist es nicht nötig, Daten zwischen den Layern zu kopieren, was Ressourcen einspart.

Es wird folgendes Prinzip implementiert:

- Der Physical Layer bestimmt die Größe eines Datenframes.
- Die Payload Größe im Physical Layer muss groß genug sein, um Header und Trailer der übergeordneten Layer einbinden zu können.
- Alle anderen Layer müssen Datenframes mit der durch den Physical Layer definierten Größe unterstützen. Ansonsten muss der Physical Layer angepasst werden.
- Jeder Layer im Kommunikationsstack muss wissen, wie viele Bytes die unteren Layer für Header und Trailer benötigen. Des Weiteren muss jeder Layer wissen, wie viele Bytes für den Payload durch den unteren Layer zur Verfügung gestellt werden. In diesen Payload schreibt dann der entsprechende Layer seinen Header, seinen Trailer und seinen Payload.
- Das Boot Modul weiß, wie viele Bytes für Header und Trailer in allen Schichten zusammen benötigt werden. Ebenso weiß das Boot Modul, welche Payloadgröße durch den APL zur Verfügung gestellt wird. Der Datenpuffer wird im Boot Modul angelegt.
- Jeder Layer bekommt den gesamten Datenpuffer übergeben, schreibt seine Daten aber nur an die für ihn vorgesehene Stelle.

Bevor Daten über den APL versendet werden können, müssen über die Funktion vBLAplGetBlockSize die Größen des Headers und des Trailers der unteren Layer sowie die max. mögliche Größe des Payloads ermittelt werden.

Die Funktion vBLAplGetBlockSize selbst ruft die Funktion vBLDatalinkGetBlockSize auf, welche selbige Daten vom Data Link Layer abfragt. Dieser wiederum ruft vBLPhysicalGetBlockSize auf.

Sind die Daten vom Physical Layer bekannt, kann der Data Link Layer berechnen, wie viele Bytes dieser als Payload zur Verfügung stellen kann, wobei noch Header und Trailer hinzugefügt werden müssen.

Bsp.:

- Physical Layer: 5 Bytes Header, 5 Bytes Trailer, 256 Bytes Payload
- Data Link Layer: 5 Bytes Header, 5 Bytes Trailer
- Application Protocol Layer: 2 Bytes Header, 0 Bytes Trailer

Der Datenpuffer sieht wie folgt aus:

5 Bytes	5 Bytes	2 Bytes	0-244 Bytes	0 Bytes	5 Bytes	5 Bytes
Physical Header	Data Link Header	APL Header	Payload	APL Trailer	Data Link Trailer	Physical Trailer

Tabelle 2: Datenpuffer im Communication Modul

Dem Boot Modul stehen also $266 - 4 * 5 - 2 = 244 Bytes$ für den Payload zur Verfügung. Werden beim Update bzw. beim Verify der Firmware Nutzdaten gesendet, so wird der Frame immer komplett gefüllt (es wäre mithilfe dieser Implementierung auch möglich, nur teilweise gefüllte Frames zu versenden). Werden keine Nutzdaten, sondern nur ein Funktionscode gesendet, so bleibt der Payload leer.

Das Boot Modul weiß also nach dem Aufruf von eBLAplGetBlockSize, dass alle Layer zusammen 12 Bytes für den Header und 10 Bytes für den Trailer benötigen. Das Boot Modul weiß auch, dass 244 Bytes für den Payload verwendet werden können, legt also einen Datenpuffer mit 266 Bytes an.

Senden von Daten:

- **Boot Modul**
  Werden Daten gesendet, wird nur das Kommando, welches gesendet wird, an den APL übergeben. Eventuelle Nutzdaten werden ab dem 13. Byte in den Datenpuffer geschrieben. Des Weiteren wird die Größe der geschriebenen Nutzdaten an den APL übergeben (entweder 0 oder 244). Zurzeit ist es jedoch nicht nötig, dass das Boot Modul Nutzdaten sendet.
- **Application Protocol Layer**
  Der APL weiß, dass der Data Link Layer und der Physical Layer gemeinsam 10 Bytes für den Header und 10 Bytes für den Trailer benötigen. Der APL schreibt also seinen Header ab dem 11. Byte des Datenpuffers, fügt den Payload aus dem Boot Modul und seinen Trailer hinzu und gibt die Größe des bereits verwendeten Datenpuffers an den Data Link Layer weiter.
- **Data Link Layer**
  Der Data Link Layer weiß, dass der Physical Layer 5 Bytes sowohl für den Header als auch für den Trailer benötigt. Er schreibt somit seinen Header ab dem 6. Byte des Puffers und fügt seinen Trailer nach den bereits durch den APL-Layer geschriebenen Datensatz hinzu. Danach wird die Größe des bereits genutzten Puffers an den Physical Layer weitergegeben.
- **Physical Layer**
  Der Physical Layer schreibt seinen Header ab dem 1. Byte des Puffers und fügt seinen Trailer am Ende des Frames an.

Empfang von Daten:

- **Physical Layer**
  Der Physical Layer empfängt einen kompletten Frame, wertet seinen Header und Trailer ab dem 1. Byte im Datenpuffer aus und gibt den Frame, sofern er gültig ist, an den Data Link Layer weiter.
- **Data Link Layer**
  Der Data Link Layer bearbeitet den Frame ab dem 6. Byte im Datenpuffer, wertet Header und Trailer aus und gibt den Frame an den APL weiter.
- **Application Protocol Layer**
  Der APL bearbeitet den Frame ab dem 11. Byte im Datenpuffer, wertet den Frame aus und gibt an das Boot Modul den empfangenen Funktionscode (welcher sich im APL-Header befindet) weiter.
- **Boot Modul**
  Das Boot Modul wertet das Kommando aus. Wenn für das empfangene Kommando auch Nutzdaten notwendig sind, so stehen diese ab dem 13. Byte im Datenpuffer.

Beim Empfangen der Daten muss gewährleistet sein, dass jeder Layer für sich erkennt, wie groß der Frame ist.

3.7.2 Application Protocol Layer (= APL)

Hier wird ein serielles Protokoll, welches in Kapitel 6 näher beschrieben wird, implementiert.

Die API umfasst folgende Funktionen:

- err_t eBLApIInit (void)
  Initialisiert das serielle Kommunikationsprotokoll. Der Prozessor muss so weit konfiguriert werden, dass alle weiteren Funktionen innerhalb des Moduls funktionieren.
- void vBLAplGetBlockSize (uint16_t * puiSizeHeaderReturn, uint16_t * puiSizePayloadReturn, uint16_t * puiSizeTrailerReturn)
  Schreibt die verwendete Header- und Trailergröße (APL+Data Link Layer + Physical Layer) in puiSizeHeaderReturn und puiSizeTrailerReturn. Die unterstützte Payloadgröße wird in puiSizePayloadReturn geschrieben.
- err_t eBLAplWaitForCommand (uint8_t uiTimeS, uint8_t * puiCommand, uint8_t * puiData, uint16_t * puiSizePayload)
  Wartet uiTimeS Sekunden auf einen Datenframe von der PC-Software. Wird kein Frame empfangen, so wird ein Fehler im Rückgabewert angezeigt. Ist uiTimeS gleich 0, so wird blockierend bis zum Empfang eines Funktionscodes gewartet.
  puiData bezeichnet den Datenpuffer, in welchen auch der eventuell empfangene Payload gespeichert wird. In puiSizePayload wird die Größe des empfangenen Payloads geschrieben.

- void vBLAplSendCommand (uint8_t uiCommand, uint8_t * puiData, uint16_t uiSizePayload)
  uiCommand beinhaltet den zu sendenden Funktionscode. Über den Puffer puiData werden entweder keine Nutzdaten oder aber Nutzdaten mit der max. unterstützten Blocksize (Rückgabewert von eBLAplGetBlockSize) übertragen. uiSizePayload gibt hierbei die Länge der bereits geschriebenen Daten an.
- err_t eBLAplClose (void)
  Hier werden alle verwendeten Ressourcen freigegeben.

3.7.3 Data Link Layer

Hier wird das eigentliche Kommunikationsprotokoll implementiert. Hier kann beispielsweise Ethernet, CAN oder ein ähnliches Protokoll definiert werden.

Beim Senden erhält der Data Link Layer die Frames vom Application Protocol Layer, bindet diesen in einen dem Kommunikationsprotokoll entsprechenden Frame ein und verschickt diesen über den Physical Layer. Beim Empfang eines Pakets verhält es sich genau umgekehrt.

Im Data Link Layer werden folgende Funktionen zur Verfügung gestellt:

- err_t eBLDatalinkInit (void)
  Initialisiert das Kommunikationsprotokoll. Das heißt, der Prozessor muss so weit konfiguriert werden, dass alle weiteren Funktionen innerhalb des Moduls funktionieren.
- void vBLDatalinkGetBlockSize (uint16_t * puiSizeHeaderReturn, uint16_t * puiSizePayloadReturn, uint16_t * puiSizeTrailerReturn);
  Schreibt die verwendete Header- und Trailergröße (Data Link Layer + Physical Layer) in puiSizeHeaderReturn und puiSizeTrailerReturn. Die unterstützte Payloadgröße wird in puiSizePayloadReturn geschrieben.
- err_t eBLDatalinkReceiveFrame (uint8_t uiTimeS, uint8_t * puiData)
  Wartet uiTimeS Sekunden auf ein zu empfangendes Paket und schreibt dieses in puiData. Ist uiTimeS gleich 0, so wird blockierend bis zum Empfang eines Frames gewartet.
- void vBLDatalinkSendFrame (uint8_t * puiData, uint16_t uiSizePayload)
  Sendet die in puiData gespeicherten Daten. uiSizePayload gibt die Größe des bereits durch den APL Layer geschriebenen Puffers an.
- err_t eBLDatalinkClose (void)
  Hier werden alle verwendeten Ressourcen freigegeben.

3.7.4 Physical Layer

Hier wird die notwendige Hardware (z.B. RS485) für die Kommunikation angesteuert. Da die Hardware bei jedem Prozessor anders angesteuert wird, muss die Implementierung der einzelnen Funktionen prozessorspezifisch erfolgen. Das entsprechende Header-File bleibt allerdings generisch.

Beim Senden erhält dieser Layer einen Frame vom Data Link Layer, fügt seinen Header und seinen Trailer hinzu und versendet diesen Frame über die verwendete Schnittstelle. Beim Empfang der Daten verhält es sich genau umgekehrt.

Im Physical Layer werden folgende Funktionen zur Verfügung gestellt:

- err_t eBLPhysicalInit (void)
  Initialisiert die nötige Hardware. Das heißt, der Prozessor muss so weit konfiguriert werden, dass alle weiteren Funktionen innerhalb des Moduls funktionieren.
- void vBLPhysicalGetBlockSize (uint16_t * puiSizeHeaderReturn, uint16_t * puiSizePayloadReturn, uint16_t * puiSizeTrailerReturn)
  Schreibt die verwendete Header- und Trailergröße (Physical Layer) in puiSizeHeaderReturn und puiSizeTrailerReturn. Die unterstützte Payloadgröße wird in puiSizePayloadReturn geschrieben.
- err_t eBLPhysicalReceiveFrame (uint8_t uiTimeS, uint8_t * puiData)
  Wartet uiTimeS auf einen Datenframe und schreibt diesen in puiData. Ist uiTimeS gleich 0, so wird blockierend bis zum Empfang eines Frames gewartet.
- void vBLPhysicalSendFrame (uint8_t * puiData, uint16_t uiSizePayload)
  Sendet einen Frame über die physikalische Schnittstelle. puiData ist ein Zeiger auf den Datenpuffer, dessen Größe durch uiSizePayload spezifiziert wird.
- err_t eBLPhysicalClose (void)
  Hier werden alle verwendeten Ressourcen freigegeben.

Wird ein Update durchgeführt, so muss der Bootloader die Firmware blockweise anfordern. D.h., der Bootloader sendet für jeden zu übertragenden Block den Befehl „NextBlock“ an die PC-Software, welche daraufhin einen Datenblock sendet. Dadurch ist gewährleistet, dass der Block erst übertragen wird, wenn der Bootloader auch bereit ist. Es kann somit nicht passieren, dass sich Datenpakete, welche über verschiede Übertragungswege laufen, gegenseitig überholen.

3.8 Crypto Modul

Dieses Modul soll über die API einen Dienst für das Entschlüsseln der Firmware zur Verfügung stellen. Dieses Modul ist rein generisch, also unabhängig vom Prozessortyp.

Die API stellt folgende Funktionen zur Verfügung:

- err_t eBLCryptoInit (void)
  Initialisiert das Modul soweit, dass alle weiteren Funktionen innerhalb dieses Moduls funktionieren.
- void vBLCryptoCipherGetBlockSize (uint16_t * puiBlocksize)

Gibt über puiBlocksize die Blockgröße des implementierten Ciphers in Bytes zurück. Wird ein Stream Cipher implementiert, so ist puiBlocksize auf 1 zu setzen.

- err_t eBLCryptoDecryptModul (uint8_t * puiBufCrypted, uint8_t * puiBufDeCrypted)
  Entschlüsselt die Daten im Puffer puiBufCrypted und schreibt sie in den Puffer puiBufDeCrypted. Die Datenpuffer müssen zumindest gleich groß dem Rückgabewert von eBLCryptoGetBlockSize sein.
- void vBLCryptoHashGetBlockSize (uint16_t * puiBlocksize)
  Retourniert über puiBlocksize die benötigte Blockgröße der Hash-Signatur.
- err_t eBLCheckHashAndStartApp (uint8_t * puiData, uint16_t uiPageSize)
  Überprüft die momentan im Flash-Speicher abgespeicherte Applikation auf Gültigkeit. Dies wird mittels einer Hashfunktion über das gesamte Firmwareimage durchgeführt. puiData zeigt auf die Position des Images im Flash-Speicher, uiPageSize übergibt die verwendete Pagegröße im Flash-Speicher, da diese Funktion Daten vom Flash-Speicher lesen muss.
- void vBLCalcCRC (uint8_t * puiData, uint16_t uiBlocksize, uint8_t * puiCRC)
  Berechnet einen CRC über die Daten in puiData, wobei uiBlocksize die Größe des Datenpuffers angibt. Der berechnete CRC wird in puiCRC geschrieben.
- err_t eBLCryptoClose (void)
  Hier werden alle verwendeten Ressourcen freigegeben.

3.9 Application

Dies ist die eigentliche Applikation, welche nach einer Gültigkeitsüberprüfung anhand einer Hash-Signatur gestartet wird, sofern keine Synchronisation mit der PC-Software erfolgt. Es muss allerdings nicht immer eine Applikation im Flash-Speicher vorhanden sein, der Bootloader würde dann unendlich lange auf ein Kommando der PC-Software warten.

4 Funktion des Bootloaders

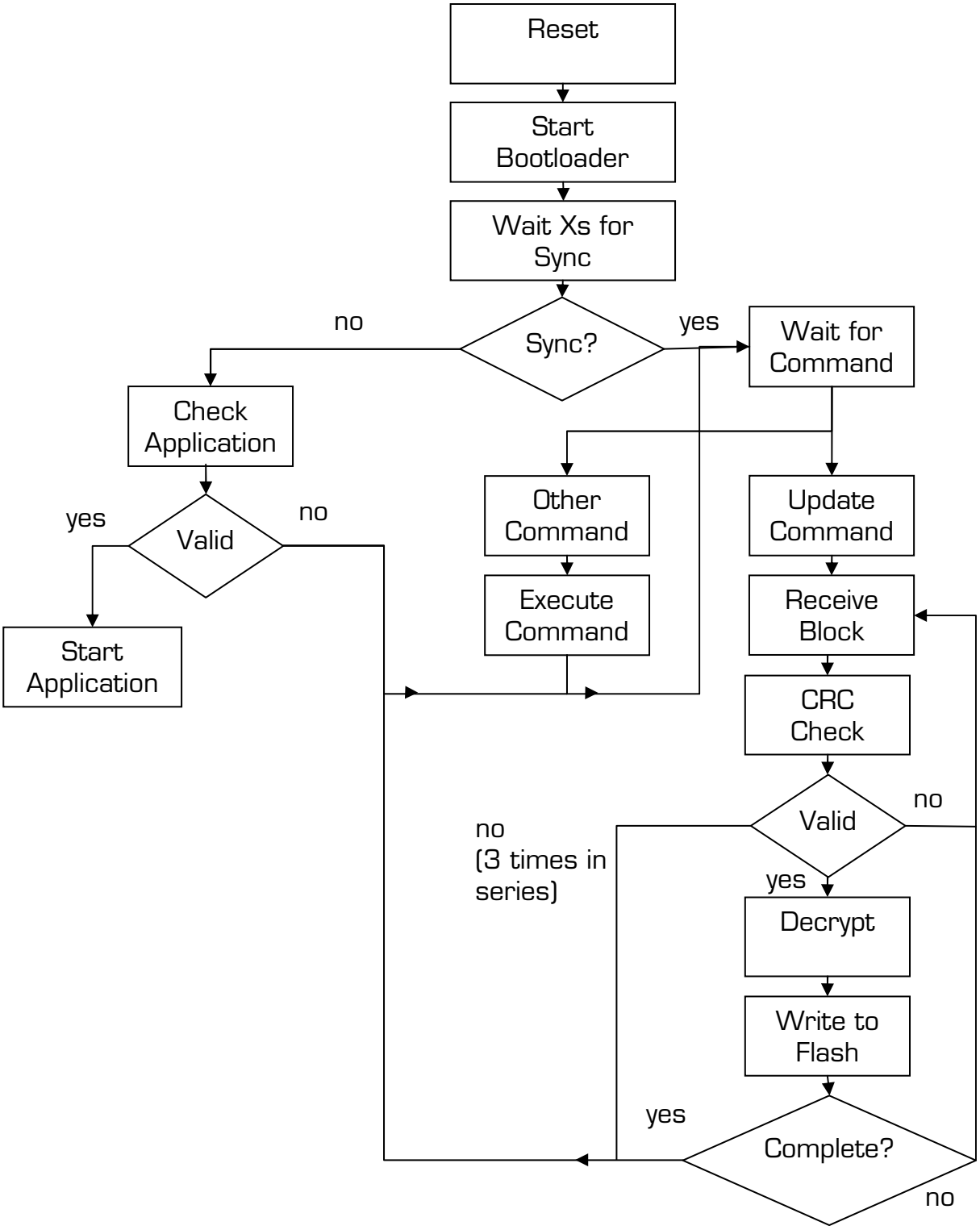

Abbildung 2: Flussdiagramm Bootloader

1) Reset des Prozessors.
2) Das Boot Modul wird gestartet.
3) Das Boot Modul wartet für beispielsweise 3s auf einen Sync-Befehl von der PC-Software über das Communication Modul.
4) Wird ein Sync-Befehl empfangen, so wird auf einen weiteren Befehl der PC-Software gewartet, welcher dann ausgeführt wird.
5) Wenn kein Sync-Befehl oder ein ungültiger Befehl empfangen wird, so wird die eventuell bereits programmierte Applikation anhand einer Hash-Signatur auf Gültigkeit überprüft. Ist keine Applikation vorhanden, oder schlägt die Gültigkeitsüberprüfung aus einem anderen Grund fehl, so wird weiterhin auf einen Befehl seitens der PC-Software gewartet. Ist eine gültige Applikation vorhanden, so wird diese gestartet.

Die Punkte 6 bis 10 werden bei Empfang des Befehls „Update“ durchgeführt:

6) Das Boot Modul lädt die Firmware blockweise über das Communication Modul in den Arbeitsspeicher.
7) Nach einer Kontrolle des CRC wird der Block durch das Crypto Modul entschlüsselt. Ist der CRC nicht korrekt, so wird der Block noch einmal übertragen (max. 3-mal, ansonsten wird der Update-Vorgang abgebrochen und auf ein weiteres Kommando der PC-Software gewartet).
8) Der Block wird über das Flash Modul in den Flash-Speicher geschrieben.
9) Überprüfung, ob Firmware bereits vollständig übertragen wurde.
10) Ist die Firmware vollständig geladen worden, so wartet der Bootloader wieder auf den nächsten Befehl der PC-Software. Ansonsten fordert der Bootloader den nächsten Block an.

5 Kryptographie

5.1 Grundlagen

5.1.1 Algorithmen und Schlüssel

Ein kryptographischer Cipher-Algorithmus ist eine mathematische Funktion, welche für die Verschlüsselung bzw. Entschlüsselung einer Datenmenge verwendet wird. Die unverschlüsselten Daten werden als Plain Text und die verschlüsselten Daten als Cipher Text bezeichnet.

Die ersten kryptographischen Algorithmen waren sogenannte „restricted algorithms“. Diese basierten auf der Idee, einen bestimmten Algorithmus auf den Daten anzuwenden, wobei die Funktionsweise des Algorithmus geheim bleiben musste. Es gab keinen Schlüssel.

Diese Idee hat allerdings einen entscheidenden Nachteil:
Erfährt eine unautorisierte Person von der Funktionsweise des Algorithmus, so wird der gesamte Algorithmus unbrauchbar.

Moderne Kryptographie verwendet Schlüssel. Hierbei ist es egal, ob die Funktionsweise des Algorithmus bekannt ist oder nicht, entscheidend ist die Kenntnis des Schlüssels. Kommt eine unautorisierte Person in den Besitz des Schlüssels, muss der verwendete Schlüssel geändert werden, der Algorithmus wird jedoch nicht unbrauchbar.

5.1.2 Symmetrische Algorithmen

Symmetrische Algorithmen zeichnen sich dadurch aus, dass sich der Schlüssel, welcher für die Verschlüsselung verwendet wird, aus dem Schlüssel für die Entschlüsselung einfach berechnen lässt. Meistens sind beide Schlüssel sogar identisch.

Bei den symmetrischen Algorithmen unterscheidet man zwischen den sogenannten Stream Ciphers und den Block Ciphers.

Stream Ciphers arbeiten mit einzelnen Datenbytes, d.h., jedes Byte wird für sich selbst verschlüsselt. Ein einfaches Beispiel hierfür sind Substitution Ciphers. Hierbei werden die einzelnen Symbole jeweils durch ein durch den Algorithmus bestimmtes anderes Symbol ersetzt.

Block Ciphers arbeiten mit Datenblöcken, d.h., ein ganzer Datenblock mit einer festgelegten Blockgröße wird verschlüsselt. Ein einfaches Beispiel hierfür sind Transposition Ciphers. Hierbei werden die Positionen der Symbole innerhalb eines Blocks durch den Algorithmus getauscht.

Sowohl Substitution Ciphers als auch Transposition Ciphers lassen sich relativ einfach implementieren, bieten jedoch nur eingeschränkte Sicherheit.

5.1.3 Public-Key (=asymmetrische) Algorithmen

Bei den Public-Key Algorithmen gibt es einen öffentlichen und einen privaten Schlüssel. Der öffentliche Schlüssel wird zur Verschlüsselung, der private Schlüssel für die Entschlüsselung verwendet.

5.1.4 Sicherheit der Algorithmen

In der Theorie ist nur ein sogenannter One-Time Pad absolut sicher, auch unter der Annahme, dass uneingeschränkte Ressourcen zur Verfügung stehen.

Hierfür wird eine Zufallsreihenfolge als Schlüssel verwendet, wobei diese genauso lang wie der zu verschlüsselnde Text sein muss.

Bsp [4]**.:**

Nachricht:	ONETIMEPAD
Schlüssel:	TBFRGFARFM
Verschl. Nachricht:	IPKLPSFHGQ

Erklärung:

O +T mod 26 = I
N + B mod 26 = P
E + F mod 26 = K

Alle anderen Algorithmen sind durch sogenannte Brute Force Attacken brechbar. Dies sind Attacken, bei welchen alle möglichen Schlüssel in systematischer Reihenfolge versucht werden.

Allerdings kann in der Praxis nicht von uneingeschränkten Ressourcen ausgegangen werden (z.B. beschränkte Rechenleistung eines PCs). Rechner müssen also über Jahre laufen, um diverse Verschlüsselungen zu brechen, auch wenn die Verschlüsselungsmethoden theoretisch nicht absolut sicher sind.

5.2 Überlegungen zum Projekt

Zu Beginn stellt sich die Frage, welches Verschlüsselungssystem für den modularen Bootloader verwendet werden soll. Theoretisch würde es folgende Möglichkeiten geben:

- **Eingeschränkter Verschlüsselungsalgorithmus**
  Es wird kein Schlüssel verwendet, sondern ein einfacher Algorithmus implementiert, welcher immer nach demselben Muster die Firmware verschlüsselt.
- **Symmetrischer Algorithmus**
  - Jeder Sensortyp erhält einen fixen Schlüssel.
- **Public Key Algorithmus**
  - Bei einem Update erzeugt der Sensor die notwendigen Schlüssel mithilfe eines Zufallgenerators (Public + Privat). Der Public Key muss an den Softwareproduzenten übertragen werden, der die Firmware mit diesem Schlüssel verschlüsselt. Beim Download auf den Sensor verwendet der Bootloader den Privat Key, welcher nur im Bootloader gespeichert ist.
  - Jeder Sensortyp erhält fixe Schlüssel (Public + Privat).

Der eingeschränkte Verschlüsselungsalgorithmus kann sehr leicht gebrochen werden, wodurch er in der Praxis unbrauchbar ist.

Die Erzeugung eines Zufallschlüssels beim Public Key Algorithmus wäre das Gegenbeispiel dazu und somit die sicherste Variante. Jedoch wäre ein Updatevorgang zu aufwendig. Erscheint eine neue Softwareversion, so wird die Firmware an zahlreiche Kunden gleichzeitig ausgeliefert. Es würde somit ein enormer Aufwand entstehen, da von jedem Kunden der öffentliche Schlüssel ermittelt werden muss.

Die sinnvollste Überlegung im Rahmen des vorliegenden Projekts ist es also, jeden Sensortyp mit einem fixen Schlüssel auszustatten. Um sich zwischen einem symmetrischen oder einem asymmetrischen Algorithmus entscheiden zu können, müssen verschiedene Verschlüsselungsalgorithmen näher untersucht werden.

5.2.1 Mögliche Verschlüsselungen im Überblick

5.2.1.1 RSA

Bei den Public-Key (=asymmetrischen) Verschlüsselungssystemen gilt unter anderem das RSA-Verfahren (benannt nach den 3 Erfindern: Ron Rivest, Adi Shamir, Leonard Adleman) als sehr sicher.

Dieser Algorithmus kann sowohl zur Verschlüsselung als auch für digitale Signaturen verwendet werden. RSA ist ein Algorithmus, welcher sehr einfach zu verstehen und zu implementieren ist.

RSA baut darauf auf, dass hohe Zahlen schwer zu faktorisieren sind.

Berechnung der Schlüssel:

- Ausgangspunkt des Algorithmus sind 2 hohe Primzahlen (mind. 100 Stellen), wobei beide Primzahlen gleiche Länge haben sollten.
- Das Produkt $n = p * q$ wird berechnet.
- Der Encryption Key e wird so gewählt, dass e und $(p-1)*(q-1)$ relativ prim (= teilerfremd) zueinander sind.
- Mithilfe des euklidischen Algorithmus kann der Decryption Key berechnet werden: $d = e^{-1} \mod ((p-1)(q-1))$
- e und n sind der öffentliche Schlüssel, d ist der private Schlüssel.
- Die beiden Primzahlen p und q werden nicht mehr benötigt und müssen daher gelöscht werden (dürfen keinesfalls bekannt werden).

Nachricht verschlüsseln:

- Die Nachricht muss in numerische Blöcke c_i, welche kleiner als n sind, aufgeteilt werden.
- Verschlüsseln der Nachricht mittels
  $c_i = m_i^e \mod n$

Nachricht entschlüsseln:

- Es wird wieder jeder Block einzeln betrachtet und entschlüsselt:
  $m_i = c_i^d \mod n$

Sicherheit: Gebrochen kann der Algorithmus werden, wenn es gelingt n zu faktorisieren, d.h., in die Primfaktoren zu zerlegen. Bisher ist aber keine Möglichkeit

gefunden worden, dies mit einem vertretbaren Aufwand durchzuführen. Eine weitere Möglichkeit wäre eine Brute Force Attacke, indem man versucht, den Term $(p-1)*(q-1)$ zu erraten.

Geschwindigkeit: Wird der Algorithmus mittels Hardware realisiert, so ist RSA ca. 1000-mal langsamer als DES (Kapitel 5.2.1.2). Bei einer Softwarelösung ist der Algorithmus ca. 100-mal langsamer als DES.

Patent: Der RSA-Algorithmus wurde lediglich in den Vereinigten Staaten patentiert. Aber auch dieses Patent ist im Jahr 2000 ausgelaufen.

5.2.1.2 DES, 3DES

Bei DES (Data Encryption Standard) handelt es sich um einen symmetrischen Block Cipher, wobei mehrere Substitutionen und Permutationen nacheinander durchgeführt werden. Es wird ein 56 Bit Schlüssel verwendet. Der gesamte Ablauf, welcher in Abbildung 3 gezeigt wird, wird 16-mal durchgeführt.

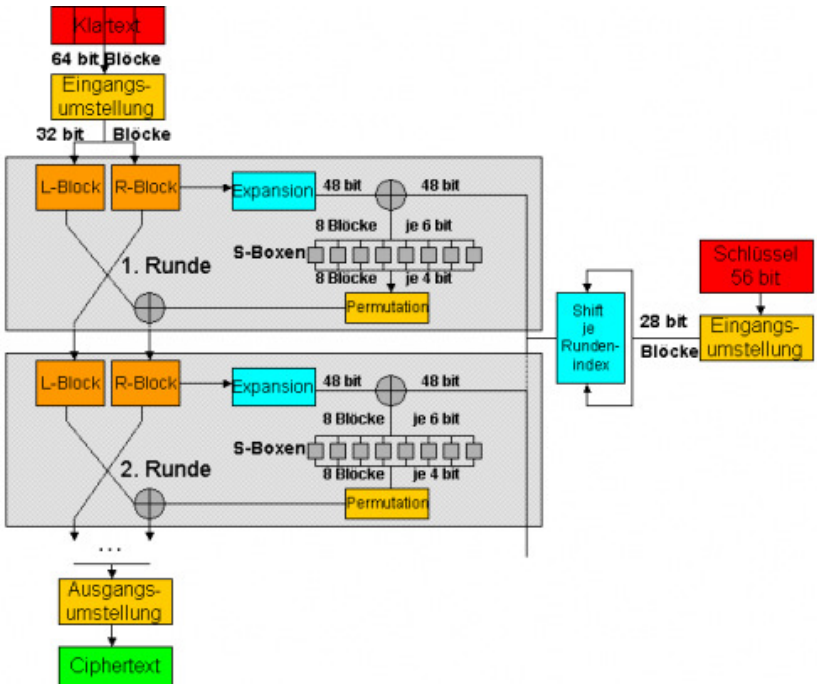

Abbildung 3: Data Encryption Standard [5]

Triple-DES (=3DES) ist der Nachfolger von DES. 3DES basiert auf dem gleichen Prinzip wie DES, verwendet jedoch 2 Schlüssel in 3 Durchgängen. Eine Verschlüsselung in Triple-DES hat folgenden Ablauf:

- Verschlüsselung mit Key 1
- Entschlüsselung mit Key 2
- Verschlüsselung mit Key 1

Sicherheit: DES kann auf Grund des kleinen Schlüssels mit 56 Bits sehr leicht durch Brute Force Attacken gebrochen werden. „Deep Crack", ein Superrechner welcher speziell für die Kryptographie entwickelt wurde, konnte bereits im Jahr 1998 eine DES verschlüsselte Nachricht innerhalb von 56 Stunden brechen. 3DES gilt hingegen als sicher.

Geschwindigkeit: Während eine DES Verschlüsselung relativ schnell durchgeführt werden kann, benötigt 3DES aufgrund der 3-fachen Verschlüsselung bereits eine beträchtliche Rechenzeit.

Patent: Es gibt kein Patent, der Algorithmus kann frei verwendet werden.

5.2.1.3 AES

AES (Advanced Encryption Standard) ist ein symmetrischer Block Cipher, der als Nachfolger von DES bzw. 3DES entwickelt wurde. Dieser sollte die Schwachstellen der beiden Algorithmen beseitigen. Es wurden 3 Standards veröffentlicht, wobei immer Blöcke mit 128 Bits verschlüsselt werden. Es werden längere Schlüssel als bei DES mit 128, 192 oder 256 Bits verwendet.

	Schlüssellänge	**Blockgröße**	**Runden**
AES-128	128 Bits	128 Bits	10
AES-192	192 Bits	128 Bits	12
AES-256	256 Bits	128 Bits	14

Tabelle 3: AES-Standards

AES verwendet den Rijndael Algorithmus. Hierfür wird jedes Byte als Polynom dargestellt:

$$a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = \sum_{i=0}^{7} a_i x^i$$

Es werden Additionen und Multiplikationen auf die einzelnen Polynome angewandt:

- **Addition:** Einfache XOR-Verknüpfung (mod 2) der Koeffizienten
  - $1 \oplus 1 = 0$
  - $1 \oplus 0 = 1$
  - $0 \oplus 0 = 0$
- **Multiplikation:** Das Polynomprodukt wird unter der Verwendung der bereits erklärten Addition ermittelt. Das Ergebnis der Multiplikation wird modulo einem irreduziblen Polynom gerechnet.
  Irreduzibles Polynom:
  Ein irreduzibles Polynom kann nicht durch das Produkt zweier nicht invertierbarer Polynome dargestellt werden.

Inverses Element:
Wird ein Element mit seinem inversen Element durch eine Rechenoperation verknüpft, so erhält man das neutrale Element.
Neutrales Element:
Wird ein Element mit seinem neutralen Element durch eine Rechenoperation verknüpft, so erhält man wieder das ursprüngliche Element.
Das verwendete irreduzible Polynom in AES:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

Es werden folgende Transformationen auf die Blöcke angewandt:

- **SubBytes():** Eine nichtlineare Byte-Substitution, es wird eine Substitutions-Tabelle (S-Box) verwendet.
- **ShiftRows():** Die Zeilen eines Blocks werden zyklisch verschoben.
- **MixColumns():** Spalten werden vertauscht.
- **AddRoundKey():** Die Daten werden mit dem Schlüssel XOR-verknüpft.

Sicherheit: AES gilt zurzeit als sicher.

Geschwindigkeit: Die Geschwindigkeit ist schneller als bei 3DES und vergleichbar mit Blowfish (Kapitel 5.2.1.4).

Patent: Es gibt kein Patent, der Algorithmus kann frei verwendet werden.

5.2.1.4 Blowfish

Blowfish ist ein unpatentierter Algorithmus und wurde von Bruce Schneier entwickelt. Folgende Design-Kriterien wurden hierbei beachtet:

- **Schnell:** Auf 32-Bit Prozessoren werden nur 26 Taktzyklen pro Byte benötigt.
- **Kompakt:** Weniger als 5 kBytes Speicheraufwand.
- **Einfach:** Es werden nur einfache Operationen wie Addition, XOR und Lookup Tables verwendet.
- **Variable Sicherheit:** Schlüssellänge ist variabel.

Blowfish ist ein 64 Bit Block Cipher mit variabler Schlüssellänge mit bis zu 448 Bits. Der Algorithmus unterteilt sich in folgende Teile:

- **Schlüsselexpansion:** Ein Schlüssel mit bis zu 448 Bits wird in mehrere Subkeys mit insgesamt 4168 Bytes umgewandelt.
- **Datenverschlüsselung:** Es werden 16 Runden durchlaufen, wobei folgende Funktionen durchgeführt werden:
  - Schlüsselabhängige Permutation
  - Schlüssel- und datenabhängige Substitution

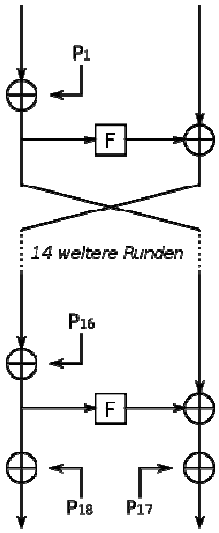

Abbildung 4: Blowfish [6]

Sicherheit: Mit voller Rundenlänge ist bis heute kein effizienter Angriff auf den Algorithmus bekannt.

Geschwindigkeit: Schnell auf 32 Bit Prozessoren, auf kleineren Prozessoren können höhere Laufzeiten auftreten.

Patent: Es gibt kein Patent, der Algorithmus kann frei verwendet werden.

5.2.1.5 Twofish

Twofish ist der Nachfolger von Blowfish und wurde ebenfalls unter der Mitwirkung von Bruce Schneier entwickelt. Auch hier handelt es sich um einen Block Cipher, der allerdings im Gegensatz zu Blowfish Schlüssel mit festen Schlüssellängen (128, 192 und 256 Bit) verwendet. Es werden 16 Runden durchlaufen, wobei folgende Operationen verwendet werden:

- Exklusive-Oder Verknüpfung
- Addition modulo 32
- Substitutionstabellen

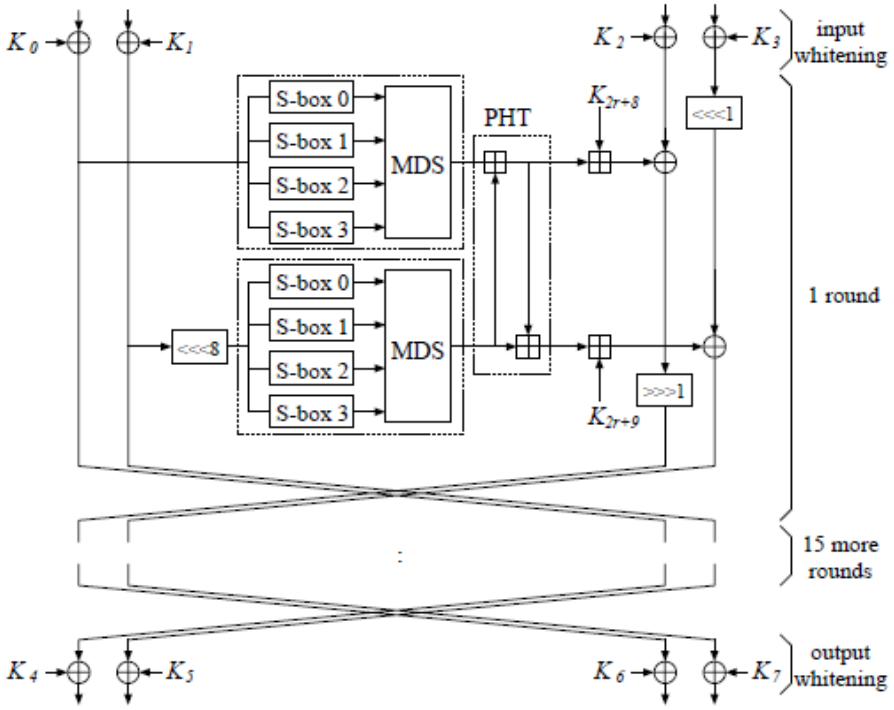

Abbildung 5: Twofish [7]

Sicherheit: Twofish gilt zurzeit als sicher, ist sogar etwas sichererer als Blowfish.

Geschwindigkeit: Ist etwas langsamer als Blowfish.

Patent: Es gibt kein Patent, der Algorithmus kann frei verwendet werden.

5.2.1.6 XXTEA

XXTEA (= Corrected Block TEA) ist der Nachfolger von XTEA (= eXtended TEA) und TEA (= Tiny Encryption Algorithm). XXTEA wurde von Roger Needham und David Wheeler entwickelt und 1998 publiziert.

Auch hier handelt es sich um einen Block Cipher, wobei variable Blockgrößen (Vielfache von 32 Bit, Minimum liegt bei 64 Bits) verwendet werden. Der Algorithmus verwendet einen Schlüssel mit einer Länge von 128 Bits.

Dieser Algorithmus kann sehr einfach implementiert werden und benötigt weniger Ressourcen als die anderen vorgestellten Algorithmen.

Es werden folgende Operationen verwendet:

- Exklusive-Oder Verknüpfung
- Addition-Modulo
- Shifting

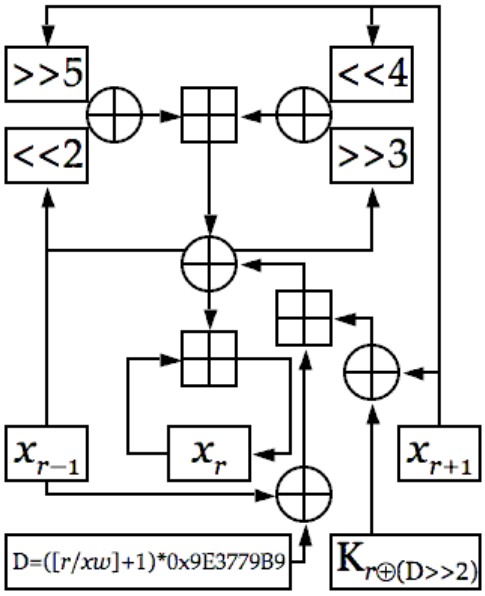

Abbildung 6: XXTEA [8]

Sicherheit: Gilt zurzeit als sicher.

Geschwindigkeit: Vergleichbar mit Blowfish.

Patent: Es gibt kein Patent, der Algorithmus kann frei verwendet werden.

5.2.1.7 Auswahl des Algorithmus

	RSA	AES	DES	3DES	Blowfish	Twofish	XXTEA
Sicherheit	+	+	-	+	+	+	+
Geschwindigkeit	-	+	+	-	+	+	+
Patent	+	+	+	+	+	+	+

Tabelle 4: Algorithmen – Ein Überblick

Tabelle 4 gibt einen Überblick über die vorgestellten Algorithmen. Der Ressourcenverbrauch ist bei allen Algorithmen in etwa gleich, lediglich XXTEA bietet den Vorteil eines etwas geringeren Ressourcenverbrauches. Alle vorgestellten Algorithmen können auf den größeren Prozessoren, wie etwa den im Rahmen dieser Arbeit verwendeten AVR32, ohne Probleme implementiert werden.

Für kleinere Prozessoren, wie z.B. den AVR8, sind die vorgestellten Algorithmen allerdings zu ressourcenaufwendig (man würde hier zusammen mit der eigentlichen Applikation knapp an die Grenzen des max. verfügbaren Speichers kommen). Auf diesen Prozessoren müsste ein einfacherer Algorithmus implementiert werden, wobei man eine eingeschränkte Sicherheit in Kauf nehmen muss.

Möglichkeiten für kleinere Prozessoren:

- RC4: Stream Cipher, patentiert von RSA-Security
- RC5: Block Cipher, in den Vereinigten Staaten patentiert
- Substitution Cipher
- Caesar Cipher
- Einfache XOR-Verknüpfung mit einem Key

Public Key vs. symmetrischer Cipher

Seitens der PC-Software hätte eine Public-Key Verschlüsselung den Vorteil, dass die PC-Software nur den öffentlichen Schlüssel für die Verschlüsselung benötigt. Der private Schlüssel müsste nur auf dem Sensor gespeichert werden.

Public Key Verschlüsselungen sind aber im Allgemeinen langsamer als symmetrische Algorithmen. Beispielsweise ist, wie bereits erwähnt, die Softwarelösung von RSA ca. 100-mal langsamer als DES. Daher erweist sich ein symmetrischer Cipher in diesem Projekt als sinnvoller.

Gewählter Algorithmus: AES-256 – dieser Algorithmus wird bereits in zahlreichen Projekten erfolgreich verwendet und es lassen sich auch mehrere fertige Bibliotheken im Internet finden.

5.3 Kryptographische Hashfunktionen

Um zu gewährleisten, dass die in den Speicher zu programmierende Applikation von S::can entwickelt wurde und der Quellcode auch in keiner Weise verändert wurde, wird eine kryptographische Hashfunktion verwendet. Damit ist sichergestellt, dass keine schädliche Software durch den Bootloader gestartet werden kann.

Ein typisches Merkmal einer solchen Hashfunktion ist, dass sie selbst sehr leicht berechnet werden kann. Die Umkehrfunktion ist allerdings nur sehr schwer oder auch gar nicht berechenbar. Ein einfaches Beispiel wäre die mathematische Funktion $y(x) = x^3$. Diese Funktion kann auch ohne Hilfsmittel wie einen Taschenrechner sehr einfach berechnet werden. Doch die Umkehrfunktion $x(y) = \sqrt[3]{y}$ ist bereits um einiges schwieriger zu berechnen.

Funktion:
Es wird über das unverschlüsselte Firmwareimage eine Hash-Signatur berechnet und im Anhang angefügt. Das erweiterte Image (mit Hash-Signatur) wird anschließend verschlüsselt und so an den Kunden geschickt.

Mathematisch lässt sich dies wie folgt beschreiben:
$C = E_k(x \parallel h(x))$

Der Bootloader berechnet nach Empfang des Firmwareimages erneut die Hash-Signatur und kann somit feststellen, ob die Software gültig ist. D.h., es kann keine Software eines fremden Herstellers über den Bootloader gestartet werden.

„Ein Hash Algorithmus gilt im kryptographischen Sinne als gebrochen, wenn es eine Methode oder einen Angriff gibt, der schneller eine Kollision herbeiführt als eine Brute Force Attacke." [SCHNEIER]

Unter einer Kollision versteht man das Finden zweier unterschiedlicher Datensätze, welche beide zur selben Hash-Signatur führen.

Im den folgenden Kapiteln werden zwei bekannte Hashfunktionen näher erklärt.

5.3.1 SHA

Bei SHA (Secure Hash Algorithm) handelt es sich um eine Gruppe standardisierter kryptologischer Hash-Funktionen, welche jeweils eine eindeutige Hash-Signatur über einen Datenblock berechnen. Es soll unmöglich sein, zwei unterschiedliche Nachrichten mit derselben SHA-Signatur zu finden. SHA wurde von NIST (National Institute of Standards and Technology) in Zusammenarbeit mit der NSA (National Security Agency) entwickelt.

Der Standard beschreibt 5 Hash Algorithmen:

Algorithm	Message Size	Block Size [Bits]	Message Digest SizeBits [Bits]
SHA-1	$< 2^{64}$	512	160
SHA-224	$< 2^{64}$	512	224
SHA-256	$< 2^{64}$	512	256
SHA-384	$< 2^{128}$	1024	384
SHA-512	$< 2^{128}$	1024	512

Tabelle 5: SHA-Standards

Die Standards SHA-224, SHA-256, SHA-384 und SHA-512 werden zusammen mit dem Sammelbegriff SHA-2 bezeichnet. Mittlerweile wurde bereits zu einem Wettbewerb aufgerufen, ähnlich wie bei AES, SHA-3 zu entwickeln.

Funktion:
Es werden 2 grundlegende Phasen bei der Berechnung der Hash-Signatur unterschieden:

- **Preprocessing**
- **Hash-Function**

Beim Preprocessing (= Vorverarbeitung) werden wieder 3 Phasen unterschieden:

- **Packing:** Fülldaten werden eingefügt, damit die Größe des Datensatzes ein Vielfaches der Blockgröße ist.
- **Parsing:** Der Datensatz wird in gleich große Blöcke aufgeteilt.
- **Initial Hash Value:** Initial Wert der Hashfunktion wird bestimmt (ist abhängig vom jeweiligen Standard).

Für die Berechnung der Hash-Signatur werden verschiedene Konstanten und Funktionen verwendet, welche sich bei den einzelnen Standards unterscheiden.

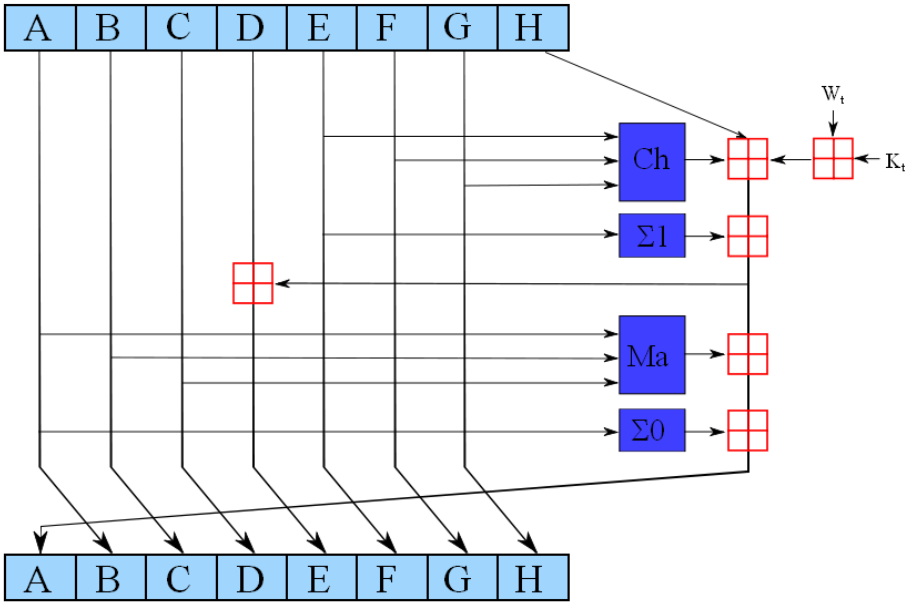

Abbildung 7: SHA-2 [9]

5.3.2 MD5

MD5 (MD: Message Digest) wurde von Ron Rivest entwickelt. MD5 gilt als Nachfolger von MD4 und soll dessen Schwachstellen beseitigen. Das Design von MD5 ist stark an das Design von MD4 angelehnt, jedoch etwas komplizierter gestaltet.

MD5 arbeitet mit Blöcken zu 512 Bits, welche in 16*32 Bit Datenblöcke aufgeteilt werden. Aus diesen Blöcken wird eine 128 Bit Hash-Signatur berechnet.

Funktion:

- Es werden 4*32 Bit Variablen (chaining variables) initialisiert:
  - A=0x01234567
  - B=0x89abcdef
  - C=0xfedcba98
  - D=0x76543210
- Hauptschleife aus 4 Runden:

- Jede Runde benutzt unterschiedliche Operationen, jeweils 16 mal:
  - Nichtlineare Funktionen - es gibt 4 Funktionen, in jeder Runde wird eine andere verwendet
    - $F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$
    - $G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$
    - $H(X,Y,Z) = X \oplus Y \oplus Z$
    - $I(X,Y,Z) = Y \oplus (X \vee \neg Z)$
  - Modulare Addition
  - Linksrotation

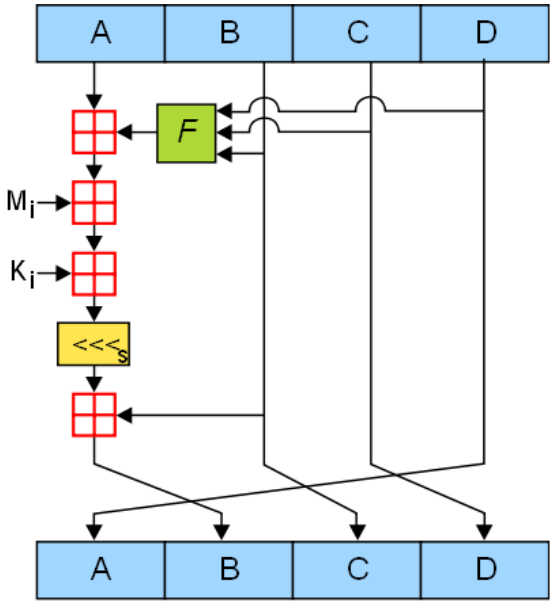

Abbildung 8: MD5 [10]

5.3.3 Fazit

MD5 gilt mittlerweile nicht mehr als sicher, da es mit überschaubarem Aufwand möglich ist, die Hash-Signatur zu brechen.
Daher wird in diesem Projekt SHA verwendet werden. SHA-1 kommt nicht in Frage, da dieser Algorithmus ebenfalls bereits als unsicher gilt. Somit bleibt die Wahl zwischen den SHA-2 Varianten. SHA-224 ist noch sehr neu und noch nicht sehr verbreitet. Daher wird im Rahmen dieser Arbeit SHA-256 verwendet werden, da diese Standardisierung am ressourcensparendsten ist und trotzdem ausreichende Sicherheit bietet.

6 Kommunikationsprotokoll

Hier wird ein einfaches serielles Protokoll im APL Layer erstellt. Das Frameformat ist wie folgt:

1 Byte	1 Byte	0-255 Bytes
CMD	**Length**	**Payload**

Tabelle 6: Frame Format

Die einzelnen Felder haben folgende Bedeutung:

- **CMD**
  Beinhaltet den aktuellen Funktionscode.
- **Length**
  Gibt die Länge des Payloads in Byte an.
- **Payload**
  Nutzdaten

Es gibt somit 2 Typen gültiger Frames:

- **Typ 1**

1 Byte	1 Byte
CMD	**Length = 0**

Tabelle 7: APL Frame Typ 1

- **Typ 2**

1 Byte	1 Byte	0-255 Bytes
CMD	**Length**	**Payload**

Tabelle 8: APL Frame Typ 2

Tabelle 9 zeigt die unterstützten Funktionscodes für das serielle Protokoll. Die Spalten BL (=Bootloader) und PC (=PC-Software) geben jeweils an, von welcher Kommunikationsseite der Befehl verwendet werden kann.

Befehl	Frame-Typ	Bedeutung	Hex Code	BL	PC
Sync	1	Zur Synchronisation mit dem Bootloader.	0x01	N	J
Ack	1	Acknowledgement	0x02	J	J
SetFlashID	1	Setzt die aktuelle Flash-ID	0x03	J	N
NextBlock	1	Fordert nächsten Datenblock beim Update an.	0x04	J	N
SameBlock	1	Fordert wiederholt denselben Datenblock beim Update an.	0x05	J	N
VerifySuccess	1	Gültige Applikation (Antwort auf Verify)	0x06	J	N
VerifyFailed	1	Ungültige Applikation (Antwort auf Verify)	0x07	J	N
Update	1	Update Kommando	0x08	N	J
Data	2	Daten werden übertragen (Firmwareimage)	0x9	N	J
DataEnd	2	Daten werden übertragen (Firmwareimage). Dies ist der letzte Datenblock des Firmwareimages.	0x10	N	J
Verify	1	Überprüfung der Applikation auf Gültigkeit.	0x11	N	J
Reset	1	Reset des Controllers.	0x12	N	J
EraseFlash	1	Löscht den Flash-Speicher.	0x013	N	J
Error	1	Bootloader teilt der PC-Software mit, dass ein Fehler aufgetreten ist (z.B. beim Update).	0x014	J	N

Tabelle 9: Funktionscodes

7 Firmware Image

Tabelle 10 zeigt den Aufbau eines Firmware-Images, so wie es von der PC-Software an den Bootloader gesendet wird.

Firmware Version	Bootloader Version
CPU Type	Length
Address Offset	
Data	
Hash-Signatur	

Tabelle 10: Firmware Image

Die Felder haben folgende Bedeutung:

- **Firmware Version (1 Byte)**
  Die Versionsnummer der zu programmierenden Firmware.
- **Bootloader Version (1 Byte)**
  Die Versionsnummer des Bootloaders. Dadurch ist der Verschlüsselungsalgorithmus bestimmt.
- **CPU Type (1 Byte)**
  Spezifiziert die verwendete Plattform.
- **Length (8 Bytes)**
  Die Länge des Feldes „Data“.
- **Address Offset (8 Bytes)**
  Offset für die Adresse im Flash-Speicher, wo das Firmware-Image abgespeichert werden soll.
- **Data**
  Das eigentliche Firmwareimage.
- **Hash-Signatur (256 Bytes)**
  SHA-256 Hash-Signatur

Die Größe des kompletten Firmwareimages muss ein Vielfaches der Blockgröße des verwendeten Ciphers sein. Ansonsten kann keine korrekte Verschlüsselung über das gesamte Image erfolgen. Wird die Größe nicht erreicht, wird das Feld „Data“ mit

einzelnen Bits mit logisch 1 aufgefüllt (diese werden dann auch bei einem Firmwareupdate in den Flash-Speicher geschrieben).

7.1 Bearbeitung des Firmwareimages im Bootloader

Das Firmwareimage wird über die PC-Software blockweise an den Bootloader gesendet.

Update:

- Im Flash-Speicher befindet sich ein Firmwareinformationsbereich, welcher CRC-geschützt ist. Zu Beginn des Updates wird der CRC im Flash-Speicher auf einen ungültigen Wert gesetzt.
- Der Bootloader betrachtet die ersten empfangenen 19 Bytes des Images als Firmware-Header. Die Informationen aus dem Header werden vorerst nur in Variablen abgespeichert.
- Die nächsten Blöcke beinhalten das Firmwareimage und werden somit in den Flash-Speicher geschrieben.
- Die Länge des Firmwareimages ist bekannt (aus dem Firmware-Header), der Bootloader weiß somit, wann die Hash-Signatur im Datenstrom beginnt.
- Der Firmware-Header wird in den Firmwareinformationsbereich geschrieben.
- Die empfangene Hash-Signatur wird in den Firmwareinformationsbereich geschrieben.
- Es wird ein CRC über den Firmwareinformationsbereich (also über den Firmware-Header und über die Hash-Signatur) berechnet und ebenfalls im Firmwareinformationsbereich abgespeichert.

Start einer Applikation:

- Bootloader wird gestartet.
- Es kommt kein Befehl von der PC-Software, es wird daher versucht, die Firmware zu starten.
- CRC über den Firmwareinformationsbereich wird überprüft.
- Wenn der CRC korrekt ist, wird die Hash-Signatur über den Firmware-Header (befindet sich im Firmwareinformationsbereich) und über das im Flash-Speicher gespeicherte Image berechnet.
- Die Hash-Signatur wird mit der im Firmwareinformationsbereich gespeicherten Hash-Signatur verglichen.
- Stimmen beide Hash-Signaturen überein, so wird die Applikation gestartet.

8 Beispiele

8.1 Beispiel für Verwendung auf 8-Bit AVR mit RS485

8.1.1 Communication Modul

Für RS485 ist kein festes Protokoll definiert. Im Data Link Layer kann somit jedes Protokoll verwendet werden, welches unabhängig von der physikalischen Schnittstelle ist. Der Application Protocol Layer beinhaltet das serielle Protokoll aus Kapitel 6.

Application Protocol Layer
Data Link Layer
RS 485

Tabelle 11: Aufbau Communication Modul Bsp. 1

Data Link Layer:

Im Data Link Layer kann beispielsweise ein eigenes Protokoll definiert werden:

1 Byte	1 Byte	0-255 Bytes	1 Byte
Address	Length	Payload	CRC

Tabelle 12: Data Link Layer - Protokoll

Die Felder haben folgende Bedeutung:

- **Address**
  Die Adresse des Kommunikationspartners.
- **Length**
  Gibt die Länge des Payloads an.
- **Payload**
  Nutzdaten
- **CRC**
  Prüfsumme.

Physical Layer:

Hier wird ein Treiber für die RS485-Schnittstelle benötigt. RS485 benötigt 2 Datenleitungen. Auf einer Datenleitung wird das Signal in positiver Logik (log. 0: 0V, log. 1: 5V) übertragen, auf der anderen Leitung wird das Signal invertiert (log. 0: 5V, log. 1: 0V) übertragen. Beide Leitungen sind sowohl mit dem Empfänger als auch mit dem Sender verbunden.

8.2 Beispiel für Verwendung auf Cortex M3 mit Ethernet

8.2.1 Communication Modul

Ethernet definiert die ersten beiden Layer des Communication Moduls: Den Physical- und den Data Link Layer. Der Application Protocol Layer beinhaltet wieder das serielle Protokoll aus Kapitel 6.

Application Protocol Layer
MAC-Layer
CSMA/CD

Tabelle 13: Aufbau Communication Modul Bsp. 2

Application Protocol Layer:

Senden der Daten:
Sendet der Bootloader einen Funktionscode an die PC-Software, so wird in diesem Layer ein APL Frame (Kapitel 6) gebildet. Sendet der Bootloader beispielsweise ein Acknowledgement, sieht der Frame wie folgt aus:

CMD	Length
0x02	**0x00**

Tabelle 14: Beispiel Ack-Frame

Dieser Frame wird dann an den Data Link Layer weitergeleitet.

Empfang der Daten:
Empfängt der Bootloader Daten von der PC-Software, so erhält der Application Protocol Layer ebenfalls einen Frame wie in Kapitel 6 spezifiziert vom Data Link Layer (in der Ethernet Spezifikation wird dieser Layer auch MAC-Layer genannt). Sendet die PC-Software z.B. während eines Updates einen „Data"-Befehl an den Bootloader, sieht der Frame wie folgt aus:

CMD	Length	Payload
0x09	**0x40**	**Data**

Tabelle 15: Beispiel Update-Frame

Der Bootloader erkennt am Funktionscode (0x09), dass es sich um ein Datenpaket für den Update-Vorgang handelt und wird somit den Payload entschlüsseln und anschließend in den Flash-Speicher schreiben.

Data Link Layer:

Im Data Link Layer (MAC-Layer) werden die Ethernet Frames generiert. Ein Frame hat folgendes Format:

7 Bytes	1 Byte	6 Bytes	6 Bytes	2 Bytes	46-1500 Bytes	4 Bytes
Preamble	**SFD**	**Destination Address**	**Source Address**	**Length / Type**	**Data**	**FCS**

Tabelle 16: Ethernet Frame Version 2.0

- **Preamble:** Wird zur Synchronisation verwendet.
- **Start Frame Delimiter:** Zeigt den Start des Frames an (1010.1011b).
- **Destination Address:** Zieladresse
- **Source Address:** Quelladresse
- **Length/Type:** Bedeutung hängt vom numerischen Wert ab. Ist der Wert größer als 0x0600, so handelt es sich um das Type-Feld. Das Type-Feld gibt Auskunft über die nachfolgenden Daten. Ist der Wert kleiner als 0x0600, so wird das Feld als Length-Feld betrachtet und gibt somit die Größe des Payloads an.
- **Data:** Nutzdaten
- **Frame Check Sequence:** Prüfsumme

Senden der Daten:
Der Data Link Layer erhält einen Frame vom Application Protocol Layer mit dem entsprechenden Funktionscode (siehe Tabelle 6). Dieser Frame wird nun als Payload behandelt, welcher entsprechend auf einen Ethernet Frame abgebildet wird und anschließend an den Physical Layer weitergeleitet wird.

Empfang der Daten:
Der Data Link Layer erhält einen Ethernet Frame vom Physical Layer, entfernt die Kontrollbytes (es bleibt also ein Frame, wie in Kapitel 6 spezifiziert), und leitet diesen an den Application Protocol Layer weiter.

Physical Layer:

Hier erfolgt der Zugriff auf das physikalische Medium über den CSMA/CD-Algorithmus (Carrier Sense Multiple Access/Collision Detection).

Jene Stationen, welche Daten senden möchten, horchen vorerst das Medium ab. Findet keine Datenübertragung statt, so senden sie. Es kann jedoch vorkommen, dass 2 Stationen gleichzeitig mit der Übertragung beginnen. Wird eine Kollision festgestellt, so warten beide Stationen eine zufällig ausgewählte Zeit und versuchen die Übertragung erneut.

Senden der Daten:
Der Physical Layer erhält einen Ethernet-Frame vom Data Link Layer. Dieser Frame wird nun übertragen, sobald das Medium frei ist.

Empfang der Daten:
Der Physical Layer erhält einen Ethernet Frame über das Medium und leitet diesen an den Data Link Layer weiter.

9 PC-Software (UpdateSW)

Diese Software ist für das Durchführen eines Firmwareupdates beim Kunden gedacht und wird in Java implementiert.

9.1 Funktionen

- **Update:** Die Firmware kann auf einen Prozessor programmiert werden.
- **Verify:** Vergleicht die im Flash-Speicher gespeicherte Applikation mit einem am PC gespeicherten Firmwareimage.
- **Reset:** Startet den Controller neu.
- **Erase:** Löscht den Flash-Speicher.

9.1.1 Kommunikation mit Sensor bei Firmwareupdate

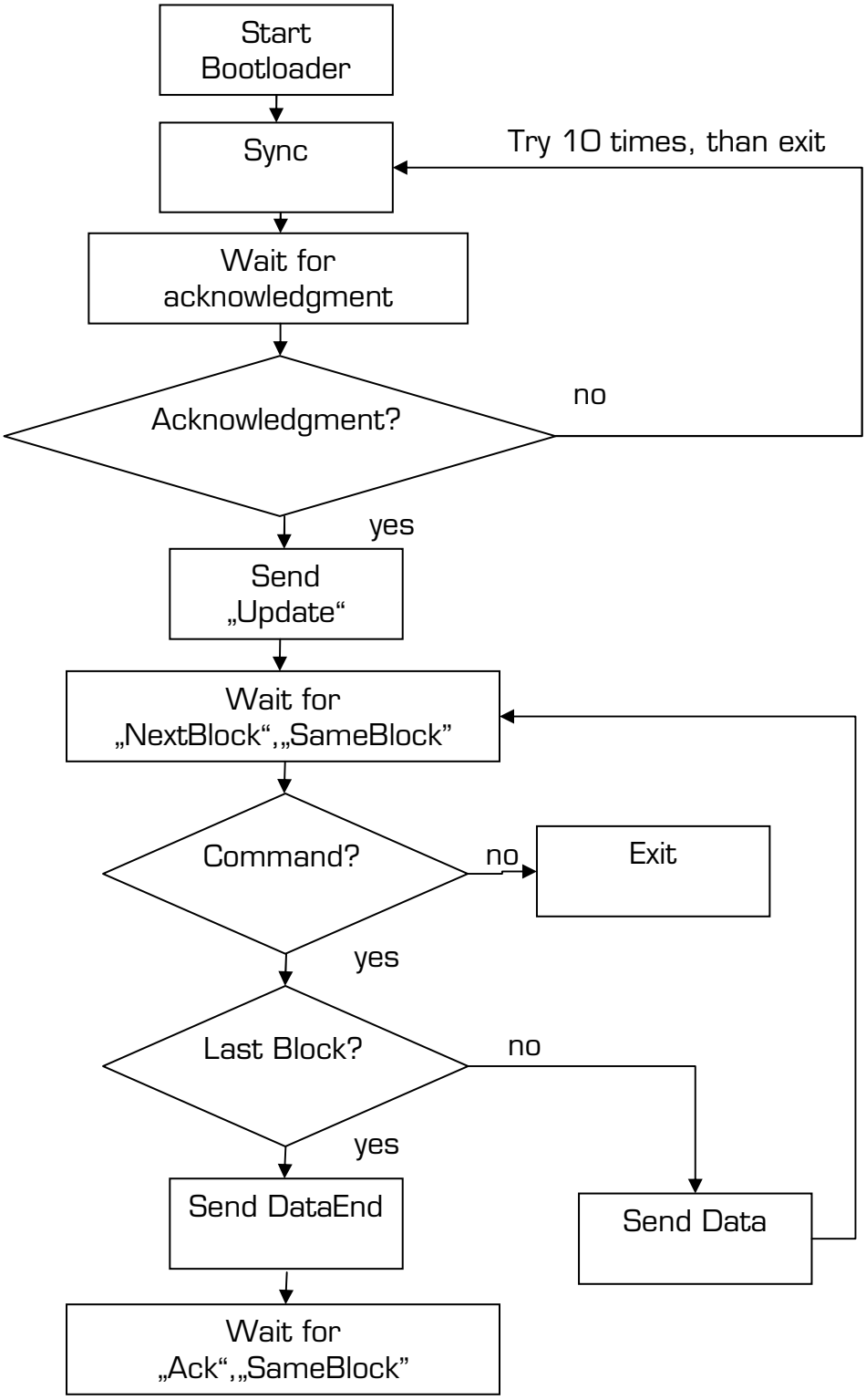

Abbildung 9: Ablauf des Updates im PC-Programm

Wird versucht, eine Verbindung mit dem Bootloader aufzubauen, so sendet die PC-Software bis zu 10-mal den Sync-Befehl und wartet jedes Mal nach dem Senden auf ein Acknowledgement des Bootloaders. Kommt dieses Acknowledgement nicht, so wird der nächste Sync-Befehl gesendet. Bei einer geglückten Synchronisation mit dem Bootloader kann die PC-Software einen Update-Befehl an den Bootloader schicken und es kann mit dem Übertragen der einzelnen Datenblöcke begonnen werden. Hierfür wartet die PC-Software jeweils auf ein „NextBlock“ bzw. auf ein „SameBlock“-Kommando. Wird ein „SameBlock“-Kommando empfangen, so ist beim Senden des vorhergehenden Datenblockes ein Übertragungsfehler aufgetreten (CRC nicht korrekt) und es muss derselbe Block noch einmal gesendet werden. Wird ein „NextBlock“ empfangen, so überprüft die PC-Software, ob dies der letzte zu sendende Block ist. Ist dies der Fall so wird der Befehl „DataEnd“ gesendet und auf ein Acknowledgment bzw. auf ein „SameBlock“ des Bootloaders gewartet. Ansonsten wird der nächste Datenblock gesendet und wiederum auf ein Kommando des Bootloaders gewartet.

9.1.2 Verify

Wird der Befehl „Verify“ an den Bootloader gesendet, so vergleicht der Bootloader die im Flash-Speicher gespeicherte Applikation mit dem angegebenen Firmwareimage. Der Vorgang ist identisch zum Update-Vorgang (9.1.1) mit dem einzigen Unterschied, dass der Bootloader das Firmwareimage nicht in den Flash-Speicher schreibt sondern die Daten nur vergleicht. Dementsprechend antwortet der Bootloader mit „Valid“ oder „Invalid“.

9.2 Kommunikationsprotokoll

Hier wird selbiger Kommunikationsstack wie im Bootloader verwendet.

9.3 Userinterface

Vorerst wird nur eine Kommandozeilenversion der PC-Software implementiert.

10 BIN2SEC

Dieses Tool steht nur dem Softwareentwickler zur Verfügung und wird in Java implementiert. Es liest ein unverschlüsseltes Bin-File einer Applikation ein und erstellt ein verschlüsseltes Firmwareimage (AES+SHA-256, siehe Kapitel 7).

11 Implementierung des Bootloaders

11.1 Bootloader

11.1.1 Allgemeines

Für die Implementierung wurden folgende Softwarepakete von Atmel [11] verwendet:

- AVR32 Studio, Version: 2.5.0, Build id: 283
- AVR32 Gnu Toolchain, Version: 2.4.2
- AVR UC3 Software Framework, Version: 1.7.0-AT32UC3

Hardware: Alvidi AVR32-Modul, AL-UC3AEB [18]

11.1.2 Communication Modul

11.1.2.1 APL

In diesem Layer wurde das serielle Protokoll aus Kapitel 6 implementiert.

11.1.2.2 Data Link Layer

In diesem Layer wurde selbiges Protokoll implementiert, welches bereits im Beispiel in Kapitel 8.1 (AVR8) im Data Link Layer erläutert wurde.

11.1.2.3 Physical Layer

Senden der Daten:
Es wird anhand von Kommunikationstimeouts erkannt, wann ein Frame komplett empfangen wurde. Dieser wird dann an den Data Link Layer weitergegeben.
Empfang der Daten:
Es wird der komplette Frame vom Data Link Layer übernommen und über die RS232 Schnittstelle gesendet.

Es gibt keinen Header und keinen Trailer im Physical Layer.

11.1.2.4 Frameformat

2 Bytes	2 Bytes	0-251 Bytes	1 Bytes
Data Link Header	APL Header	Payload	Data Link Trailer

Tabelle 17: Frameformat

11.1.3 Crypto Modul

Sowohl für die AES-Verschlüsselung als auch für die Berechnung der Hash-Signatur wurde die Small Cryptographic Library von PolarSSL [17] verwendet (Version

0.12.1). Diese Bibliothek ist Open Source und wurde speziell für embedded Systems entwickelt (ressourcensparend).

11.2 PC-Software (UpdateSW)

Die Implementierung erfolgte in Java. Es wurden folgende externe Bibliotheken eingebunden:

- java-getopt-1.0.13 [19]: Argumentbehandlung
- RXTX 2.1-7 [20]: Serielle Kommunikation

11.3 BIN2SEC

Die Implementierung erfolgte in Java. Es wurde folgende externe Bibliothek eingebunden:

- java-getopt-1.0.13 [19]: Argumentbehandlung

12 Benutzerhandbuch

12.1 PC-Software (UpdateSW)

Die PC-Software kann über die Kommandozeile mit folgendem Befehl aufgerufen werden: java UpdateSW

Prinzipiell gibt es 5 unterschiedliche Betriebsmodi:

- -u: Es wird ein Update durchgeführt.
- -v: Es wird ein Verify durchgeführt.
- -e: Nach einer Passwortabfrage wird der Flash-Speicher gelöscht.
- -r: Der Controller wird neu gestartet.
- -h: Es wird eine Hilfe ausgegeben.

Alle diese Betriebsmodi benötigen jedoch zusätzliche Argumente. Werden diese Argumente nicht angegeben, so wird eine kurze Hilfe ausgegeben.

- -u:
  - –c: Mit dieser Option wird der COM-Port spezifiziert (z.B. „–c COM5“). Diese Option muss angegeben werden.
  - –b: Mit dieser Option wird das verschlüsselte Firmwareimage spezifiziert, welches beim Update verwendet wird (z.B. „-b example.bin). Diese Option muss angegeben werden.
  - –f: Mit dieser Option kann ein externer Flash ausgewählt werden, falls die Applikation nicht in den Default-Speicher geladen werden soll (z.B. „-f 2“). Diese Option ist optional.

- -v :
  - –c: Mit dieser Option wird der COM-Port spezifiziert (z.B. „–c COM5“). Diese Option muss angegeben werden.
  - –b: Mit dieser Option wird das verschlüsselte Firmwareimage spezifiziert, welches beim Verify verwendet wird (z.B. „-b example.bin). Diese Option muss angegeben werden.
  - –f: Mit dieser Option kann ein externer Flash ausgewählt werden, falls sich die Applikation nicht im Default-Speicher befindet (z.B. „-f 2“). Diese Option ist optional.
- -e:
  - –c: Mit dieser Option wird der COM-Port spezifiziert (z.B. „–c COM5“). Diese Option muss angegeben werden.
  - –f: Mit dieser Option kann ein externer Flash ausgewählt werden, falls sich die Applikation nicht im Default-Speicher befindet (z.B. „-f 2“). Diese Option ist optional.
- -r:
  - –c: Mit dieser Option wird der COM-Port spezifiziert (z.B. „–c COM5“). Diese Option muss angegeben werden.

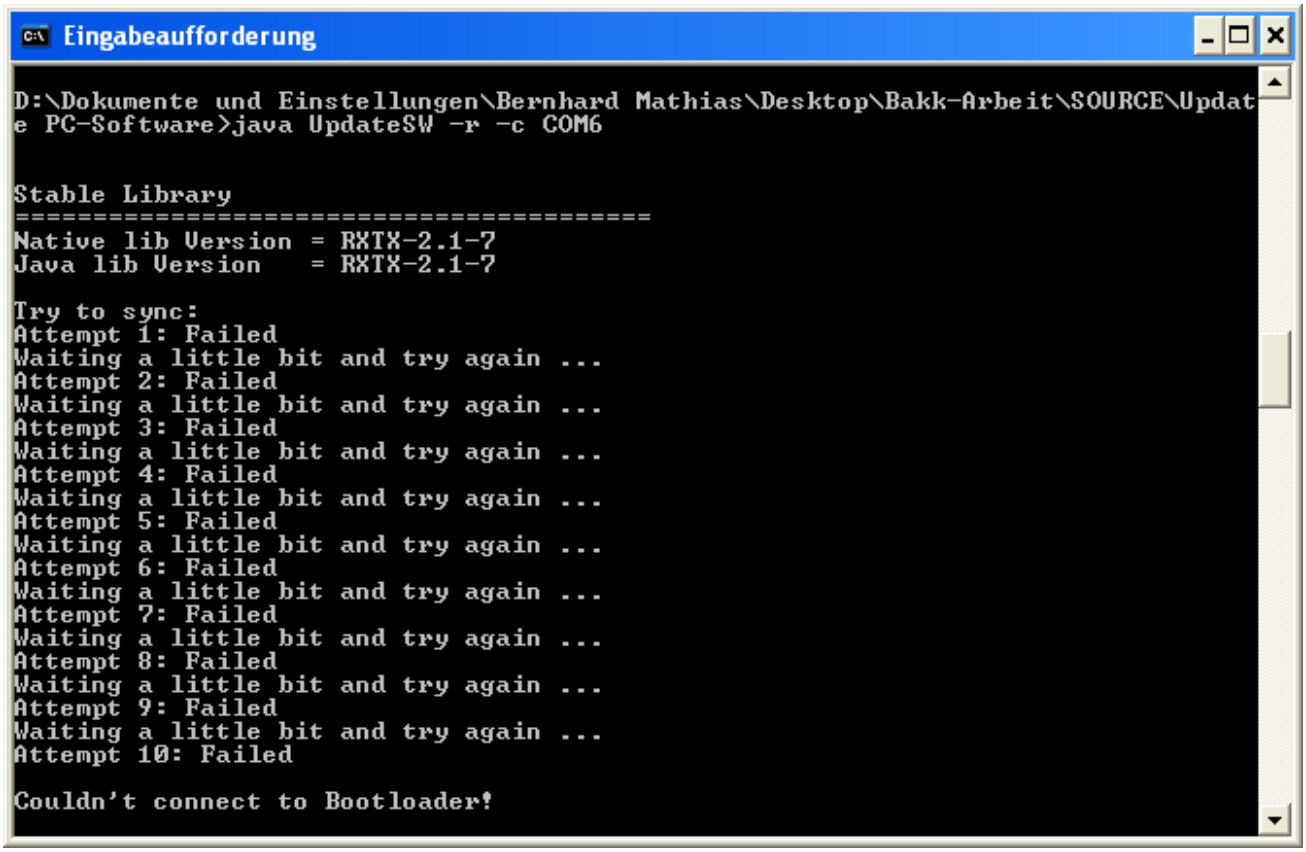

Abbildung 10: Erfolglose Synchronisation mit Bootloader

Die Software versucht zunächst, sich mit dem Bootloader zu verbinden (gilt für alle Betriebsmodi bis auf „-h“). Abbildung 10 zeigt die PC-Software nach einem erfolglosen

Synchronisationsversuch. Um sich mit den Bootloader verbinden zu können, muss dieser auch auf dem Prozessor laufen, d.h. der Prozessor muss eventuell neu gestartet werden (falls bereits eine andere Applikation auf dem Prozessor läuft).

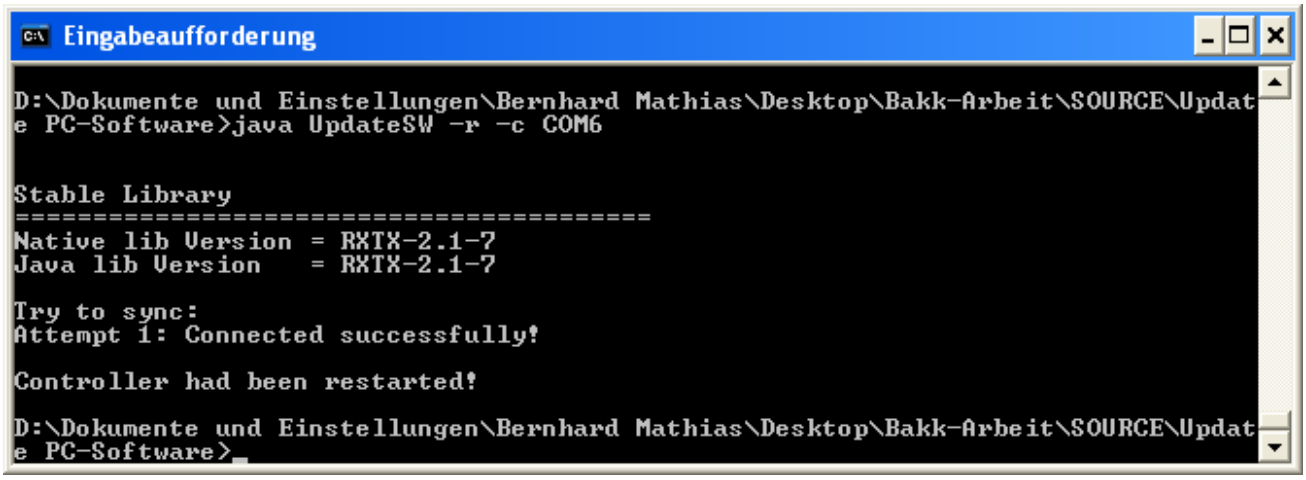

Abbildung 11: Erfolgreiche Synchronisation mit Bootloader

Abbildung 11 zeigt die PC-Software nach einer erfolgreichen Synchronisation mit dem Bootloader. Hier wurde der Betriebsmodus „-r“ gewählt, der Controller wurde somit neu gestartet.

Beispiele für gültige Kommandos:

- Update: java UpdateSW –u –c COM5 –b example.bin
- Verify: java UpdateSW –v –c COM5 –b example.bin
- Restart: java UpdateSW –r –c COM5
- Erase: java UpdateSW –e –c COM5

12.2 BIN2SEC

BIN2SEC kann über die Kommandozeile mit folgendem Befehl aufgerufen werden:

java BIN2SEC

Die Software erwartet sich jedoch mehrere Argumente:

- -s: Spezifiziert das Source Bin-File, welches verschlüsselt werden soll.
- -t: Spezifiziert das Target Bin-File, also den Dateinamen des verschlüsselten Images.
- -f: Spezifiziert die Firmwareversion der Applikation.
- -b: Spezifiziert die Bootloaderversion. Diese muss identisch zur Version des genutzten Bootloaders sein.
- -c: Spezifiziert die CPU-Version. Legt also fest, auf welcher Architektur der Bootloader läuft.

- -a: Spezifiziert den Adress-Offset (dezimal), legt also fest, wo die Firmware letztendlich im Flash-Speicher abgespeichert werden soll. Der Offset muss größer als der Speicherbedarf des Bootloaders gewählt werden.
- -h: Gibt eine kurze Hilfe aus.

Alle diese Argumente müssen angegeben werden (bis auf „-h“)!

Beispiel für gültiges Kommando:

```
java BIN2SEC -s test.bin -t crypted_test.bin -f 1 -b 1 -c 1 -a 128000
```

13 Zusammenfassung

Das Ziel dieser Bakkalaureatsarbeit war es, einen modularen Bootloader mit kryptographischen Funktionen zu spezifizieren und zu designen. Darauf aufbauend sollte der Bootloader auf einen AVR32 UC3A1512 Prozessor implementiert werden.

Nach den einführenden Kapiteln 1 und 2 wird in Kapitel 3 die Architektur des Bootloaders spezifiziert. Es werden folgende 5 Module näher erklärt:

- Boot Modul: Steuert die Koordination zwischen den einzelnen Modulen.
- Control Modul: Beinhaltet prozessorspezifische Methoden.
- Flash Modul: Steuert den Flashzugriff.
- Communication Modul: Ist für die Kommunikation mit der PC-Software verantwortlich.
- Crypto Modul: Beinhaltet kryptographische Funktionen.

Durch den modularen Aufbau können einzelne Module in Abhängigkeit vom verwendeten Prozessortyp abgeändert werden, ohne dass sich die Änderung auf die Funktionsfähigkeit des Bootloaders auswirkt.

Kapitel 4 gibt Aufschluss über die grundlegende Funktionsweise des Bootloaders, beschreibt also, wie sich der Bootloader nach dem Neustart verhält.

Kapitel 5 umfasst eine Analyse gängiger Verschlüsselungs- und Hashfunktionen. Für die Implementierung auf dem AVR32 wurde letztendlich eine AES-Verschlüsselung und eine SHA-256 Hashfunktion gewählt.

Nach einer Spezifikation des verwendeten Kommunikationsprotokolls (Kapitel 6) und einer Spezifikation der unterstützten Firmwareimages (Kapitel 7) wird in Kapitel 8 anhand von Beispielen die mögliche Umsetzung des Bootloaders auf verschiedenen Architekturen erläutert.

Die Kapitel 9 und 10 beschreiben 2 zugehörige PC-Programme, welche für die Verwendung des Bootloaders notwendig sind.

In Kapitel 11 findet man kurze Erläuterungen zur Implementierung des Bootloaders, Kapitel 12 beinhaltet ein kurzes Manual zur Bedienung der zugehörigen PC-Software.

14 Quellenverzeichnis

[1] DENX Software Engineering, „Das U-Boot – the Universal Boot Loader“, “*DENX*”, 2002. [Online]. Available: http://www.denx.de/wiki/U-Boot [Accessed: Oct. 17, 2009]

[2] Vector Group, “Flash Bootloader”, “Vector”, 2006. [Online]. Available: http://www.vector.com/vi_flashbootloader_en.html [Accessed: Oct. 17, 2009]

[3] Shaoziyang's Personal Site, “AVR Universal Bootloader”, Juli 2009. [Online]. Available: http://sites.google.com/site/shaoziyang/Home/avr/avr-universal-bootloadere [Accessed: Oct. 17, 2009]

[4] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd Edition, Canada: John Wiley & Sons, 1996

[5] Regenechsen, “DES, Triple-DES”, 2010. [Online]. Available: http://www.regenechsen.de [Accessed: Jan. 09, 2010]

[6] Ursalab, “Blowfish (cipher)”, 2006. [Online]. Available: http://www.ursalab.com/ [Accessed: Jan. 09, 2010]

[7] Bruce Schneier, “Cryptoanalysis of Twofish (II)”. [Online]. Available: http://www.schneier.com [Accessed: Jan. 09, 2010]

[8] Cryptolib, “Ciphers - RTEA”. [ONLINE]. Available: http://cryptolib.com [Accessed: Jan. 09, 2010]

[9] Security and the Net, “First conference in SHA-3 competition start next week”, 2009.
[Online]. Available: http://securityandthe.net [Accessed: Jan. 09, 2010]

[10] ADVERTISING KINGS MEDIA GROUP, “MD5-Hashed Suppression Lists and The Affiliate Marketer”, 2009. [ONLINE]. Available: http://www.akmg.com [Accessed: Jan. 09, 2010]

[11] Atmel Corporation, “AVR32 32-bit MCU”, 1998. [Online]. Available: http://atmel.com [Accessed: Oct. 17, 2009]

[12] ARM, “ARM-Cortex M3”, 2007. [Online]. Available: http://arm.com [Accessed: Oct. 17, 2009]

[13] Jayavardhan R. Kandi, „Embedded Cryptography:
An Analysis and Evaluation of Performance and Code Optimization Techniques for Encryption and Decryption in Embedded Systems“, 2003. [Online]. Available: http://etd.fcla.edu/SF/SFE0000151/THESIS-Jay16.pdf [Accessed: Oct. 21, 2009]

[14] Andreas Schwarz, Forumsbereich. [Online]. Available: www.Mikrocontroller.net [Accessed: Apr. 08, 2010]

[15] Forumsbereich. [Online]. Available: http://www.avrfreaks.net [Accessed: Apr. 08, 2010]

[16] NIST, “Secure Hash Standard”, National Institute of Standards and Technology, 2004. [Online]. Available:
http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf [Accessed: Nov.19 2009]

[17] PolarSSL, “Small Cryptographic Library”, Brainspark, 2008. [Online]. Available: http://www.polarssl.org/ [Accessed: Apr.01 2010]

[18] Alvidi, 2009. [Online]. Available: http://alvidi.de/ [Accessed: Apr.06 2010]
[19] Urbanophile, "Passionate about cities", Java getopt, 2008. [Online]. Available: http://www.urbanophile.com/arenn/hacking/getopt/ [Accessed: Apr.06 2010]
[20] RXTX, 2006. [Online]. Available: http://rxtx.qbang.org/wiki/index.php/Main_Page [Accessed: Apr.06 2010]

15 Abbildungsverzeichnis

Abbildung 1: Modularer Bootloader 10
Abbildung 2: Flussdiagramm Bootloader 22
Abbildung 3: Data Encryption Standard [5] 27
Abbildung 4: Blowfish [6] 30
Abbildung 5: Twofish [7] 31
Abbildung 6: XXTEA [8] 32
Abbildung 7: SHA-2 [9] 35
Abbildung 8: MD5 [10] 36
Abbildung 9: Ablauf des Updates im PC-Programm 45
Abbildung 10: Erfolglose Synchronisation mit Bootloader 49
Abbildung 11: Erfolgreiche Synchronisation mit Bootloader 50

16 Tabellenverzeichnis

Tabelle 1: Aufbau Communication Modul 15
Tabelle 2: Datenpuffer im Communication Modul 17
Tabelle 3: AES-Standards 28
Tabelle 4: Algorithmen – Ein Überblick 32
Tabelle 5: SHA-Standards 34
Tabelle 6: Frame Format 37
Tabelle 7: APL Frame Typ 1 37
Tabelle 8: APL Frame Typ 2 37
Tabelle 9: Funktionscodes 38
Tabelle 10: Firmware Image 39
Tabelle 11: Aufbau Communication Modul Bsp. 1 41
Tabelle 12: Data Link Layer - Protokoll 41
Tabelle 13: Aufbau Communication Modul Bsp. 2 42
Tabelle 14: Beispiel Ack-Frame 42
Tabelle 15: Beispiel Update-Frame 42
Tabelle 16: Ethernet Frame Version 2.0 43
Tabelle 17: Frameformat 47