universitätbonn

Realisierung externer kryptographischer Funktionen für Arduino

Bachelorarbeit

von

René Julian Neff

Institut für Informatik IV
Arbeitsgruppe IT-Sicherheit
Prof. Dr. Michael Meier
Rheinische Friedrich-Wilhelms-Universiät Bonn

Oktober 2014

Erstgutachter: Prof. Dr. Michael Meier
Zweitgutachter: Dr. Matthias Frank
Betreuer: Dipl. Inf. Matthias Wübbeling

Selbstständigkeitserklärung

Hiermit erkläre ich, die vorliegende Bachelorarbeit selbstständig nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Bonn, 08.10.2014

_____

René Julian Neff

Zusammenfassung

In dieser Bachelorarbeit wird der Entwurf und die Implementierung von zwei auf der Arduino Mikrocontrollerplattform basierenden Systemen zur externen Berechnung von kryptographischen Funktionen beschrieben.
Ein in dieser Arbeit realisiertes Koprozessorsystem, das ebenfalls mit einem weiteren Arduino Mikrocontroller arbeitet, kann zur Authentifizierung von Nachrichten genutzt werden, die über unsichere Kommunikationskanäle ausgetauscht werden. Ein alternatives System nutzt die von einem Atmel AT-SHA204 Mikrochip angebotenen Funktionalitäten zur Berechnung von MACs.
Die notwendigen Grundlagen zum Verständnis beider Systeme sowie ihres dargelegten Entwurfs, ihrer realisierter Implementation und ihrem genutzten Aufbau werden einleitend vorgestellt. Zudem werden Probleme während des Entwicklungsprozesses erläutert.

Inhaltsverzeichnis

1 Einleitung 1
- 1.1 Zielsetzung . . . 2
- 1.2 Related Work . . . 2

2 Grundlagen 4
- 2.1 Authentifizierter Nachrichtenaustausch . . . 4
- 2.2 Arduino . . . 6
- 2.3 Atmel AT-SHA204 . . . 7
- 2.4 Mikrocontroller Kommunikation . . . 9
- 2.5 Systemarchitektur Arduino . . . 12

3 Entwurf 14
- 3.1 Arduino als Koprozessor . . . 14
  - 3.1.1 Systementwurf . . . 14
  - 3.1.2 Hardwareaufbau . . . 16
- 3.2 Atmel AT-SHA204 . . . 18
  - 3.2.1 Systementwurf . . . 18
  - 3.2.2 Hardwareaufbau . . . 19

4 Umsetzung 21
- 4.1 Arduino als Koprozessor . . . 21
  - 4.1.1 Softwareentwurf . . . 21
  - 4.1.2 Verwendete Bibliotheken . . . 25
  - 4.1.3 Implementierung . . . 26
  - 4.1.4 Besonderheiten und Probleme . . . 28
- 4.2 Arduino und Atmel AT-SHA204 . . . 29
  - 4.2.1 Softwareentwurf . . . 30
  - 4.2.2 Verwendete Bibliotheken . . . 37
  - 4.2.3 Implementierung . . . 37
  - 4.2.4 Besonderheiten und Probleme . . . 39

5 Evaluation 41
- 5.1 Ardunio als Koprozessor . . . 41
- 5.2 Arduino und Atmel AT-SHA204 . . . 43

6 Zusammenfassung 44

A Literatur 46

1 Einleitung

In dieser Bachelorarbeit werden zwei unterschiedliche Ansätze zur externen Berechnung von kryptographischen Funktionen für ein auf einem Arduino Mega2560 basierenden System entworfen und umgesetzt.

Im Projekt *Arduino Remote Infrared Network Adapter*, kurz ARINA, entstand eine Lösung zur Bedienung von infrarotgesteuerten Geräten ohne direkten Sichtkontakt [43]. Entworfen und realisiert wurden zwei kompakte ethernetbasierte Adapter, welche die Steuerung von einem oder mehreren entfernten, stationären und insbesondere per Infrarot ansteuerbaren Multimediageräten ermöglichen. Die in diesem Projekt bisher umgesetzte Kommunikation zwischen den beiden Adaptern erfolgt ohne Authentifizierung oder Verschlüsselung. Ausgetauschte Daten können abgefangen und Anweisungen manipuliert werden. Ziel dieser Bachelorarbeit ist es diesem und eventuellen zukünftigen Projekten, die eine solche potenziell abgreifbare Kommunikation nutzen, eine Möglichkeiten zu schaffen, anerkannte Methoden der Authentifizierung ihrer Kommunikation bereitzustellen.

Aus eigener Erfahrung tendieren Entwickler dazu sich fokussiert mit ihrem funktionalen Ziel auseinander zu setzen und lassen wichtige Aspekte, wie unter anderem eine gegen Manipulation gesicherte Übertragung, außen vor. Somit ist es wichtig, einen simplen und intuitiven Zugang zur Nutzbarkeit von Technologien zu entwickeln, welcher es auch Einsteigern erlaubt, eine solche Kommunikation in ihren Projekten einzusetzen.

Ein wichtiger Aspekt von ARINA ist es, die empfangenen Signale möglichst in Echtzeit zu kapseln, anschließend zu übertragen und wiederzugeben. Auch in anderen Projekten, bei denen Daten über potenziell ungesicherte Wege ausgetauscht werden, kann dies eine bedeutende Rolle spielen. Deshalb werden an etwaige zwischengeschaltete Abläufe, wie das Berechnen einer Hashsumme vor Beginn der Übertragung sowie die entsprechende Prüfen einer solchen nach Empfang der Datenpakete, Anforderungen in Form einer hohen Datendurchsatzrate gestellt.

Im Bereich der Mikrocontrollerprogrammierung wird meist mit beschränkten Ressourcen in Bezug auf verfügbare Rechenleistung, Arbeitsspeicher und nicht flüchtigem Speicher gearbeitet. Daraus folgt eine Ergänzung des Anforderungsprofils insofern, als dass möglichst ressourcenschonend die notwendigen Berechnungen durchgeführt werden sollen.

In der hier vorgestellten Arbeit wurde ein System beschrieben und realisiert, das mittels freiverkäuflichen *specialpurpose* Chips arbeitet, welche die Berechnungen der

kryptographischen Funktionen übernehmen können. Zusätzlich wurde eine zweites alternatives System entworfen und umgesetzt, dass die kryptographischen Berechnungen auf einen zusätzlichen dediziert für diese Berechnungen zuständigen Arduino auslagert; dieser dient als eigenständiger Koprozessor. Ein zur Kommunikation mit dem Koprozessor notwendiger Ablauf wurde erarbeitet und wird im Detail erläutert.

1.1 Zielsetzung

Zielsetzung dieser Bachelorarbeit ist es, zwei verschiedene Möglichkeiten zur authentifizierten Kommunikation auf der Arduino Plattform zu schaffen. Die hierzu notwendigen Berechnungen sollen ausgelagert stattfinden, so dass zeitkritische Prozesse in einem schon etablierten Entwurf durch die hier entworfene Erweiterung möglichst wenig beeinträchtigt werden. Die Auslagerung erfolgt auf speziell dafür vorgesehener Hardware.

Eine der umgesetzten Varianten soll hierzu auf einen zweiten Arduino Mikrocontroller zurückgreifen, wohingegen die andere Variante auf einem speziellen Mikrochip basiert. Eine Ausarbeitung der verschiedenen Kommunikationswege zwischen zwei Mikrocontrollern sowie Mikrocontroller und Mikrochip, als auch eine genauere Beschreibung des zur Authentifizierung gewählten Verfahrens, werden als Grundlage der anschließend durchgeführten Implementierung dienen. Die zu beachtenden Besonderheiten, welche sich aus der zu verwendenden Arduino Mikrocontroller Plattform ergeben, werden ebenfalls vorgestellt.

1.2 Related Work

Schon 2012 wurde das Projekt *smARtDUINO* vorgestellt, das durch den *smARtBUS* ein offenes aber komplett kompatibles Protokoll zur Kommunikation zwischen verschiedenen Arduino Plattform Komponenten bieten soll. Mittels sogenannter *smARtBUS EXTENSION* Boards wurde ein Aufbau beschrieben, der es ermöglicht, vier Arduino Mikrocontroller in einem gemeinsamen Aufbau zu verwenden. Wird ein SD-Kartenreader mit in den Aufbau eingebunden, sollten alle Mikrocontroller Zugriff auf diese dann gemeinsame Ressource haben.

Nähere Erklärungen zu verwendeten Protokollen sind nicht mehr einsehbar, da die Homepage des Anbieters unerreichbar ist. Das Projekt, welches per gemeinschaftlicher Finanzierung (eng. *Crowdfunding*) realisiert werden sollte, konnte fast 1000 Unterstützer finden und eine Gesamtsumme von annähernd $160.000 US Dollar, was mehr als dem Sechsfachen des ursprünglich angestrebten Betrags entspricht, einsammeln [32]. Leider scheinen mehrere Unterstützer jedoch keine oder nur fehlerhafte Endprodukte geliefert bekommen zu haben. Ein nachträgliches Bestellen ist ebenfalls nicht mehr möglich, da der Entwickler und Vertreiber Webseiten und E-Mailadressen abgeschaltet hat [33, 47].

Ein weiteres Projekt, das zum Ziel hatte in einem verbundenen Aufbau mehrere Arduino Mikrocontroller zur gemeinsamen Berechnung zu verwenden sollte ebenfalls per *Crowdfunding* Kampagne vorfinanziert werden.

Tobias Strauch stellte im April 2014 den Prototypen eines von ihm entworfenen Multicore-Arduino-Boards unter dem Namen *Arduissimo* vor [30]. Strauch hatte ein *Field Programmable Gate Array* (kurz FPGA) Board genutzt, um 16 Arduino Mikroprozessoren äquivalent nachzustellen. Es sollte möglich sein, jedem dieser Mikroprozessoren eine dedizierte Aufgabe wie beispielsweise das Verarbeiten von I^2C oder SPI Kommunikation zuzuweisen (vlg. Kapitel 2.4). Ebenso wurde beschrieben, dass sich die jeweiligen Mikroprozessoren gegenseitig Interrupts senden können.

Im Gegensatz zu der in der vorliegenden Bachelorarbeit angestrebten Lösung wird bei *Arduissimo* ein gemeinsamer Daten- und Programmspeicher genutzt [30]. Strauchs Entwurf sollte in geringer Stückzahl von 400 Einheiten produziert werden. Die Kampagne endete jedoch am 29. Juni 2014 ohne das genannte Finanzierungsziel zu erreichen.

2 Grundlagen

In diesem Kapitel werden die Grundlagen dargelegt, auf denen die spätere Ausarbeitung aufbaut. Zunächst wird eine Beschreibung von authentifizierter Kommunikation den Aufbau von MAC und HMAC erläutern. Anschließend wird die Mikrocontroller Plattform Arduino vorgestellt und der für diese Arbeit relevante Arduino Mega2560 näher beschrieben.

Aufbauend auf diesen Informationen werden die wesentlichen und für diese Arbeit relevanten Details der Arduionosystemarchitektur geschildert, nachdem die Kenndaten des ebenfalls in dieser Arbeit genutzten Atmel AT-SHA204 Chips aufgeführt und technische Grundlagen dazu erläutert werden. Dieses Kapitel fasst somit in unterschiedlicher Granularität die für die Arbeit vorausgesetzten Grundlagen zusammen und hilft beim Verständnis des Entwurfs und der Umsetzung.

2.1 Authentifizierter Nachrichtenaustausch

Dieser Abschnitt befasst sich mit der Authentifikation von Nachrichten, genauer damit, wie diese einem Sender sicher zugeordnet werden können. Zu diesem Zweck werden zunächst die dazu benötigten Grundlagen im Einzelnen erläutert sowie anschließend die für diese Arbeit relevanten Verfahren dargelegt.

Bei Kommunikation über unsichere Kanäle oder anderweitig nicht abgesicherte Medien kann ein Empfänger eine Manipulation der empfangenen Daten oder ein Vortäuschen des eigentlichen Senders nicht ausschließen. Dem Empfänger obliegt es somit die Authentizität einer empfangenen Nachricht zu prüfen. Man spricht in diesem Zusammenhang davon den Ursprung, genauer die Identität, einer Nachricht sicherzustellen und manipulierte Nachrichten beziehungsweise deren Integrität zu erkennen [38].

Verfahren, die zur Durchführung solch einer Authentifizierung sichere Schlüssel nutzen, nennt man *Message Authentication Codes* (MACs). Bei diesen Verfahren wird jede versandte Nachricht durch ein entsprechendes Stück generierten Code begleitet, welcher durch das gewählte Verfahren berechnet wurde. Als Berechnungsgrundlage dient hierbei sowohl die Nachricht als auch der geteilte sichere Schlüssel. Es handelt sich somit um ein symmetrisches Verfahren, weshalb Sender und Empfänger das gleiche geheime Schlüsselmaterial teilen. Der Empfänger wiederholt zum Überprüfen einer Nachricht die gleichen Schritte, die auch der Sender zum Erstellen des MAC durchführte. Konnte mit der empfangenen Nachricht der gleiche MAC erzeugt werden, welcher diese

begleitete, kann die Nachricht als unverändert und eindeutig dem Sender zuordbar angesehen werden. Das *Handbook of Applied Cryptography* [39] definiert einen MAC Algorithmus wie folgt:

Definition eines MAC nach Menezes

Ein *Message Authentication Code (MAC)* Algorithmus ist eine Funktionsfamilie h_k über dem Schlüssel k für die gilt:

- Einfache Berechenbarkeit bei bekannten Eingaben, daher von Nachricht und Schlüssel. Das Ergebnis einer solchen Berechnung wird *MAC-Wert* oder *MAC* genannt.
- Komprimierung von beliebig langer endlicher Eingabe auf eine konstante Länge, *Blocklänge* genannt.
- Schwere Errechenbarkeit von gültigen MAC-Werten für eine neue Nachricht y, wenn keine oder beliebig viele (Nachricht x, MAC(Nachricht x))-Wertepaare bekannt sind. Wobei gilt $x \neq y$.

Nach dieser allgemeinen Definition eines MAC wird nun eine spezielle Konstruktion des allgemeinen MAC Verfahrens genauer betrachtet: Der Aufbau des *Keyed-Hashed Message Authentication Code* (HMAC). Dieses Verfahren bietet den Vorteil von austauschbaren kryptographischen Hashverfahren zur Berechnung eines MAC-Wertes [39]. Die hierbei verwendeten Hashverfahren erfüllen weitaus strengere Anforderungen, als es die Definition eines MAC erfordert. Der Aufbau einer HMAC wird von der *Internet Engineering Task Force* (kurz IETF) im RFC2104 sowie vom armerikanischen Wirtschaftsministerium als ein Standard bei der Informationsverarbeitung definiert (FIPS-198) [35] [62].

Die grundsätzliche Idee von Nachricht, Schlüssel und in diesem Falle Hashfunktion wird auch bei HMAC beibehalten und eine Berechnung erfolgt nach einem fest definierten Schema. Zum Berechnen einer HMAC wählt man zunächst eine zugrundeliegende Hashfunktion H.

Im ersten Schritt wird der Schlüssel durch Anhängen von Nullen auf die charakteristische Blocklänge der jeweiligen Hashfunktion erweitert. Alternativ, falls der gewählte Schlüssel länger als die Blocklänge ist, wird fortan mit dem Hashwert über den gewählten Schlüssel weiter gerechnet. Im Folgenden wird der dann erzeugte Schlüssel jeweils mit einer von zwei Konstanten mit einer bitweisen XOR-Operation verknüpft. Diese Konstanten sind schon durch den Standard in RFC2104 vorgegeben und müssen nicht frei gewählt werden. Eine Konkatination aus den beiden so erzeugten Werten bildet im dritten und letzten Schritt, verbunden mit der eigentlichen Nachricht, die gewählte Hashfunktion zur HMAC.[35]

Berechnung eines HMAC nach RFC2104

$$\mathrm{HMAC}_k(M) = H((K \oplus opad \parallel H((K \oplus ipad) \parallel M))$$

mit

$$opad = 0\text{x}5C...0\text{x}5C \text{ und } ipad = 0\text{x}36...0\text{x}36$$

als feste Konstanten der jeweiligen Blocklänge,
$\oplus$ als bitweise XOR-Operation und $\parallel$ als einfache Konkatenation.[35]

Es konnte gezeigt werden, dass es einen engen Zusammenhang mit der Eigenschaft der Pseudozufälligkeit der zugrundeliegenden Hashfunktion und der mit ihr gebildeten HMACs gibt [17]. Daher ist es einem Angreifer nicht möglich mit beliebiger Eingabenachricht Aussagen über die erzeugte Ausgabe-HMAC zu machen und umgekehrt, mit der Ausnahme, dass dieselbe Eingabe immer dieselbe Ausgabe erzeugt. Anwendung finden HMACs auch in den Protokollen *Transport Layer Security* (TLS) und *Secure Shell* (SSH) [22, 66].

Neben den vorgestellten theoretischen Vorteilen von HMACs gibt es auch praktische Vorzüge. So sind standardisierte Hashverfahren als fertige in Hardware implementierte Bausteine zu erhalten. Diese ermöglichen es auch, Kleinsysteme, wie zum Beispiel ARINA oder andere so genannte eingebettete Systeme (engl. *embedded systems*), mit solch einer Funktionalität auszustatten.

2.2 Arduino

Die Arduino-Plattform ist eine sich aus Soft- und Hardwarekomponenten zusammensetzende Physical-Computing-Plattform. Der Idee von Open Source folgend sind die einzelnen Komponenten quelloffen verfügbar. Ein definiertes Boardlayout mit festen Positionen besonderer Pins und Verbindungsstellen ermöglicht die Realisierung von einfach anzuschließenden Erweiterungsboards, sogenannter *Shields*. Dies und der geringe Preis unterstützt computer- und hardwareaffine Hobbybastler in eigenständigen Erweiterungen, sowohl mittels zusätzlicher Hardwarekomponenten als auch durch den Entwurf von eigenen Softwarebibliotheken [10].

Die Softwareentwicklung kann in der ebenfalls plattformunabhängigen und frei verfügbaren Arduino IDE erfolgen; die verwendete Programmiersprache ist ein vereinfachtes C++ und sein Vorgänger C. Die ursprüngliche Hardwarebasiskomponente des Arduino (Uno) ist ein Mikrocontrollerboard, das mit einem 16 MHz getakteten Atmel AVR ATmega328 und 32 KiB Flashspeicher bestückt ist. [10]

Eine Reihe digitaler Ein- und Ausgänge, welche unter anderem pulsweitenmodulierte Signale unterstützen, sowie analoge Eingänge stehen dem Entwickler zur Verfügung. Das Bespielen - auch *flashen* genannt - mit dem entworfenen Programmcode, erfolgt mittels eines USB-Anschlusses, der auch für die nötige Spannungsversorgung genutzt

wird. Alternativ kann ein *In Circuit Serial Programming* (kurz: ICSP) Interface genutzt werden, welches bei der Mikrocontrollerprogrammierung sehr verbreitet ist [40]. Die Spannungsversorgung kann zudem auch über eigens dafür vorgesehene Pins erfolgen [50]. Die Leistungsaufnahme eines Arduino Uno ohne weitere angeschlossene Verbraucher ist bei 7V mit 260mW gering und so können auch mobile Entwürfe ohne eine kabelgebundene Stromversorgung realisiert werden [44].

Der ursprüngliche Entwurf wurde bei den nachfolgend veröffentlichten Arduinos beibehalten. Diese unterscheiden sich hauptsächlich durch unterschiedliche Baugrößen und einzelnen Ausstattungsmerkmalen, meist jedoch nur in Bezug auf Flashspeicher und die Anzahl der zur verfügungstehenden Ein- und Ausgänge. Der Arduino Due stellt eine Ausnahme dar, da er keinen der sonst verwendeten Atmel ATmegaAVR Mikrocontroller verwendet, sondern mit einem Atmel ARM Cortex-4 ausgestattet ist [50].

Shields können die grundlegenden Anschlussmöglichkeiten eines Arduinos erweitern. So existiert beispielsweise ein Ethernet-Shield, das den angeschlossenen Arduino in ein Ethernet Netzwerk einbinden kann. Der in dieser Arbeit verwendete Arduino Mega2560 ist mit einem Atmel ATmega2560 ausgestattet, der mit 16 MHz getaktet ist; für den Programmcode stehen dem Entwickler 256 KiB Flashspeicher zur Verfügung [9]. Er verfügt über 54 digitale Ein- und Ausgabeanschlüsse (14 PWM-tauglich) und 16 analoge sowie vier separate *Universal Asynchronous Receiver Transmitter* (kurz: UART) Anschlüsse, wobei letztere insbesondere zur Kommunikation mit einem Computer, aber auch zur Kommunikation mit einem weiteren Mikrocontroller genutzt werden können. [4]

Der Arduino unterstützt verschiedene serielle Datenbussysteme zum Austausch von Daten zwischen extern angeschlossenen Schaltbausteinen. Mittels der digitalen Anschlüsse wird das von Motorola entwickelte *Serial Peripheral Interface* (kurz: SPI) unterstützt [20].

Zudem kann auch der ehemals von Philips Semiconductors (jetzt NXP Semiconductors) entwickelte *Inter-Integrated Circuit* (kurz: I²C) Datenbus genutzt werden [45], der jedoch von Atmel *Two-Wire-Interface* (kurz: TWI) genannt wird [48].

2.3 Atmel AT-SHA204

Der Atmel AT-SHA204 ist ein Mikrochipbaustein, welcher verschiedene kryptographische Funktionen unterstützt. Als typische Anwendungsszenarien werden auswechselbare Elemente eines meist für den Endanwender entworfenen Systems beschrieben, die mit Hilfe von verbauten Atmel AT-SHA204 Mikrochipbausteinen auf ihre Authentizität geprüft werden können.

Die Dokumentation des Bausteins beschreibt beispielsweise Druckerpatronen und Batterieeinheiten für Mobiltelefone und Unterhaltungselektronik als typische Anwendungsbereiche [13]. Er verfügt auch über die Möglichkeit, seine kryptographischen Berechnungen einem als *Key Rolling Scheme* bezeichneten Verfahren entsprechend durchzuführen. Dieses Verfahren erlaubt, basierend auf einem fixen geteilten Start-

wert, dass ein Sender einen sich bei jeder Übertragung ändernden Schlüssel an einen Empfänger überträgt. Der Empfänger kann durch das zuvor geteilte Wissen diesen Schlüssel auf Gültigkeit prüfen. Solch ein System findet in Zugangskontrollsystemen wie beispielsweise Garagentorsteuerung Anwendung. [13]

Der Mikrochip kann mit einer Spannung von 2-5 V betrieben werden; seine Leistungsaufnahme übersteigt 3 mA nicht. Mittels eines Ruhezustands kann er in einen Bereitschaftsmodus versetzt werden, der die Leistungsaufnahme auf bis zu 2 µA senkt.[13]

Der Baustein hat, entsprechend der beworbenen Anwendungsszenarien, sehr geringe Abmessungen und kann als fertig verlötetes Produkt, auch in Einzelabgabe, unter der Bezeichnung *AT-SHA204 Authentication Chip Breakout* erworben werden [57]. Die Abbildung 1 verdeutlicht die sehr kleinen Abmessungen, in dem die Trägerplatine auf einer viertel US Dollar Münze, mit einem Durchmesser von circa 25 mm, plaziert wurde [65].

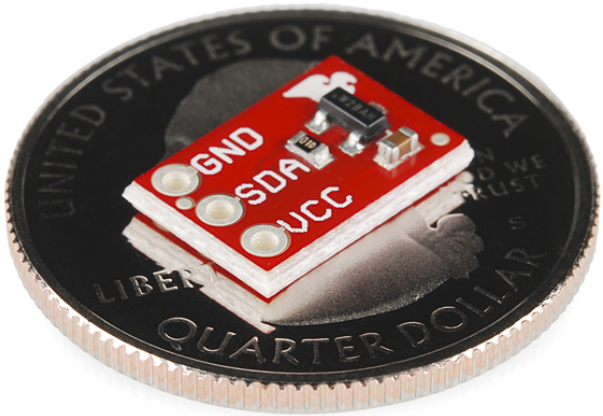

Abbildung 1: *Abbildung der als „ATSHA204 Authentication Chip Breakout“ erwerbbaren Trägerplatine auf einer viertel US Dollar Münze. Abbildung unter „CC BY 3.0 DE“ Lizenz, entnommen von [57]*

Der in dieser Ausführung verbaute Mikrochip ist in einer als *SOT23* bezeichneten Bauform gehalten und verfügt über nur drei Anschlusspins [13]. Die auf einer Trägerplatine vorverarbeitete bestellbare Ausführung wird über das Atmel-spezifische *Single-Wire Interface* (kurz SWI) angesprochen, das angebunden an einen entsprechenden digitalen Anschlusspin des Arduino genutzt werden kann. Das SWI wird in der Dokumentation als eine zu UART kompatible Art der Kommunikation beschrieben [13]. Weitere Ausführungen mit unterschiedlichen Verbau- und Anschlussmöglichkeiten sind ebenfalls verfügbar; neben dem SWI kann in bestimmten Ausführungen auch mittels *Two-Wire Interface* (TWI) mit dem Mikrochip kommuniziert werden [13]. Alternative Ausführungen werden jedoch in einer nicht vorverarbeiteten Form verkauft und müssen auf eine Trägerplatinen oder entsprechenden Steckplätz aufgebracht werden.

Der Baustein unterstützt sowohl in der Version des Single-Wire Interfaces als auch in der des alternativ wählbaren Two-Wire Interfaces (siehe Kapitel 2.4) die Möglichkeit, mehrere Bausteine seiner Art in einem Entwurf einzubinden und entsprechend

anzusteuern. Während das TWI zwei Datenpins zur Kommunikation benötigt, unterstützt das SWI diese Funktionalität bereits mit lediglich einem als Datenbus dienenden Anschluss.

Der AT-SHA204 ist mit einer eindeutigen 72 Bit langen Seriennummer versehen und verfügt über einen 4.500 Bit großen *Nicht Flüchtigen Elektronisch Löschbaren Programmierbaren Nur-Lese-Speicher* (kurz EEPROM), der mit Anwendungsdaten gefüllt werden kann.

Der Chip bietet die Funktion einzelne Speicherbereiche vor unerwünschtem Zugriff oder Veränderung zu sperren. Dies ermöglicht es bis zu sechzehn 256 Bit lange (kryptographische) Schlüssel sicher auf dem Baustein zu speichern. Das entsprechende Setzen der Zugriffsrechte erfolgt in einer Konfigurationszone, die Bestandteil eines *Nicht Flüchtigen Nur Einmal Beschreibbaren Speichers* (kurz PROM) ist. Zusätzlich zum fest vorgegebenen Konfigurationsbereich (*ConfigZone*) stehen dem Entwickler weitere 512 Bits frei zur Verfügung.

Eine weitere Funktionalität des AT-SHA204 besteht darin, 256 Bit lange zufällige Zahlen zu erzeugen. Diese Zahlen sind den Herstellerangaben zufolge unabhängig von vorangegangenen Operationen. [13]

2.4 Mikrocontroller Kommunikation

Bei der Kommunikation zwischen Mikrocontrollern als sowie Mikrocontrollern und Mikrochips gibt es verschiedene Ansätze und Schnittstellen. In dieser Arbeit soll die Mikrocontrollerplattform Arduino als Entwicklungsgrundlage verwendet werden, es wird sich daher im Folgenden Abschnitt auf die von dieser unterstützten Kommunikationsvarianten konzentriert.

Um die Kommunikation zwischen zwei oder mehreren Arduinos zu realisieren, gibt es verschiedene Möglichkeiten. Eine davon ist es, Daten mittels der *Universal Asynchronous Receiver Transmitter*-Schnittstelle (kurz UART) zu übertragen. Diese ist eng verwandt mit der Schnittstellenspezifikation der in den 1990-Jahren populären RS-232 seriellen Schnittstelle, welche häufig zur Kommunikation mit externen Modems oder anderen Computer-Peripheriegeräten verwendet wurde. Mittels des RS-232 Standard wurde unter anderem definiert, welches Anschlussdesign mit welcher Pinbelegung zu nutzen ist, sowie welche Spannung auf den entsprechenden Leitungen anzuliegen hat. [23]

Genutzt wurde ein UART Baustein, der die übertragenen seriellen Signale in Datenworte umsetzt. Diese serielle Schnittstelle ermöglicht es, Daten von bis zu 9 Bit Länge zu verschicken und wird zur Kommunikation zwischen Computer und Arduino genutzt; hierzu werden die digitalen Anschlüsse PIN 0 und 1, bezeichnet mit *Serial*, verwendet [4]. Der in dieser Arbeit genutzte Arduiono Mega2560 bietet zudem noch drei zusätzliche serielle Anschlüsse an PIN 18 und 19 (*Serial1*), PIN 17 und 16 (*Serial2*) sowie PIN 14 und 15 (*Serial3*) [4].

Das heute übliche Verfahren der Kommunikation zwischen Computer und Arduino nutzt die USB-Schnittstelle des Arduino und hat die vorherige Variante nahezu voll-

ständig abgelöst, da sie deutlich nutzerfreundlicher ist. Für die Kommunikation per USB-Schnittstelle wird ein USB-to-TTL Chip verwendet, der eine serielle Schnittstelle zu dem vorher genannten Anschluss (*Serial*) für den angeschlossenen Computer bereitstellt [7]. Zur Kommunikation mit dem Arduino kann eine maximale Datenrate von bis zu 115.200 Bits pro Sekunde verwendet werden [5].

Die maximal mögliche Länge der Verbindungskabel ist stark abhängig von der gewählten Geschwindigkeit der zu führenden Übertragung. Bei praktischen Tests von Texas Instruments konnte gezeigt werden, dass bei nur einem Sechstel der maximalen Übertragungsgeschwindigkeit eines Arduinos, daher 19.200 Bits pro Sekunde, die maximal mögliche Leitungslänge einer RS-232 Verbindung bei knapp 15 Metern liegt. Reduziert man die Übertragungsgeschwindigkeit weiter, erhöht sich die maximal mögliche Leitungslänge exponentiell [52][2]. Zur Kommunikation zwischen mehreren Mikrocontrollern bietet es sich weniger an UART als Schnittstelle zu nutzen, da die Datenübermittlung ausschließlich von einem Gerät zum anderen begrenzt ist und so Verbindungen zwischen mehreren Geräten nicht ohne Weiteres realisiert werden können.

Eine Alternative zum UART stellt das I²C Interface dar. Die Abkürzung I²C (gesprochen *I-squared-C*) steht für *Inter-Integrated Circuit* und ist eine standardisierte, einfach zu nutzende Alternative zum UART. I²C wurde von Philips Semiconductors (jetzt NXP Semiconductors) 1979 entwickelt und wird aus Lizenzgründen mitunter auch als Two-Wire Interface bezeichnet [21]. I²C ist nach Angaben von NXP Grundlage für eine Reihe von Kontrollsystemen gewesen, so zum Beispiel der *System Management Bus* (kurz SM-Bus), welcher unter anderem in Notebooks zum Abfragen des aktuellen Ladezustands genutzt wird [56]. Beworben wird I²C als *de facto world standard*, der mehr als 1000 integrierte Schaltkreise unterstützt und von über 50 Unternehmen in unterschiedlichen Produkten genutzt wird [64].

Durch Weiterentwicklung des ursprünglichen I²C Protokolls konnte neben dem *standard mode*, welcher bis zu 100 kbit/s, und dem *fast mode*, der bis zu 400 kbit/s übertragen kann, der *high-speed mode* definiert, bei dem eine Übertragungsrate von bis zu 3,4 Mbit/s möglich ist [64].

Es können über den digitalen PIN 4 und 5 des Arduino (respektive PIN 20 und 21 in der hier genutzten Arduino Mega2560 Variante) nahezu beliebig viele Geräte miteinander verbunden werden, da sich alle angeschlossenen Arduinos über den gleichen Datenbus austauschen können[29]. Dies ist möglich, da die Geräte durch eine eindeutige Kennnummer adressierbar gemacht werden. Ein Bus aus mehreren Arduinos besteht aus mindestens einem Master und maximal 127 Slaves, also 128 Geräten. Diese Begrenzung ist durch den sieben Bit langen Adressraum vorgegeben [29].

Die Verbindungsleitungen können, abhängig von der gewählten Übertragungsgeschwindigkeit, Nähe zu möglichen Störquellen sowie Art und Weise der gewählten Abschirmung der genutzten Leitungen, mehrere Meter lang sein. Spezielle Entwürfe konnten sogar bis auf 55 Meter Länge ein sauberes Signal übertragen [27]; andere theoretische Berechnungen halten eine Übertragung bei einer Buslänge von bis zu 300 Metern für möglich [61]. Auch die Umsetzung von mehreren als Master agierenden Komponen-

ten innerhalb eines I²C Bussystems ist möglich; man spricht in diesem Zusammenhang von einem *Multi-Master-Bussystem* [21].

Die Informationen werden seriell mittels nur zwei verschiedener Leitungen zwischen den Arduinos übermittelt: eine Leitung ist als spezielle Datenleitung gedacht (die *Serial Data Line*, kurz SDA), die andere kontrolliert die Start- und Stopp-Signale des - beziehungsweise der - Master und dient daher als Taktgeber. Sie wird aus diesem Grund *Serial Clock Line* (SCL) genannt. In einem einfachen Entwurf mit nur einem Master gibt dieser das Taktsignal vor und initiiert jegliche Kommunikation auf dem Bus. Als Slaves agierende angeschlossene Mikrochips (IC, integrierte Schaltungen) und andere Mikrocontroller können selber keine Übertragung beginnen. [21]

Eine weitere weitverbreitete Alternative ist das *Serial Periphal Interface* (kurz SPI), das von Motorola entwickelt wurde, mit geringen Veränderungen von National Semiconductor aber auch unter dem Namen Microwire geführt wird [42].

Entwickelt wurde SPI mit der Absicht einen Datenaustausch zwischen einem als Master fungierenden Mikrocontroller und weiteren Peripherie-Chips zu ermöglichen. Explizit wurde auf eine hohe Übertragungsrate geachtet, welche jedoch Einschränkungen in der Leitungslänge fordert; so wird SPI häufig zur Kommunikation von fest auf Leiterplatinen verlöteten Komponenten verwendet.[59] Mittels speziellen Vorkehrungen können jedoch auch deutlich größere Strecken überbrückt werden [36].

Zur Kommunikation verwendet SPI im Gegensatz zu UART und I²C einen Bus, der aus vier Leitungen besteht: Zunächst zwei getrennten Datenleitungen zum Austausch vom und zum Master, erstere bezeichnet mit *Master Out Slave In* (MOSI) und letztere genannt *Master In Slave Out* (MISO). Außerdem gibt es eine Taktleitung, genannt *Serial Clock* (SCL), und eine Auswahlleitung namens *Slave Select* (SS) für jeden einzelnen Slave.

Wie auch bei I²C wird das Taktsignal nur am Master generiert, kann bei SPI jedoch mehrere Megahertz betragen [59]. Möchte man Daten von einem Slave empfangen oder ihm Daten zu kommen lassen, wird die entsprechende SS Leitung genutzt, um diesen in einen Bereitschaftsmodus zu versetzen. Diese zusätzliche Anbindung bringt zwar erhöhten Hardwareaufwand mit sich, ermöglicht es aber nahezu beliebig viele Slaves in einem solchen Bussystem zu verwenden. Es werden insgesamt $3+x$ Leitungen benötigt, wobei x die Anzahl der anzusteuernden Slaves darstellt.

Die bei den meisten Bausteinen genutzten getrennten Datenleitungen erlauben den Betrieb einer vollduplex Kommunikation und somit ein gleichzeitiges Senden und Empfangen von Daten. Meist werden acht Bit lange Worte oder Vielfache dessen übertragen. In Abweichung zum Regelfall kann ein Baustein jedoch auch längere oder kürzere Signale verarbeiten.

Ohne einen genau formulierten Standard gibt es noch weitere Ausnahmen, so zum Beispiel Bausteine, die keine vier Leitungen zur Kommunikation verwenden, oder LCD-Controller, welche über keinen Anschluss einer MISO Leitung verfügen [21]. Die offene Spezifikation gestattet ebenfalls Variation in weiteren Bereichen der Übertragung; so ist nicht festgelegt, ob eine Kommunikation mit dem niederwertigen (*least significant bit*) oder mit dem höherwertigen (*most significant bit*) Bit zuerst durchgeführt wird.

Ein Arduino verfügt über entsprechende Pins, welche das Steuern eins vollständiges SPI-Bussystem ermöglichen. Es kann mit einem maximalen Takt von circa 8 MHz betrieben werden, wobei 4 MHz der empfohlene Standardwert ist [58]. Verschiedene Arduino Shields nutzen SPI um ihre jeweilige Funktionalität an der Hauptplatine anzubinden.

Das Arduino Ethernet Shield stellt so dem Entwickler einen vollwertigen RJ45 Port zur Kommunikation per Ethernet sowie einen SD-Kartencontroller mit entsprechendem SD-Kartenslot bereit. Da kein kombinierter Controller für diese Funktionalitäten genutzt wird, sondern es sich um einen Netzwerk-Controller und einen seperaten SD-Kartencontroller handelt, sind diese als jeweils einzelne Slaves per SPI anzusprechen. Dies erklärt, warum kein gleichzeitiges Lesen oder Schreiben auf einer eingelegten SD-Karte und gleichzeitiges Kommunizieren per Ethernet möglich ist. [6]

2.5 Systemarchitektur Arduino

Im Folgenden wird eine grundlegende Beschreibung der Arduinosystemarchitektur und die aus ihr resultierenden Vor- und Nachteile erläutert. Peter Mandl beschreibt in seinem Buch *Grundkurs Betriebssysteme* eine vierstufige Entwicklung der Computertechnik und der damit verbunden Betriebssysteme [37]. Ein Arduniosystem als solches hat kein eigentständiges Betriebssystem, sondern nutzt einen so genannten austauschbaren *Bootloader*, welcher ein vorher aufgespieltes Programm ausführt und grundlegende für Arduino boardspezifische Einstellungen vornimmt [41]. Auf diese Weise wird beispielsweise der verbaute Quarz Oszillator als Taktgeber für die dadurch verfügbare Systemuhr verwendet [46].

Aufgrund eines fehlenden Betriebssystems kann der Entwickler nicht auf Funktionalitäten wie echtes Multithreading oder einen Scheduler zurückgreifen. Obgleich ambitionierte Projekte versuchen, diese Funktionalität durch trickreiches Nutzen der gegebenen hardwareseitigen Funktionalitäten zu ermöglichen, ist der ursprüngliche einheitliche Programmablauf durch ein einmaliges Aufrufen einer *setup()*-Routine zu Beginn und ein dann kontinuierliches Durchlaufen einer *loop()*-Funktion fest vorgegeben [16, 19].

Der Arduino ist ebenfalls, wie schon in Kapitel 2.2 beschrieben, ein Einprozessorsystem, so dass auf Hardwareseite alle verfügbaren Ressourcen zur Programmabarbeitung im jeweiligen gerade durchlaufenen Programmabschnitt belegt sind. Am Beispiel von ARINA lässt sich verdeutlichen, was dies für praktische Auswirkungen auf ein zeitkritisches System haben kann: ARINA versucht kontinuierlich mögliche IR-Signale zu empfangen und an eine Gegenstelle zu übertragen, ein Berechnen von beispielsweise HMACs der dann in Nachrichten verschickten Signale steht somit im Konflikt zum kontinuierlichen Empfang neuer Signale [43].

Es wäre daher wünschenswert diese beiden Aufgaben parallel durchzuführen und so sowohl alle eintreffenden Signale zu empfangen als auch authentifizierte Nachrichten verschicken zu können. Peter Mandl beschreibt dies als Mehrprogrammbetrieb oder auch

Multitasking, welches im Gegensatz zum Einprogrammebetrieb respektive Singletasking eine Parallelverarbeitung unterstützt [37].

Heutige Smartphones setzen ebenso wie Computer zunehmend auf Mehrprozessorsysteme, die durch Koprozessoren unterstützt werden. Das Apple iPhone 5s nutzt beispielsweise den Apple M7, einen Koprozessor, welcher explizit für die Verarbeitung von Bewegungsinformationen auch im Ruhezustand eingesetzt wird [55, 3]. Zusätzlich zum Apple M7 wird der Apple A7 verwendet und als Hauptprozessor genutzt. Dieser von Apple entworfene Chip vereint einen Mehrprozessorkern mit zwei unabhängigen Prozessoreinheiten und einem eigenständigen Grafik-Koprozessor [51].

Das Auslagern von Informationsverarbeitung auf Hardware, die auf diese Verarbeitung spezialisiert ist, wird bei der Nutzung von Grafikkarten oder speziellen Physik-Koprozessoren sehr deutlich; so können diese solche Berechnungen deutlich schneller durchführen als eine Allzweck-CPU [28].

Aber auch weniger berechnungsaufwändige Vorgänge werden auf externe Mikrocontroller ausgelagert, zum Teil werden sogar ausschließlich Mikroprozessoren dazu genutzt. Beispielsweise werden Tastatureingaben, welche per PS/2-Anschluss an den Rechner übergeben werden, zuvor durch einen in die Tastatur verbauten Mikrocontroller in entsprechende Scancodes umgewandelt [60].

Durch das für die Tastatureingaben definierte Protokoll leitet dieser Mikrocontroller einen *Interrupt Request* bei dem angeschlossenen Computer ein. Dieser weist den Computer an, seinen aktuellen Programmablauf zu unterbrechen und mittels einer *Interrupt Service Routine* (kurz ISR) auf den eingetroffenen Interrupt zu reagieren und den eintreffenden Scancode zu verarbeiten [54]. Solche Interrupts werden von modernen Betriebsystemen auf unterschiedliche Weise gelöst, jedoch verfügt der Arduino, wie schon zuvor beschrieben, über kein solches Betriebssystem, weshalb es dem Entwickler obliegt eine Interruptbehandlung entsprechend den Anforderungen seines Anwendungsfalles zu entwerfen.

Das direkte Bearbeiten von Interrupt Requests ist durch das Verlassen das aktuellen Progammabschnittes zwar möglich, jedoch führt dies bei zeitkritischen Aufgaben, wie dem Empfangen von IR-Signalen (siehe ARINA [43]), zu einem Verlust von eintreffenden Informationen. Auch kann es zu Dateninkonsistenzen kommen, wie sie durch das *Nonrepeatable Read Problem* beschrieben werden [18]. Dabei kann eine Datenänderung während der ISR Variablenwerte verändern, die von einer aktuell unterbrochenen Funktion zum Funktionsaufruf gelesen wurden.

Allgemein wird empfohlen, dass während einer ISR nur wenige Instruktionen durchgeführt werden und somit ein schnelles Zurückkehren in den allgemeinen Programmablauf möglich ist. Sind mehrere unterschiedliche Interrupts auf die ein Arduino reagieren soll möglich, so ist es besonders wichtig, die entsprechenden ISRs zügig zu bearbeiten. Dies hat den Grund, dass es nicht möglich ist mehrstufige Interrupts zu behandeln; trifft ein Interrupt zur Laufzeit einer schon aktiven ISR ein, so wird er verworfen und nicht behandelt.[11]

3 Entwurf

Im folgenden Kapitel wird der Entwurf dargelegt, der als Grundlage für die spätere Softwareentwicklung dient. Es wird zunächst erläutert, welcher Hardwareaufbau zur Kommunikation zwischen zwei Arduino Mega2560 konzipiert und welche Komponenten verwendet wurden. Ebenso wird der entsprechende Aufbau beschrieben, der für die Kommunikation zwischen einem Arduino Mega2560 und einem Atmel AT-SHA204 Chip verwendet wird. Besonderheiten und Probleme, die beim jeweiligen Entwurf beachtet werden müssen oder zu beobachten waren, werden ebenso dargelegt.

3.1 Arduino als Koprozessor

Der Systementwurf steht als grundlegende vorangehende Überlegung vor allen anderen Entwürfen. Im Folgenden wird beschrieben, welche System- und Umgebungsvoraussetzungen zu den Planungsentscheidungen führten, die im Hard- und Softwareentwurf umgesetzt werden. Einleitend wird die Wahl des Kommunikationsweges zwischen zwei Arduino Mikrocontrollern beschrieben. Der konkrete Aufbau und die Verbindung von zwei Arduino Mega2560 folgt darauf.

3.1.1 Systementwurf

Grundsätzlich wird eine Eigenentwicklung einer Kommunikationstechnologie wie I²C und SPI ausgeschlossen. Wie bereits erläutert, gibt es eine Reihe verschiedener solcher Technologien, die als anerkannte Industriestandards gelten und die sich durch langjährige Iterationen der Optimierung und Fehlerbeseitigung bewährt haben.
Zur Kommunikation zwischen zwei Arduinos stehen, wie zuvor in Kapitel 2.4 erläutert, verschiedene Möglichkeiten zur Verfügung. Bei der Abwägung des in diesem Projekt eingesetzten Kommunikationsweges wurden verschiedene Auswahlkriterien zugrunde gelegt: Es sollte versucht werden, die allgemeinen Systemeigenschaften der Arduino Mikrocontroller Plattform zu nutzen, dabei aber nicht einzuschränken. Wenn möglich sollen auch eingeschränktere Baugruppen wie ein Ardunio Uno unterstützt werden. Es soll weiterhin möglich sein, den Master mit verschiedenen Shields, wie beispielsweise ein Ethernet Shield in Kombination mit dem als Koprozessor agierenden Slave, zu nutzen. Die Nutzung des per USB-Verbindung auslesbaren seriellen Monitors sollte ebenfalls nicht behindert werden. Damit sich ein Slave Arduino auch in schon gegebene Projekte integrieren lässt, sollten möglichst wenige Pin Steckplätze belegt werden. Bei-

spielhaft sei hier das Projekt ARINA zu nennen, dass bereits eine Reihe von Pins des Arduino belegt [43].

Zeitkritische Prozesse, wie das Empfangen von IR-Signalen, sollten nur in kürzestmöglichem Ausmaß unterbrochen werden, weswegen die zu wählende Kommunikationsart möglichst kurze Gesamtlaufzeiten sowie eine hohe Datenrate aufweisen sollte. Ein erweiterbarer Entwurf, der zusätzliche Koprozessoren sowie möglichst lange Verbindungswege zwischen Master und Slave gestattet, ist ebenfalls ein wichtiger Aspekt.

Diese Anforderungen führten mit den in Kapitel 2.4 erarbeiteten Argumenten dazu, I²C als Grundlage der Kommunikation zwischen Master und Slave zu wählen. Weniger benötigte Verbindungen, längere mögliche Verbindungswege und die einfachere Erweiterbarkeit eines bestehenden Bussystems ohne weitere Verkabelung waren der Grund, I²C den Vorzug vor SPI zu geben. UART wurde wegen seiner auf Punkt zu Punkt Kommunikation ausgelegten Spezifikation nicht ausgewählt; zudem verfügt ein Arduino Uno nur über eine UART Kommunikatonsschnittstelle, was den angestrebten Entwurf auf diesen Arduino Modellen einschränken würde.

Die I²C Kommunikation bietet wie bereits vorgestellt eine systematische Einschränkung im Datenaustausch zwischen Master und Slave; so ist es dem Slave nicht möglich Daten direkt an den Master zu übertragen. Eine Übertragung von Slave zu Master ist nur auf eine zuvor initiierte Anfrage des Masters möglich.

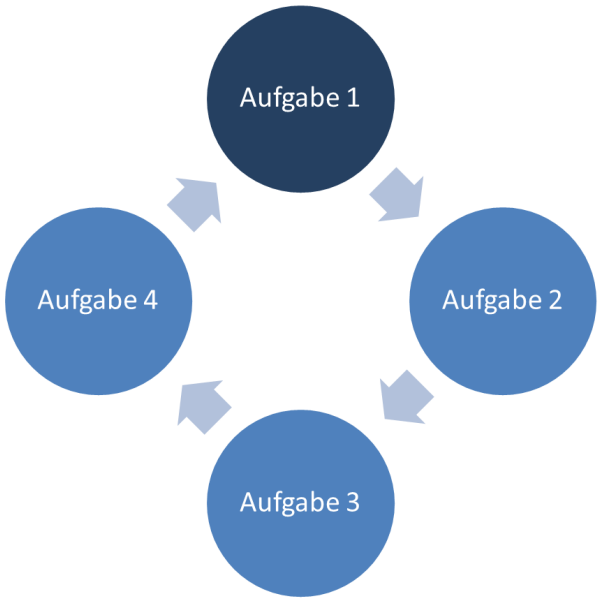

Abbildung 2: *Darstellung einer kontinuierlichen Abarbeitung, mit einem aktiven Prozessor, bei vier unterschiedlicher Aufgaben, mittels des Round Robin Verfahrens.*

Durch ein Round Robin Verfahren, siehe Abbildung 2, kann ein Master kontinuierlich alle ihm bekannten Slaves auf das Vorhandensein von neuen Daten hin abfragen. Da dies jedoch stets ein vollständiges Auf- und Abbauen der Kommunikation zwischen Master und Slaves bedeuten würde, ist in dieser Arbeit die Entscheidung getroffen worden,

neue, für den Master bestimmte Daten per Interrupt am selbigen anzumelden, so dass dieser sie in einem entsprechend dafür vorgesehenen Programmabschnitt beim Slave erfragt. So wird ein möglicherweise zeitkritischer Programmabschnitt, in dem sich der Master befindet, nur minimal unterbrochen und ebenso wird ein möglicher Konflikt zwischen Variablenwerten des aktuellen Programmabschnitts und durch die vom Slave gelieferten neuen Werte bewusst vermieden.

3.1.2 Hardwareaufbau

Wie zuvor erläutert, wird in dieser Arbeit ein Koprozessor, fortan *Slave* genannt, in Form eines Arduino Mega2560 genutzt, um Berechnungen von SHA256 Summen sowie HMAC zu ermöglichen. Dies erlaubt es, dass ein als *Master* fungierender Arduino Mega2560 für diese Berechnungen keine Laufzeit von zeitkritischen Aufgaben entbehren muss. Der zu Grunde liegende Aufbau des Arduino-Koprozessor Entwurfs besteht demnach aus zwei Arduino Mega2560, welche mittels fünf Steckverbindungen verbunden sind.

Zwei der Verbindungen dienen der Spannungsversorgung des als Koprozessors agierenden Arduino. Entsprechend ist der $5V$ Steckplatz des Master-Arduinos mit dem *Vin* Steckplatz des Slave-Arduinos verbunden und ebenso die *GND* Steckplätze der beiden Arduinos. Dieser Aufbau wird als eine Alternative zur Spannungsversorgung per USB beschrieben [9]. Der als Master agierende Arduino wird mittels einer USB Verbindung mit Spannung versorgt. Ein alternativer Aufbau, in dem beide Arduinos per USB mit einem Computer verbunden sind und keine zusätzliche Spannungsversorgung mit einer $5V$ zu *Vin* Verbindung notwendig ist, ist ebenso möglich; hierbei würde die *GND* zu *GND* Verbindung beibehalten. Dieses Setup erlaubt es, die seriellen Ausgaben beider Arduinos auszuwerten und kann somit zur Fehleranalyse hilfreich genutzt werden.

Der Ardunio Mega2560 verfügt insgesamt über sechs Interrupt Pins, jedoch stehen nicht alle sechs Pins zur Verfügung. Aufgrund der internen Belegung werden die als Interrupt Pins ausgewiesenen Pins auch für andere Funktionalitäten des Arduinos genutzt, so teilen sich die digitalen Pins 20 und 21 ihre Funktionalität als Interrupt Pins und dienen gleichzeitig als Pins zur Kommunikation per I²C. Ebenso sind die digitalen Pins 18 und 19 sowohl als Interrupt Pins als auch als Pins zur seriellen Kommunikation ausgewiesen. Es finden sich weitere Überschneidungen zwischen als Interrupt Pins beschriebenen Steckplätzen und Pins, die zur Pulsweitenmodulierung ausgewiesen wurden. [9]

Um auch durch den hier beschriebenen Entwurf keine unnötigen Einschränkungen für Projekte zu erzeugen, die eine Kombination dieser Funktionalitäten sowie das hier beschriebe System einsetzen wollen, wurde darauf geachtet einen Interrupt Pin zu wählen der überschneidungsfrei ist. Daher wurde Pin 3 als Interrupt Pin auf Seite des Masters gewählt; der Interrupt wird durch den Pin 10 des Slave Arduinos angesteuert. Wie bereits zuvor dargelegt werden Pin 20 und 21 für die Kommunikation per I²C genutzt; diese Pins sind ebenso jeweils untereinander verbunden. Ein Schaltplan des beschriebenen Aufbaus wird in Abbildung 3 gezeigt, ebenso zeigt Abbildung 4 das Foto eines funktionstüchtigen Aufbaus.

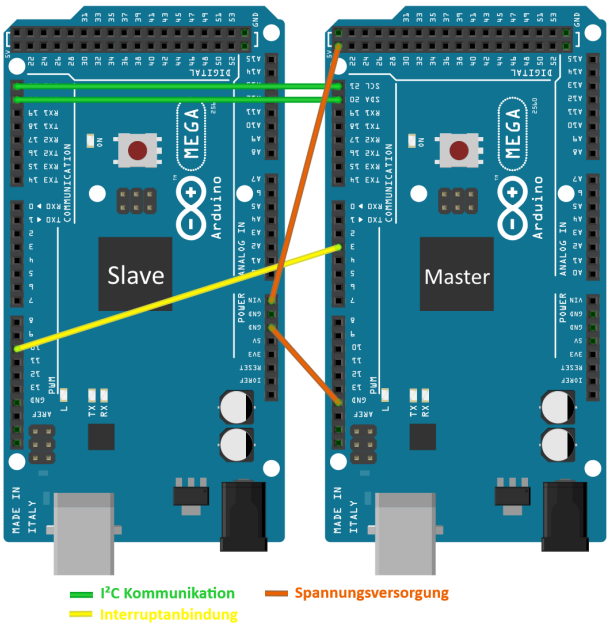

Abbildung 3: *Darstellung der Verbindungen zwischen Master und Slave Arduino Mikrocontroller. Erstellt mit Fritzing [24].*

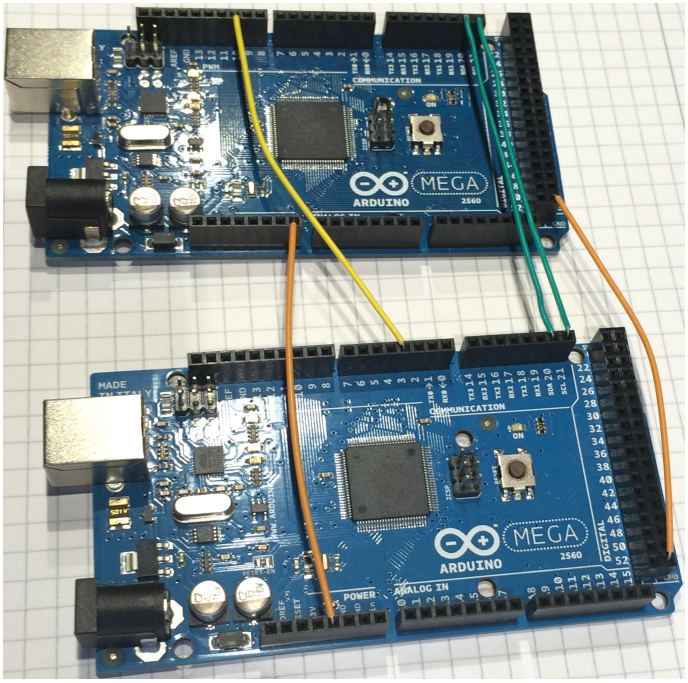

Abbildung 4: *Aufbau des Arduino Koprozessor Systems mit zwei Ardunio Mega2560 Mikrocontrollern*

3.2 Atmel AT-SHA204

Wie auch im zuvor beschriebenen Systementwurf werden auch bei der Beschreibung der Konzeption von Arduino in Kombination mit einem Atmel AT-SHA204 Mikrochip grundlegende vorangehende Überlegungen beschrieben. Eine Erläuterung der System- und Umgebungsvoraussetzungen begründen die getroffenen planerischen Entscheidungen für Hard- und Softwareentwurf. Beschrieben wird zunächst der Kommunikationsweg zwischen einem Arduino Mikrocontroller und dem Atmel AT-204 Mikrochip. Die Beschreibung des konkreten Aufbaus und der Verbindung schließen dieses Kapitel ab.

3.2.1 Systementwurf

Um den Atmel AT-SHA204 zur Authentifizierung nutzen zukönnen, müssen einige Besonderheiten und Spezifikationen beachtet werden. Auf dem Chip wird - wie bereits in Kapitel 2.3 angedeutet - zwischen drei verschiedenen Bereichen unterschieden: der *ConfigZone*, *DataZone* und der OTP (*One Time Programmable*) Zone. Die beiden letztgenannten Zonen weisen dabei mit jeweils 512 Byte beziehungsweise 64 Byte Größe den größten Teil der insgesamt zur Verfügung stehenden 664 Bytes auf [13]. Die sechzehn Slots (siehe Kapitel 2.3), die für die genannten bis zu 256 Bit langen Schlüssel genutzt werden können, sind hierbei in der DataZone zu finden. Die 88 Byte große ConfigZone beinhaltet neben der Seriennummer vor allem Zugriffsinformationen für die DataZone und muss konfiguriert werden, bevor mit der DataZone in gewünschter Weise verfahren werden kann. Damit die DataZone und OTPZone beschrieben werden können, muss zunächst die ConfigZone *gelocked* (oder *gesperrt*) werden. Ohne vorheriges *Locken* (oder *Sperren*) der ConfigZone sind die DataZone und die OTPZone weder beschreibbar noch lesbar.

Die werkseitigen Werte der ConfigZone setzen unterschiedliche Lese- und Schreibeinschränkungen für die entsprechenden Datenfelder in der DataZone; es lässt sich kein einheitliches Muster erkennen. In dieser Arbeit wurde sich dafür entschieden, keine weiteren Einschränkungen für die Werte in der DataZone zu treffen. Daher wurden die Lese- und Schreibrechte entsprechend gesetzt.

Für diese Arbeit und in einem experimentellen Kontext ist es nicht kritisch, dass Schlüsselmaterial, welches in den Datenfeldern der DataZone gespeichert ist, im Klartext wieder ausgelesen werden kann. Wird der Atmel AT-SHA204 im produktiven Umfeld verwendet, muss sich mit den entsprechenden Sicherungsmaßnahmen, die durch das Setzen entsprechender Beschränkungen für die jeweiligen Datenfelder in der DataZone ermöglicht wird, auseinandergesetzt werden.

Das Sperren erfolgt mittels des entsprechenden *execute*-Befehls aus der AT-SHA204-Breakout Bibliothek, die im Kapitel 4.2.2 näher beschrieben wird. Bei der Kommunikation mit dem vorverarbeiteten Atmel AT-SHA204 ist zu beachten, dass diese auf *least-significant-bit-first* basiert, wohingegen I²C auf *most-significant-bit-first* setzt.

Der schon zuvor genannte *execute*-Befehl stellt die Grundlage für die Kommunikation mit dem Atmel AT-SHA204 dar; er nimmt über verschiedene Parameter Byte-Folgen

entgegen. Eine genaue Erläuterung zu den umgesetzten Befehlen findet sich im Kapitel 4.2.

3.2.2 Hardwareaufbau

Der Atmel AT-SHA204 ist, wie bereits in Kapitel 2.3 vorgestellt, in verschiedenen Ausführungen erhältlich. In dieser Arbeit wurde ein schon für den Einsatz vorbereiteter Chip verwendet. Dieser ist auf einer Trägerplatine verlötet und mit einem 1 MΩ starken Pullup-Widerstand für ein dadurch *sauberes* Signal versehen. Ein ebenso verbauter 0,1 Farad Kondensator stellt für die Baugruppe eine *geglättete* Spannung sicher [53]. Der beschriebene Aufbau ist in der Abbildung 5 des Schaltplans der Trägerplatine nachzuvollziehen.

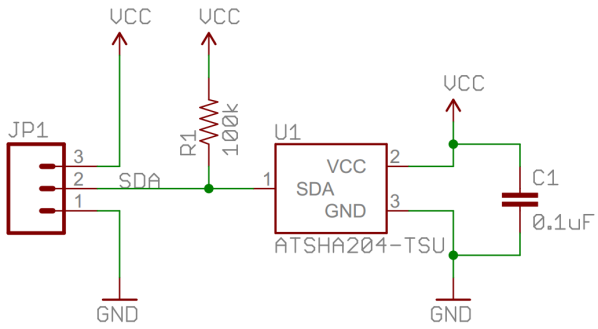

Abbildung 5: *Schaltplan der als „ATSHA204 Authentication Chip Breakout“ erwerbbaren Trägerplatine. Abbildung unter „CC BY 3.0 DE“ Lizenz, entnommen aus [53]*

Der zuvor beschriebene Breakout Aufbau wurde mit einer Steckverlängerung verlötet, was den Aufbau in Verbindung mit einer Steckplatine vereinfacht. Durch schon vorhandenen Widerstand und Kondensator sind keine weiteren Komponenten erforderlich, so dass der Mikrochip mit entsprechenden Steckverbindungen mit dem Arduino Mikrocontroller verbunden werden kann.

Der Chip wird entsprechend mit der Spannungsquelle an dem 5*V* Steckplatz sowie mit einem *GND* Steckplatz des Arduino verbunden. Der am Chip als *SDA* beschriebene Anschluss wird an einen freien pulsweitenmodulierbaren digitalen Pin angeschlossen; im hier verwendeten Aufbau wurde der Pin 7 gewählt. Der beschriebene Aufbau wird in Abbildung 6 schematisch dargestellt.

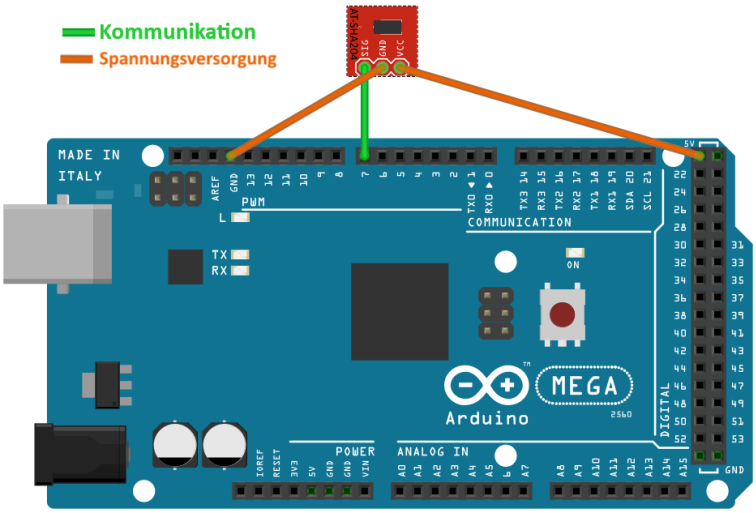

Abbildung 6: *Darstellung der Verbindungen zwischen Atmel AT-SHA204 und Arduino Mega2560 Mikrocontroller. Erstellt mit Fritzing [24].*

4 Umsetzung

In diesem Kapitel wird auf den Entwurf des 3. Kapitels sowie den in Kapitel 2 beschriebenen Grundlagen aufgebaut und die softwareseitige Umsetzung dargelegt. Es werden die Funktionen des dazu erstellten Systems vorgestellt und erklärt. Zur Programmierung wurde die auf der Webseite des Arduino Projekts frei verfügbare Arduino IDE in der Version 1.0.5 R2 verwendet.

Beginnend mit der Beschreibung der entwickelten Arduino Koprozessor Software wird auch für die auf dem Atmel AT-SHA204 basierende Implementierung eine aufgeteilte Erläuterung gegeben. Einleitend wird der Softwareentwurf für die jeweilige Variante dargelegt und der logische Ablauf erklärt. Anschließend wird auf verwendete Bibliotheken und gegebenenfalls nötige Anpassungen an diesen eingegangen. Im Abschnitt Implementierung werden wichtige umgesetzte Funktionen erläutert. Auf Besonderheiten, die bei der jeweiligen Implementierung beachtet wurden, wird jeweils abschließend eingegangen.

4.1 Arduino als Koprozessor

Zunächst wird für die Variante Arduino Koprozessor der logische Aufbau der implementierten Softwarelösung beschrieben. Dabei ist der Softwareentwurf unterteilt in Master und Slave. Es werden verwendete Bibliotheken benannt und kurz die für diese Arbeit relevanten Funktion aus diesen Bibliotheken beschrieben sowie getroffene Anpassungen erläutert. Im Abschnitt Implementierung werden umgesetzte Funktionen erläutert und auf Besonderheiten eingegangen.

4.1.1 Softwareentwurf

Der Softwareentwurf beschreibt die vorangegangenen Überlegungen, welche Grundlage der anschließend beschriebenen Implementierung sind. Zunächst erfolgt die Beschreibung des für den als Master agierenden Arduino Mikrocontroller entworfenen Softwareentwurfs und anschließend wird auf die für den Slave entworfene Software eingegangen.
Durch den im Kapitel 2.5 beschriebenen Aufbau ist es nicht ohne Weiteres möglich, mehrere Prozesse gleichzeitig auf einem Arduino Mikrocontroller auszuführen. Aufgrund der in der Zielsetzung (Kapitel 1.1) beschriebenen Anforderung, zeitkritische Prozesse so wenig wie möglich einzuschränken, wurde eine thread-basierte Umsetzung nicht

weiter verfolgt. Existierende Lösungen zu einer thread-basierenden Prozessarchitektur auf einem Arduino Mikrocontroller setzen hierzu geschickt die interne, durch den Quarz Oszillator getaktete, Systemuhr als Interruptmechanismus ein [26]. Wie bereits erwähnt (Kapitel 2.5), kann jedoch kein mehrstufiger Interrupt verarbeitet werden, so dass Interrupts in solch einer Lösung schon durch den gewählten Softwareentwurf nicht immer behandelt werden können.

In dieser Arbeit wurde daher ein an einen endlichen Automaten angelehnter Lösungsansatz gewählt. Der grundlegende Aufbau lässt den als Master fungierenden Arduino Mikrocontroller seinen hauptsächlichen Programmablauf, beispielsweise das Senden und Empfangen von IR-Signalen wie im Projekt ARINA, umsetzen und wird nur durch einen *kurzen* Interrupt unterbrochen. In diesem Zusammenhang beschreibt ein solcher *kurzer* Interrupt das Setzen eines einzelnen Variabelwertes. Als Folge finden keine Berechnungen oder anderen zeitintensiven Verarbeitungsschritte statt.

Der Master kann unter Verwendung der implementierten Funktionen verschiedene Funktionalitäten des als Koprozessor agierenden Slave Arduino Mikrocontrollers nutzen:

- **SHA256 Summe** Ein an den Slave Arduino Mikrocontroller übergebener `String` dient als Eingabe für die Berechnung einer SHA256 Summe nach dem in FIPS180 und RFC4634 beschriebenen SHA256 Hashalgorithmus [25, 1].
- **HMAC** Der Slave Arduino Mikrocontroller berechnet basierend auf der ihm übergebenen Nachricht und einem Schlüssel einen *Keyed-Hash Message Authentication Code*, siehe Kapitel 2.1.
  Der dazu benötigte Schlüssel kann über zwei verschiedene Quellen in die Berechnung mit eingehen. So ist es einerseits möglich einen beispielsweise nur einmalig zu verwendenden Schlüssel für eine einzelne Berechnung an den Slave zu übergeben. Alternativ können bis zu acht Schlüssel auf dem als Slave fungierenden Arduino Mikrocontroller gespeichert werden.

  Diese einmal übertragenen Schlüssel können auch für spätere Berechnungen verwendet werden und können nur durch Überschreiben neu gesetzt werden. Sollte die Spannungsversorgung des Slave Arduinos unterbrochen werden, werden gespeicherte Schlüssel nicht erhalten, da sie in einem so genannten flüchtigen Speicher abgelegt wurden. Dieser Speicher wurde bewusst gewählt um bei einem physischen Entwenden des Koprozessors die Gefahr eines unbefugten Auslesen der gespeicherten kryptographischen Schlüssel zu minimieren.

  Einmal übertragene Schlüssel können zu dem nicht wieder ausgelesen werden, es ist nur möglich sich die SHA256 Hashsumme über jeden Schlüssel zurückgeben zu lassen.
- **Zufallswert** Angelehnt an die Funktionalität des Atmel AT-SHA204 (siehe Kapitel 2.3), kann bei dem in dieser Arbeit entworfenen Slave ein sich bei jeder Anfrage unterschiedlicher Zufallswert angefragt werden. Dies kann zur Gewinnung eines sogenannten *Salt* (eng. Salz) Wertes genutzt werden, der verknüpft mit einer Eingabe die Entropie einer über diese Verknüpfung berechneten Hashsumme erhöht.

- **Echo32** Hierbei werden die ersten 32 Byte einer übertragenen Nachricht an den Master zurückgegeben; diese Funktion eignet sich insbesondere zu Testzwecken des Hardwareaufbaus.

Eine abgeschlossene Berechnung des Slaves wird dem Master per Interrupt angekündigt. Das Anfordern des berechneten Rückgabewertes wird jedoch nicht in der *Interrupt Service Routine* abgehandelt, da dies zeitkritische Prozesse unnötig lange unterbrechen würde. Stattdessen wird ein entsprechender Variablenwert, welcher für eine Fallunterscheidungsanweisung (engl. *Switch-Case-Statement*) genutzt wird, gesetzt. Diese Fallunterscheidung umschließt den Prozessablauf, welcher vom Master durchlaufen wird. Daher wird eine ursprüngliche Bearbeitung, beispielsweise das Berechnen von als IR Signalen auszugebenden Steuercodes, oder explizit das Behandeln eines zuvor registrierten abholbereiten Rückgabewertes vom Slave durchgeführt. Diese klare Trennung von Berechnungsprozessen im Programmablauf unterstützt die Vermeidung von eventuell auftretenden Dateninkonsistenzen, siehe Kapitel 2.5.

Der Slave ist ebenfalls nach dem Prinzip eines endlichen Automaten entworfen worden, sein Funktionsumfang wurde durch die vom Atmel AT-SHA204 zur Verfügung gestellten Funktionen inspiriert. Das Ablaufdiagramm in Abbildung 7 stellt den Programmablauf des Slaves entsprechend dar.

Beginnend mit einem Bereitschaftszustand (engl. *Idle state*) wartet der Koprozessor auf eintreffende Übertragungen. Zu Beginn jedes Auftrags wird anhand eines einzelnen Bytes der Slave Arduino entsprechend auf eine mögliche nächste bis zu 255 Byte lange Nachricht vorbereitet. Der Slave wechselt dazu in einen „Instruktionsdaten vorhanden“ Zustand (eng. *Data available state*). Das einzelne Kommandobyte spezifiziert in seinen drei höherwertigen Bits welche Funktion der Master beim Slave anfragt; eine genaue Erläuterung dazu findet sich im Abschnitt 4.1.3. Die fünf niederwertigen Bits werden für die Operationen verwendet, die einen bestimmten Speicherplatz (engl. *Slot*) in ihrem Aufruf verwenden; es können 32 Speicherplätze auf diese Art adressiert werden. Eine genaue Erläuterung, welche Funktionen dies sind, findet sich ebenfalls im folgenden Abschnitt 4.1.3.

Mittels der aus dem Kommandobyte entnommenen Information kann daraufhin entschieden werden, ob direkt eine Rückgabe generiert werden kann, oder ob mit einer weiteren Nachricht vom Master noch zusätzliche, für die entsprechende Operation notwendige, Daten übertragen werden. Nur die Funktionen, die eine Zufallszahl generieren und die Hashsumme eines zuvor gespeicherten Schlüssels ausgeben, können direkt abgearbeitet werden und erfordern keine weiteren Informationen vom Master. So kann direkt in den „Antwort vorbereitet“ Zustand (engl. *Return value ready state*) gewechselt werden. Alle anderen Operationen erfordern noch weitere Werte, die, wie zuvor erläutert, in einer zweiten Nachricht vom Master übertragen werden.

Sind alle erforderlichen Daten vorhanden, kann der Slave in einen entsprechenden Zustand wechseln (engl. *Data loaded state*) und mit der Berechnung der zuvor gewählten Operation beginnen. Wenn alle erforderlichen Berechnungen abgeschlossen sind, wechselt auch in diesem Fall der Slave in den schon vorher beschriebenen „Antwort vorbereitet“ Zustand. In diesem Zustand angelangt, werden somit keine Änderungen

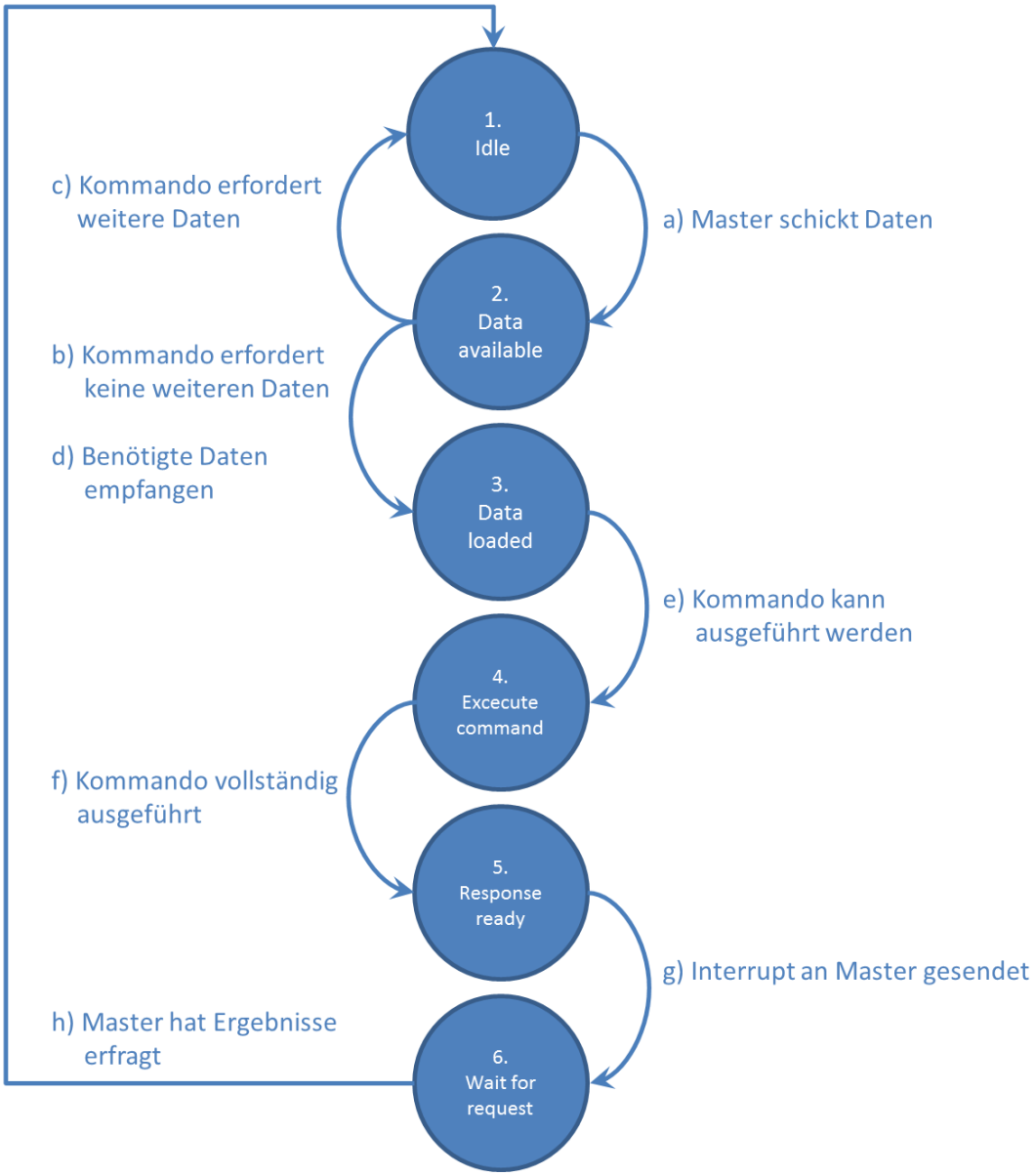

Abbildung 7: *Darstellung des Slave Programmablaufs anhand eines Ablaufdiagramms*

an nun zu übertragenden Daten erwartet und es kann eine Interruptmeldung an den Master gemacht werden; dieser wird dadurch den entsprechenden Variablenwert für seine routinemäßig durchlaufene Fallunterscheidung setzen und kann beim nächsten Durchlauf eine Daten anfragende Funktion aufrufen.

Wie bereits im Abschnitt 2.4 und 3.1.1 erläutert ist dies notwendig, da eine Kommunikation zwischen zwei, mittels eines I²C Bus kommunizierenden, Mikrocontrollern nur vom jeweiligen I²C Master angestoßen werden kann. Die Übertragung der zuvor angefragten Daten ist damit abgeschlossen und der Slave kann in den Bereitschaftszustand zurückkehren.

4.1.2 Verwendete Bibliotheken

Für die Umsetzung des Arduino Koprozessors wurde eine externe Bibliothek verwendet und eine weitere, in die Arduino IDE integrierte Bibliothek, angepasst.

Für die Kommunikation per I²C wurde die im Arduino Projekt entwickelte *Wire* Bibliothek genutzt [8]. In ihrer ursprünglichen Ausführung unterstützt sie das Senden und Empfangen von bis zu 32 Byte langen Nachrichten. Diese kurze Nachrichtenlänge ist mutmaßlich aus Kompatiblitätsgründen zu anderen Mikrocontrollern, welche mit noch weniger Arbeitsspeicher ausgestattet sind, gewählt worden. In der Dokumentation findet sich keine Erläuterung zur Wahl der Nachrichtenlänge.

Durch eine Anpassung im Quellcode konnte dieses Limit jedoch auf 255 Bytes für den Versand einer einzelnen Nachricht bei der Kommunikation vom Master zum Slave angehoben werden. Hierzu wurde in den Headerdateien der Bibliothek die Präprozessordirektive, welche die Größe des Nachrichtenpuffers angibt, von 32 auf 256 Byte, geändert.

Für die Berechnung von SHA256 Hashsummen sowie HMACs wurde eine als *Cryptosuite* bezeichnete Bibliothek, entwickelt von Peter Knight, verwendet. Durch beigefügte Tests konnte Knight zeigen, dass seine Bibliothek SHA1 und SHA256 Hashsummen sowie auf diesen Hashalgorithmen basierende HMACs nach den in Kapitel 2.1 vorgestellten Standards FIPS180 und RFC4868 berechnet. [34]

4.1.3 Implementierung

In diesem Abschnitt wird die softwareseitige Implementierung für die Kommunikation zwischen Arduino Mikrocontrollern und die zur Berechnung von SHA256 Hashsummen sowie HMACs auf einem als Koprozessor agierenden Arduino Mikrocontroller notwendigen Funktionen vorgestellt. Beginnend beim als Master agierenden Arduino Mega2560 wird dem Datenfluss entsprechend gefolgt und die jeweils Daten verarbeitende Funktion näher erläutert.

Wie schon in Abschnitt 4.1.1 erläutert, können verschiedene Funktionen des Koprozessors in einem, das entworfene System nutzenden, Softwareentwurf verwendet werden. Hierzu werden die entsprechend kodierten Kommandocodes und Nachrichtendaten übertragen. Für eine einfache und intuitive Verwendung sind einzelne Funktionen vorhanden, so dass ein Einsatz des Koprozessors ohne weitere Kenntnisse bezüglich des zur Kommunikation notwendigen Ablaufs möglich ist. Es wurden insgesamt acht unterschiedliche Funktionen implementiert.

Für die Berechnung von SHA256 Summen nach den in Kapitel 2.1 vorgestellten Standards kann die `SHA256sum(msg)` Funktion genutzt werden, der übergebene `String` Wert kann eine Länge von bis zu 255 Byte haben. Wie bei allen parameterabhängigen Funktionen des Koprozessors wird der Eingabewert zu Beginn auf eine valide Länge hin geprüft. Um die Konsistenz der Rückgabewerte des Koprozessors sicherzustellen wird ebenso geprüft, ob eine neue Anfrage zu einem Zeitpunkt durchgeführt wird, zu dem keine noch ausstehenden Ergebnisse erwartet werden. Sind diese vorbereitenden Prüfungen erfolgreich durchlaufen, wird ein entsprechender Kommandocode an den Slave übertragen.

Vor dem Senden des übergebenen `Strings` wird eine kurze Wartezeit (engl. *Delay*) eingehalten, so dass der Slave für die anstehende Berechnung entsprechend konfiguriert werden kann. Eine weitere Erläuterung zu der Notwendigkeit dieser eingehaltenen Wartezeit erfolgt im folgenden Abschnitt 4.1.4. Die eigentliche Übertragung erfolgt dann als eine Bytefolge durch die von der *Wire* Bibliothek zur Verfügung gestellten Funktion `write`. Auf eine abgeschlossene Übertragung hin wird ein entsprechender Variablenwert gesetzt, welcher für die zuvor erläuterte Prüfung auf noch ausstehende Ergebnisse genutzt wird.

Auf Seite des als Slave agierenden Arduino Mikroprozessors wird das zuerst übertragene Kommandobyte entsprechend dekodiert und der Arduino in einen der gewählten Berechnung entsprechenden Zustand gebracht. Zur Berechnung von SHA256 Summen werden entsprechende Berechnungen von der vorgestellten *Cryptosuite* Bibliothek durchgeführt und das Ergebnis in einem Rückgabearray gespeichert.

Um fertige Berechnungsergebnisse dem Master Arduino anzukündigen, wird die Hilfsfunktion `sendInterrupt()` genutzt, welche durch einen kurzen Spannungswechsel an einem per Präprozessordirektive definierten digitalen Pin den Interruptpin des Masters anspricht und diesen zum Behandeln der für diesen Pin definierten *Interrupt Service Routine* veranlasst.

Die hierbei behandelnde Hilfsfunktion `handleSlaveReturn()` setzt ausschließlich eine entsprechende Variable, welche beim nächsten Fallunterscheidungsdurchlauf des Masters ausgewertet wird, wie auch schon in Abschnitt 4.1.1 erläutert. Durch Auswerten des Variablenwertes wird der Master die Hilfsfunktion `requestHashedData(I2Caddress, reqSize)` aufrufen, welche mit der, ebenfalls per Präprozessordirektive definierten, I²C Adresse des Slaves und der für die erwartete Antwort spezifischen Länge parametisiert ist.

Beim Aufruf von Hashfunktionen oder Antworten, die auf deren Ergebnissen basieren, ist die jeweilige charakteristische Blocklänge der Hashfunktion, wie in Kapitel 2.1 erläutert, als Antwortlänge zu wählen. Wurde die komplette Länge der Antwort übertragen kann das Ergebnis als Empfangen angesehen werden.

Das so erhaltene Ergebnis kann nun in weitere Prozesse des Master Arduinos einfließen und die Aufgabe des Koprozessors ist abgeschlossen. Dieser Ablauf ist für die Funktion `Echo32(msg)` sowie bei der Funktion `GetRND32()` bis auf das nicht notwendige Übertragen eines Eingabewertes äquivalent. Die implementierte `Echo32(msg)` Funktion realisiert die als **Echo** beschriebene Funktionalität, wobei die `GetRND32()` Funktion zur Generierung eines 32 Byte langen **Zufallswertes** genutzt werden kann (vgl. Kapitel 4.1.1).

Der am Beispiel der Funktion `SHA256sum(msg)` ausführlich beschriebene Ablauf der Kommunikation zwischen Master und Slave ist für die folgend erläuterten Funktionen identisch. Es werden nun einzelne, für die jeweilige Funktion besonderen, Aspekte genauer erläutert und auf eine Wiederholung des schon dargelegten Kommunikationsweges verzichtet.

Die Funktionen `HMACTempKey(msg)` und `HMACSlotKey(msg, slotid)` können zur Berechnung einer HMAC nach den ebenfalls in Kapitel 2.1 vorgestellten Standards genutzt werden. Beide Funktionen erfordern es, dass vor ihrem Aufruf ein entsprechender für die Berechnung notwendiger Schlüssel auf den Arduino Koprozessor übertragen worden ist, dies ist vor Aufrufen sicherzustellen. Soll häufiger auf den selben Schlüssel zurückgegriffen werden, kann dieser zuvor mit der Funktion `SetKey(msg, slotid)` auf den Slave übertragen werden. Der bis zu 255 Byte lange Schlüssel wird in einem dafür angelegtem Array gespeichert, der Zugriff erfolgt mittels der der gewählten *slotid*. In dem in dieser Arbeit entworfenen Slave Arduino Koprozessor können, wie schon im Abschnitt 4.1.1 beschrieben, bis zu acht solcher Schlüssel gespeichert werden.

Das direkte Auslesen eines Schlüssels wird aus Sicherheitsgründen nicht unterstützt. Es ist jedoch mittels der Funktion `GetKey(slotid)` möglich, sich eine SHA256 Hashsumme, welche den gewählten Schlüssel als Eingabe hatte, vom Koprozessor zurückgeben zu lassen.

Wird ein beständig wechselnden Schlüssel zur Berechnung der HMAC benötigt, kann alternativ auf die Funktion `HMACTempKey(msg)` zurückgegriffen werden; hierzu wird ein entsprechend nur für den Zeitraum der Berechnung vorgehaltener Schlüssel per `SetTempKey(msg)` Eingang übertragen. Wie schon in Abschnitt 4.1.1 erläutert, wird ein in einem Byte codierter Kommandocode zu Beginn jeder an den Koprozessor ausgelagerten Berechnung übertragen, um die jeweilige Funktionalität zu spezifizieren. Die

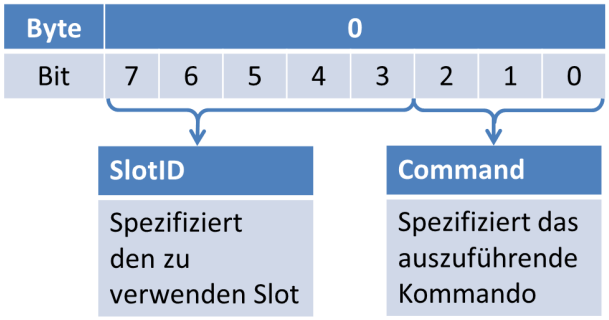

Abbildung 8: *Darstellung des Aufbaus eines Kommandobytes zur Übertragung von Anweisungen an den Slave*

Abbildung 8 zeigt den Aufbau eines solchen Bytes. Die in der Tabelle von Abbildung 9 dargelegte Zuweisung beschreibt zusätzlich bei welchen Operationen eine zweite Übertragung mit den entsprechend zu verarbeitenden Daten notwendig ist.

Command	Kommando	Erwartet Daten
0	Berechne SHA256 Summe	✓
1	Echo der Eingabe	✓
2	Temp. Schlüssel setzen	✓
3	Berechne HMAC mit Temp. Schlüssel	✓
4	Berechne HMAC mit gespeichertem Schlüssel	✓
5	Speichere Schlüssel	✓
6	Gib SHA256 Summe eines gespeicherten Schlüssels zurück	×
7	Berechne Zufallszahl	×

Abbildung 9: *Tabelle zur Darstellung der Zuweisungen von Kommandowerten zu ausführbaren Operationen sowie Verdeutlichung von Operationen, die weitere Daten in einer anschließenden Übertragung erwarten.*

4.1.4 Besonderheiten und Probleme

Im Folgenden wird explizit auf Probleme welche bei der Umsetzung und Implementierung des Koprozessor Systems aufgetreten sind eingegangen. Ebenso werden Besonderheiten des realisierten Aufbaus herausgestellt.

Jegliche Kommunikation von Master an Slave kann eine maximale Nachrichtenlänge von 255 Bytes nicht überschreiten; diese Einschränkung ergibt sich aus der verwendeten *Wire* Bibliothek. Durch die an ihr vorgenommenen Änderungen ist nunmehr eine fast acht Mal so große Nachrichtenlänge für eine einzelne Übertragung möglich. Es wurde

von der Entwicklung eines Protokolls, welches die Übertragung mehrerer einzelner Nachrichten ermöglicht, im Zuge dieser Arbeit abgesehen, da dies den Umfang des in der Zielsetzung erläuterten Hauptaspektes - das Auslagern von kryptographischen Berechnungen - verlässt.

Es ist möglich bis zu neun, ein temporärer sowie acht mittels SlotID wählbare, Schlüssel auf dem Slave zu speichern, diese werden im aktuellen Entwurf bei einer Spannungunterbrechung jedoch nicht erhalten und müssen neu übertragen werden. Das Abspeichern von mehr als nur neun Schlüsseln sowie deren Erhalt über eine Spannungsunterbrechung hinaus ist möglich, hierzu muss ein auf dem Arduino Mega2560 vorhandene nicht flüchtige Speicherbereich gewählt werden. Dieser kann jedoch bei entwenden des Slaves auch von Unbefugten ausgelesen werden. Der aktuelle Entwurf sieht es nicht vor gespeicherte Schlüssel zu verschlüsseln, es wurde daher von dieser Möglichkeit, diese dauerhaft zu speichern, kein Gebrauch gemacht.

Ebenso wird in dem beschriebenen Programmablauf jede zweistufige Übertragung, somit alle Funktionen außer `GetRND32()` und `GetSlot(slotid)`, durch ein aus Beobachtung festgesetztes Warteintervall unterbrochen. Diese kurze Wartezeit zwischen zwei Kommunikationen ist notwendig, damit der Koprozessor den jeweiligen Kommandcode entsprechend verarbeiten kann. Es konnte mehrfach unerwartetes Verhalten beobachtet werden, wenn diese Wartezeit nicht eingehalten wurde, dies äußerte sich in unvollständigen oder falschen Rückgabewerten.

Jegliche Übertragung zwischen Master und Slave findet byteweise statt, jedoch werden für die hier vorgestellten Funktionen Eingaben vom Typ `String` erwartet, da diese sich zum Verwalten für Menschen lesbaren Eingaben einfacher nutzen lassen. Eine alternative Implementierung, welche Bytearrays als übergebene Parameter erwartet, ist jedoch ebenso möglich. Die grundsätzliche Funktionalität - das Auslagern von kryptographischen Berechnungen - konnte gezeigt werden und so wurde auf Grund des begrenzten zeitlichen Rahmens dieser Arbeit davon abgesehen dies ebenfalls umzusetzen.

4.2 Arduino und Atmel AT-SHA204

Der folgende Abschnitt beschreibt die softwareseitige Implementierung der Nutzung des Atmel AT-SHA204 als externen Mikroprozessor zur Berechnung von kryptographischen Funktionen.

Beginnend mit der Erläuterung, wie mit dem Chip kommuniziert wird und welche Funktionalitäten der Chip genau bereitstellt, wird anschließend dargelegt, welche externe Bibliothek verwendet wurde. Eine Vorstellung der in dieser Arbeit implementierten Funktionen wird von einer Beschreibung von beobachteten Problemen und Besonderheiten abgeschlossen.

4.2.1 Softwareentwurf

Als Grundlage für die Beschreibung der Kommunikation mit dem Atmel AT-SHA204 dient die von Atmel bereitgestellte Dokumentation [13]. Beginnend mit der gewählten Art des Kommunikationsweges wird anschließend auf die verschiedenen Konfigurationsmöglichkeiten des Mikrochips eingegangen. Durch die in Abschnitt 4.2.4 näher beschriebenen Probleme im Umgang mit dem Mikrochip wird anschließend genauer der interne Aufbau und der Kommunikationsablauf des Mikrochips erläutert. Diese dienen folgend als Grundlage zur Beschreibung der gewählten Grundkonfiguration, welche vor Beginn jedes Einsatzes als externer Hashfunktionen berechnender Baustein fest auf dem Mikrochip abgespeichert wird.

Bei dem in dieser Arbeit verwendeten Atmel AT-SHA204 handelt es sich, wie schon in Kapitel 2.3 erläutert, um eine sogenannte *SOT23* Ausführung, welche auf einer als *Breakout* bezeichneten Trägerplatine verbaut ist. Diese Ausführung erlaubt aufgrund ihrer Beschränkung auf drei Anschlusspins (*Vin*, *GND* und *SDA*) nur die Kommunikation per SWI, da kein Pin zum Anlegen des Taktsignals der I²C Kommunikation vorhanden ist, wie in Kapitel 2.4 angemerkt.

Wie schon in Kapitel 3.2.1 erläutert, müssen verschiedene Stufen beim Sperren des Mikrochips durchlaufen werden, damit abschließend alle Funktion genutzt werden können. Die Dokumentation des Mikrochips führt diese notwendige Vorgehensweise verteilt auf mehrere Beschreibungen und nur unklar aus (vgl. Abschnitt 4.2.4). Eine verständliche Übersicht der unterschiedlichen notwendigen Schritte sowie eine einheitliche Erläuterung der aus den Sperren folgenden Konsequenzen fehlt. Ebenso ist keine klare Übersicht über Unterschiede verschiedener Arten von Sperren gegeben; so existieren im Folgenden erläuterte Sperren für einzelne Speicherfelder oder ganze Speicherbereiche, sowie Sperren, die Einschränkungen bezüglich der Funktionen, die auf die entsprechenden Daten des Speichers zugreifen, treffen.

Die im Zuge dieser Arbeit erlangten Erkenntnisse, die sich nicht direkt klar verständlich aus der Dokumentation ergeben, aber notwendig sind, um anschließend mittels eines Atmel AT-SHA204 kryptographische Berechnungen durchzuführen, werden im Folgenden dargelegt.

Der Chip wird werkseitig in einem nicht direkt für den produktiven Einsatz verwendbaren Zustand ausgeliefert, die Abbildung 10 stellt dies schematisch dar.

Um den Mikrochip, mit seinen anschließend noch erläuterten Funktionen, verwenden zu können, müssen zwei besondere Sperren aktiviert werden, die einen direkten Einfluss auf die Zugriffsmöglichkeiten zu den verschiedenen Speicherbereichen haben. Eine Übersicht, welche Art von Zugriff auf den jeweiligen Speicherbereich möglich ist, liefert die Tabelle in Abbildung 11.

Der Auslieferungszustand des Mikrochips unterstützt keine Nutzung seiner unterschiedlichen kryptographischen Funktionen; auch rudimentäre Funktionen, wie beispielsweise das Berechnen einer Zufallszahl, werden nicht unterstützt. In diesem Zustand muss der Entwickler zunächst eine für seinen Anwendungsfall passende Konfiguration im als *ConfigZone* bezeichneten Speicherbereich ablegen. Diese Konfiguration

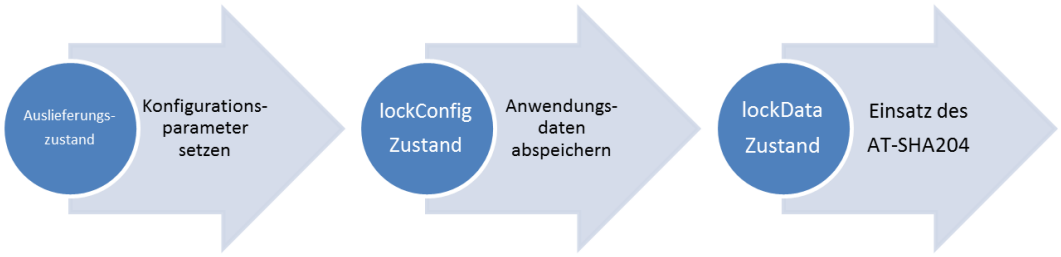

Abbildung 10: *Darstellung der verschiedenen notwendigen Konfigurationsschritte zum Einsatz des Atmel AT-SHA204 Mikrochips.*

Auslieferungszustand		nach *lockConfig*		nach *lockData*	
ConfigZone		**ConfigZone**		**ConfigZone**	
lesen	✓	lesen	✓	lesen	✓
schreiben	✓	schreiben	×	schreiben	×
DataZone		**DataZone***		**DataZone***	
lesen	✓	lesen	✓	lesen	✓
schreiben	×	schreiben	✓	schreiben	×
OTP		**OTP**		**OTP**	
lesen	×	lesen	×	lesen	×
schreiben	×	schreiben	✓	schreiben	×

Abbildung 11: *Tabelle zur Darstellung der verschiedenen Konfigurationsstadien und der daraus resultierenden Zugriffsmöglichkeiten auf die jeweiligen Speicherbereiche. * Der Zugriff unterliegt des Weiteren den in der ConfigZone zuvor hinterlegten Zugriffsberechtigungen.*

umfasst das Festlegen auf unterschiedlichste Parameter, wie beispielsweise die I²C Adresse, welche, bei Verwendung dieses Kommunikationsweges, zur Adressierung eines spezifischen Buselements benötigt wird (vgl. Kapitel 2.4).

Des Weiteren wird durch einen zwei Byte langen Konfigurationswert für jeden Speicherslot, der im als *DataZone* bezeichneten Speicherbereich zur Verfügung steht, eine Zugriffsregelung festgehalten. Die Abbildung 12 zeigt die Zuweisung der einzelnen Bits dieser zwei Byte zu den jeweiligen Parametern und benennt die durch sie konfigurierbaren Zugriffsregelungen. Der durch diese Regeln beeinflusste *Speicherslot* in der *DataZone* umfasst bis zu 256 Byte und kann zusammenhängend oder aber in acht 32 Byte Abschnitten ausgelesen werden. Die durch den Konfigurationswert bestimmte Zugriffsregelung kann das Auslesen der abgespeicherten Daten des jeweiligen Speicherslots komplett oder aber in unterschiedlichen Abstufungen forcieren.

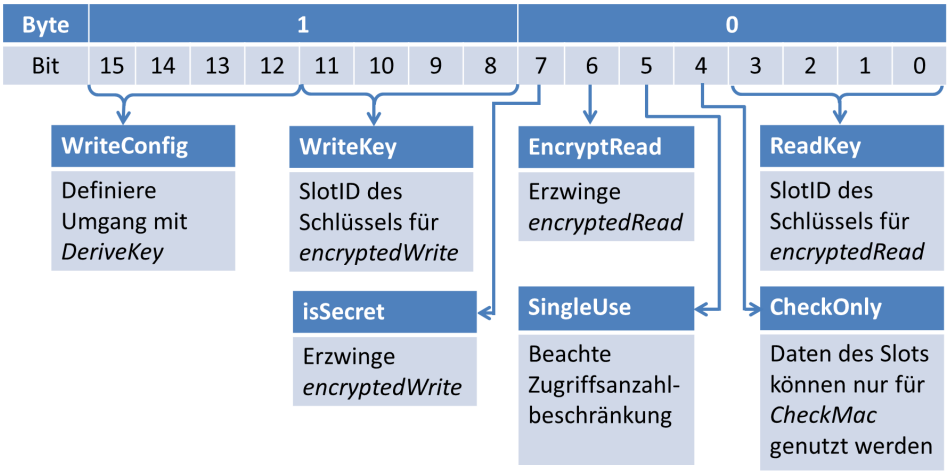

Abbildung 12: *Schematische Darstellung eines Konfigurationsfeldes in der ConfigZone zur Regelung verschiedener Zugriffsparameter eines Speicherslots der DataZone [13].*

Es ist dadurch möglich, einen so abgespeicherten kryptographischen Schlüssel als Grundlage zur Kommunikation bei lesenden Anfragen auf speziell dafür konfigurierte Speicherslots zu erzwingen; dies wird als *encryptRead* bezeichnet. Soll des Weiteren schreibender Zugriff auf einen Speicherslot zwischen einem Hostsystem und einem angeschlossenen Atmel AT-SHA204 verschlüsselt ablaufen, kann ein hierfür spezifizierter Speicherslot den dafür notwendigen Schlüssel bereit halten. Ein so anzusprechender Speicherslot hat in seiner Zugriffskonfiguration das sogenannte *isSecret* Bit gesetzt und ein den kryptographischen Schlüssel bereit haltender Speicherslot wird ebenfalls gesetzt.

Diese eher grundlegenden Zugriffbeschränkungen werden durch eine weitere, durch den eigentlichen Zugriff auf einen Speicherslot beeinflusste, Beschränkung ergänzt. Ist das als *SingleUse* bezeichnete Bit in der Zugriffskonfiguration eines Speicherslots gesetzt, kann das Ausführen kryptographischer Funktionen, welche die Daten dieses Speicherslots in ihren Berechnungen nutzen, beschränkt werden. Diese striktere Zugriffskonfiguration steht jedoch nur für die Speicherslots 0 bis 7 zur Verfügung und es können maximal acht Zugriffe durch zuvor beschriebene Funktionen kontrolliert werden.

Sollte während eines solchen Zugriffs die Spannungsversorgung abbrechen, kann dies unter Umständen jedoch ausreichen, um die verbleibenden Zugriffe auf den Speicherslot zu beeinflussen. Die Dokumentation rät daher, diese Funktionalität explizit nur für das Einschränken auf einen einmaligen Zugriff zu nutzen und die so verbleibenden sieben weiteren Zugriffe als einen Fehlerspielraum anzusehen [13].

Ein weiterer Parameter, der sich speziell auf eine einzelne Funktion des Atmel AT-SHA204 auswirkt ist mittels des *CheckOnly* Bits konfigurierbar. Ist dieses Bit gesetzt, können keine kryptographischen Berechnungen auf Grundlage des diese Konfiguration betreffenden Speicherslots durchgeführt werden. Es ist lediglich möglich, eine zuvor

mittels eines Atmel AT-SHA204 berechnete MAC, die den im betreffenden Speicherslot hinterlegten Schlüssel nutzt, mittels der *CheckMac* Funktion zu prüfen.

Mittels der Bits 12 bis 15 des Konfigurationswertes kann die Verwendung des *Derive-Key* Kommandos genauer kontrolliert werden. Dieses Kommando ermöglicht es dem Entwickler, ein sogenanntes *Rollingcode System* zu nutzen, das in Kapitel 2.3 vorgestellt wurde.

Der dritte Speicherbereich wird als *One Time Programmable Zone* (kurz *OTP*) bezeichnet und verfügt über 16 vier Byte große Speicherslots. Atmel beschreibt diesen Speicherbereich als Möglichkeit statische Daten abzuspeichern, welche nach Beenden des Konfigurationsablaufs nicht mehr manipulierbar sind. Der schreibende Zugriff auf diese Speicherslots ist nur in der zweiten Sperrphase, daher nach Ausführen des *lockConfig* aber vor der Ausführung von *lockData*, möglich. Die in diesen Speicherslots hinterlegten Daten können anschließend als zusätzliche Eingabe bei der Berechnung verschiedener kryptographischer Funktionen genutzt werden.[13]

Eine tabellarische Übersicht über die drei zuvor vorgestellten Speicherbereiche, ihre vorgesehene Verwendung, der jeweilige Zugriff beschränkende Mechanismus sowie Besonderheiten gibt die Abbildung 13.

	DataZone	ConfigZone	OTP
Enthält	- Kryptographische Schlüssel - Anwendungsdaten	- Konfig.parameter (Seriennummer, I²C Adresse, ...) - Zugriffsrechte für jedes Feld der DataZone	- Fixe Anwendungsdaten
Zugriffsmodell	- Zugriffsrechte regulierbar per ConfigZone - *lockData & lockConfig*	*lockConfig*	*lockConfig* & *lockData*
Zugriff	- Lesend nur nach *lockConfig* und bei entsprechenden Zugriffsrechten möglich - Schreibend nur nach *lockConfig* und vor *lockData* möglich	- Lesend immer möglich - Schreibend nur in bestimmten Feldern und nur vor *lockConfig* möglich	- Lesend nur nach *lockConfig* und *lockData* möglich - Schreibend nur nach *lockConfig* und vor *lockData* möglich
Besonderheit	Verschlüsselter Zugriff erzwingbar über entsprechende Rechte in ConfigZone		

Abbildung 13: *Tabelle mit Beschreibung der verschiedenen Speicherbereiche, ihrer Aufgaben, Zugriffsmodelle und Besonderheiten.*

Die in dieser Arbeit gewählte Konfiguration nutzt keine der zuvor beschriebenen Zugriff einschränkenden Konzepte, da sich eine solche Einschränkung nicht aus der Zielsetzung dieser Arbeit ergibt.

Es wurden keine Werte in den OTP Speicherbereich geschrieben, so dass in diesen Speicherslots, wie auch in der Atmel Dokumentation beschrieben [13], alle Bits werkseitig auf 1 gesetzt sind. Die hexadezimale Darstellung dieser Belegung ergibt bei einem vier Byte langen Speicherslot `0xFFFFFFFF`. Der genannte Wert konnte belassen werden, da in dem in dieser Arbeit geplanten Anwendungsfall keine Vorgabe zu statischen Werten, welche in die Berechnung von kryptographischen Funktionen eingehen sollen, gegeben ist.

Es wurde sich dafür entschieden, die 16 Speicherslots der DataZone für das Speichern von Schlüsseln zu nutzen. Daher wurden beim Speicherslot 0 alle Bits und beim Speicherslot 1 kein Bit gesetzt. Die verbleibende Speicherslots wurden mit zufällig generierten Werten beschrieben, die im Anhang dieser Arbeit beigefügt sind. Diese Belegung erlaubt es, unterschiedliche Eingaben als Grundlage für kryptographische Funktionen zu nutzen.

Nach dieser Erläuterung der zur Verfügung stehenden Speicherbereiche und der jeweils gewählten Belegung wird im Folgenden auf Funktionen eingegangen, die der Atmel AT-SHA204 unterstützt. Es wird eine Auswahl von Funktionen genauer beschrieben, sofern diese für die Umsetzung der Arbeit genutzt werden und deren Beschreibung in der Dokumentation ebenfalls zu Teilen unklar beziehungsweise auf unterschiedliche Stellen verteilt erfolgt [13].

Eine schon vorgestellte Funktion ist *CheckMac*, die eine intern berechnete sowie eine übergebene MAC vergleicht und ein einzelnes Bit über die Auswertung zurückgibt. Ebenso genannt wurde *DeriveKey*, welche für die, in dieser Arbeit nicht näher betrachtete, Umsetzung eines *Rollingcode System* genutzt werden kann. Eine als *DevRev* bezeichnete Funktion des Mikrochips gibt indessen die Versionsnummer zurück. *GenDig* und *Nonce* können zum Generieren von bestimmten, sich aus mehreren oder einzelnen internen als auch externen Eingabewerten zusammensetzenden Zwischenergebnissen genutzt werden. Diese können anschließend in die Berechnung verschiedener kryptographischer Funktionen eingeben werden. Für die Umsetzung dieser Arbeit wird jedoch nur die durch *Nonce* angebotene Funktionalität genutzt.

Das *Pause* Kommando erlaubt es dem Entwickler, alle an den gleichen Bus angeschlossenen Atmel AT-SHA204 für die im Anschluss an das *Pause* Kommando folgende Instruktion zu deaktivieren. Ein einzelner Mikrochip, der entsprechend der Parameter des *Pause* Kommandos ausgewählt wird, kann mittels dieses Kommandos für die anstehende Berechnung ausgewählt werden. Die Dokumentation beschreibt die Ausführungsdauer aller zur Verfügung stehenden Funktionen, Ergebnisse können nach einer ebenfalls angegeben maximalen Berechnungszeit über das SWI abgerufen werden [13].

Die erste genauer vorgestellte Funktion des Atmel AT-SHA204 wird mittels der als *Write* Kommando bezeichneten Instruktion aufgerufen und erlaubt das Abspeichern von Daten in verschiedenen Speicherbereichen. Diese Funktion richtet sich hierbei nach den zuvor erläuterten Beschränkungen durch *lockConfig*, *lockData* sowie der entsprechenden Zugriffsbeschränkungen. Es wird nun exemplarisch anhand des für diese Funktion genutzten Kommandos der allgemeine Instruktionsaufbau zum Kommunizieren mit dem Atmel AT-SHA204 gezeigt, welcher in Abbildung 14 dargestellt wird.

Byte	Bezeichnung	Nutzen
0	Kommando-indikator	Kündigt dem Mikrochip ein folgendes Kommando an.
1	Kommandolänge	Länge des Kommandos über alle Felder, außer Kommando-indikator und Checksumme.
2	Operationscode	Spezifiziert die auszuführende Operation.
3	Parameter 1	Funktionsparameter, häufig Auswahl interner Speicherbereiche.
4 – 5	Parameter 2	Weitere Funktionsparameter, häufig spezifische Auswahl von Speicherslots.
6 – N	Daten	Übertragung von externen Daten.
N + 1 – N + 2	Prüfsumme	Prüfsumme über die gesamte Kommandoinstruktion ohne Kommandoindikator.

Abbildung 14: *Tabelle zur Darstellung des Aufbaus eines an den Atmel AT-SHA204 übertragenen Kommandos mit Benennung der verschiedenen Kommandoparameter und ihrer Bedeutung. Entworfen nach [13].*

Im Allgemeinen setzt sich eine Instruktion aus mindestens fünf Byte zusammen, eingeleitet von einem mit 0 gesetzten Bit, das ein folgendes Kommando ankündigt. Das zweite Byte dient der Übermittlung der Instruktionslänge. Es folgt ein weiteres einzelnes Byte, welches die durchzuführende Funktion spezifiziert; für das *Write* Kommando ist dies 0x12. Das dritte Byte wird genutzt, um funktionsspezifische Parameter anzugeben. Im betrachteten Fall kann zwischen den verschiedenen Speicherbereichen gewählt werden. Ein einzelnes Bit veranlasst den Mikrochip, vier oder 32 Byte lange Abschnitte zu schreiben, ein weiteres Bit spezifiziert, ob dies ein *encryptWrite* Vorgang ist. Die zwei folgenden Bytes werden zur Angabe des Speicherslots genutzt, bei anderen Operationen können sie für weitere Parameter gebraucht werden. Gefolgt von den für die jeweilige Funktion notwendigen Parametern werden anschließend Daten zu Verarbeitung an den eigentlichen Kommandokopf angehängt, beim hier vorgestellten *Write* Kommando folgt auf die eigentlich abzuspeichernden Daten ein diese Daten sowie die Speicheradresse authentifizierender MAC. Jedes Kommando wird von einer über alle Kommandofelder berechneten Prüfsumme finalisiert, die es dem Mikrochip erlaubt Fehler in der Datenübertragung festzustellen. Konnte der Atmel AT-SHA204 das *Write* Kommando erfolgreich bearbeiten, gibt er ein einzelnes Null Byte zurück.

Als Gegenstück zur schreibenden Funktionalität des *Write* Kommandos kann ein Entwickler das *Read* Kommando zum Auslesen von Speicherbereichen nutzen. Die Auswahl des Speicherbereiches und des eindeutigen Speicherslots erfolgt auf gleiche Weise wie der des zuvor erläuterten *Write* Kommandos.

Ein weiteres Kommando ist das für die beschriebenen Konfigurationschritte notwendige *Lock* Kommando, das über einen entsprechend gewählten Parameter die Data- oder

ConfigZone sperrt. Ein weiterer Parameter erlaubt das Deaktivieren einer andernfalls notwendigen Prüfsumme über alle Speicherslots des zu sperrenden Bereiches. Die Auswahl des Speicherbereiches sowie der Rückgabewert bei erfolgreichem Sperren sind hierbei ebenfalls identisch mit denen des *Write* Kommandos.

Neben diesen für Verwaltung und Konfiguration notwendigen Funktionen unterstützt der Atmel AT-SHA204 auch das Berechnen einer Zufallszahl. Dies wird mittels des *Random* Kommandos initiiert. Der Entwickler kann bei jedem Aufruf per Parameter entscheiden, ob ein Startwert (engl. *seed*), aus dem eine Zufallszahl abgeleitet wird, neu gesetzt oder beibehalten werden soll.

Des Weiteren gibt es eine per als *MAC* Kommando bezeichnete Funktionalität: Diese berechnet eine SHA256 Hashsumme auf einer Kombination von Eingabe und im Speicher hinterlegten Werten. Die Eingabe kann hierbei auch leer sein. Es fließen stets fest vorgegebene Daten, wie beispielsweise Teile der Seriennummer des Mikrochips, das die Instruktion identifizierende Byte und das die Parameter spezifizierende Byte des übertragenen Kommandos mit in die Berechnung ein. Ebenso gehen jeweils Daten oder aus diesen abgeleiteten Daten eines frei wählbaren Speicherslots der DataZone mit in die Berechnung mit ein.

Es ist ebenfalls möglich, statische Daten aus der OTP mit einfließen zu lassen; ist dies nicht erwünscht, werden stattdessen entsprechend Null Bytes in den Eingangswert der Berechnung eingefügt. Wie genau die Daten intern verknüpft werden, wird in der von Atmel bereitgestellten Dokumentation nicht näher erläutert [13]. Der Rückgabewert ist ein, der Blocklänge des SHA256 Hashalgorithmus entsprechend, 32 Byte langer Wert.

Explizit für die auf diese Art und Weise berechneten Hashsummen bietet der Atmel AT-SHA204 eine per *CheckMac* Kommando aufrufbare Funktion. Als Eingabe werden die beim Aufruf des *MAC* Kommandos verwendeten Parameter und Eingabewerte verwendet, sowie der daraus resultierte Hashwert.
Der Mikrochip vergleicht intern den nach einem *MAC* Kommando mit diesen Eingabewerten generierten Hashwerts mit dem übergeben Hashwert; stimmen beide überein, bestätigt er dies durch Rückgabe eines Null Bytes.

Alternativ zum *MAC* Kommando kann ebenso das *HMAC* Kommando genutzt werden. Dieses wird mit einem identischen Parameteraufbau wie auch das *MAC* Kommando aufgerufen. Der Unterschied zwischen den Parametern besteht hierbei nur darin, dass externe Daten zuvor per *Nonce* Kommando auf den Chip übertragen werden, anstatt ein Teil der Kommandoinstruktion zu sein. Die auf diese Weise übertragenen Daten dürfen maximal 32 Byte lang sein [13]. Weitere fixe Eingabewerte liefern dieselben Felder, wie bereits beim *MAC* Kommando beschrieben. Die Kombination aus variabel wählbaren externen und fixen internen Daten wird dann, im Unterschied zum *MAC* Kommando, jedoch in der anschließende Berechnung einer HMAC nach FIPS-198 Standard genutzt [13] [62]. Wie auch beim *MAC* Kommando gibt die Dokumentation keinen Aufschluss über die Art und Weise, wie diese Daten kombiniert werden.

Durch die Besonderheit der zusammengesetzten Eingabe für die für diese Arbeit relevanten Funktionen *MAC* und *HMAC*, wie auch im Abschnitt 4.2.4 näher erläutert, muss sich darauf beschränkt werden, Nachrichten, die in einem System, welches den

Atmel AT-SHA204 als Koprozessor zur Authentifizierung nutzt, nur in Kombination mit einem identisch konfigurierten System authentifizierbare Nachrichten austauschen zu lassen. Die Dokumentation erläutert dies nicht. Diese Einschränkung führt zu einer eingeschränkten Verwendbarkeit mit schon existierenden Systemen, die möglicherweise einen Lösungsansatz, wie er in Kapitel 4.1 vorgestellt wird, nutzen.

4.2.2 Verwendete Bibliotheken

Als Grundlage zur Kommunikation mit dem Atmel AT-SHA204 wurde eine von Jim Lindblom auf GitHub veröffentlichte Bibliothek verwendet [31]. Diese bietet insbesondere durch die in ihr definierte *execute* Funktion eine Möglichkeit, den wie in Abschnitt 4.2.1 beschriebenen Instruktionsaufbau an den Mikrochip zu übertragen. Die Bibliothek ermittelt für eine erstellte Instruktion automatisch die entprechende Prüfsumme und nutzt so die vom Mikrochip unterstützte Funktionalität, um Übertragungsfehler zu erkennen.

Es sind neben der *execute* Funktion nur vier weitere Funktionalitäten des Chips direkte durch die Bibliothek nutzbar, diese sind das Generieren einer Zufallszahl, das Ausgeben der Seriennummer sowie das Auslesen der vorliegenden Hardwareversion des aktuell angeschlossenen Mikrochips. Eine implementierte *read* Funktion unterstützt nur das Auslesen von jeweils 32 Byte aus den Speicherbereichen des Chips, welche zu Teilen auch in 4 Byte großen Blöcken auslesbar sind. Eine explizite Funktion zum beschreiben einzelner Speicherbereiche des Chips existiert nicht.

4.2.3 Implementierung

Im Folgenden werden die in dieser Arbeit implementierten Funktionen erläutert, die es ermöglichen, den Atmel AT-SHA204 Mikrochip als externes System zur Berechnung von kryptographischen Funktionen zu nutzen.

Die in dieser Arbeit erstellten Funktionen erlauben es, den Mikrochip zu konfigurieren und ihn mittels der verschiedenen, in Abschnitt 4.2.1 erläuterten, notwendigen Sperren in einen für die Verwendung geeigneten Zustand zu bringen. Die Beschreibung der implementierten Funktionen folgt der in Abbildung 10 im Abschnitt 4.2.1 dargestellten notwendigen Vorgehensweise zur Konfiguration eines Atmel AT-SHA204. Die in Abschnitt 4.2.2 vorgestellte Bibliothek von Jim Lindblom übernimmt insbesondere den direkten Austausch von Daten zwischen Mikrochip und Arduino Mikrocontroller.

Für die Konfiguration des Mikrochips wurde eine grundsätzliche Funktion implementiert, die es erlaubt, die in Abschnitt 4.2.1 erläuterten Konfigurationsfelder der *ConfigZone* zu beschreiben. Diese Funktion unterstützt das Beschreiben der einzelnen Speicherfelder der *ConfigZone* mit acht Byte langen Konfigurationswerten. Wird ein zuvor schon gesetzter Konfigurationsfeld erneut zum Speichern eines Konfigurationswertes angegeben, überschreibt dies den zuvor enthaltenen Wert.
Es wurde darauf geachtet, dass die Adressierung der Speicherfelder konsistent mit der in der Dokumentation des Mikrochips gegebenen Übersicht der *ConfigZone* ist [13].

Wie zuvor in Abschnitt 4.2.1 bereits erläutert, wurde die Entscheidung getroffen keine der unterstützten Zugriff einschränkenden Konzepte für die Speicherslots der *DataZone* zu nutzen. Für die Konfiguration der in dieser Arbeit verwendeten Mirkochips wurde eine, die zuvor erläuterte Funktion nutzende, Methode implementiert, die alle Konfigurationsfelder mit entsprechenden Konfigurationswerten (0x00000000) setzt.

Konsistent zum Setzen der gewünschten Konfiguration wurde eine Funktion implementiert die, nach gleicher Art der Adressierung, auch einzelne Konfigurationswerte auslesen kann. Dies kann insbesondere dazu genutzt werden, um vor dem Sperren der *ConfigZone* die abgespeicherte Konfiguration zu prüfen, da, wie in Abschnitt 4.2.1 erläutert, nachträgliche Änderungen nicht möglich sind.

Nach erfolgreichem Übertragen der Konfiguration kann die *ConfigZone* des Mikrochip gesperrt werden; zu diesem Zweck wurde eine explizite Funktion, `lockConfigZone`, implementiert. Es wurde sich dagegen entschieden, eine allgemeine parametrisierte Sperrfunktion zu verwenden, die ebenfalls auch das Sperren der *DataZone* unterstützt. Eine solche Funktion birgt die Gefahr, dass ein einzelner falsch gesetzter Parameter den verwendeten Mikrochip permanent sperrt und dieser so in einem permenenten falsch konfiguriertem Zustand ist, da keine der beiden Sperren umkehrbar ist. Eine explizite Sperrfunktion unterstützt darin den Mikrochip nur absolut bewusst in den nächsten, in Abbildung 10 dargestellten Zustand zu bringen.

Zum Prüfen, ob ein Sperren der *ConfigZone* erfolgreich war oder der Mikrochip noch mit weiteren Konfigurationswerten bespielt werden kann, wurde eine entsprechende Funktion implementiert. Erfolgreich war das Sperren der *ConfigZone*, wenn im Speicherfeld 87 der *ConfigZone* nicht mehr der werkseitige Wert 0x55 gesetzt ist [13]. Die Funktion prüft, ob diese Speicherstelle den Wert 0x55 enthält; ist dies der Fall wird der Mikrochip als nicht gesperrt gemeldet. Wird der Wert 0x00 ausgelesen, wird ein korrekt gesperrter Chip gemeldet. Unterscheidet sich der ausgelesen Wert von den beiden zuvor beschrieben Fällen, wird ein gesperrter aber möglicherweise defekter Chip gemeldet. Eine Erklärung zu dieser Einschätzung findet sich im folgenden Abschnitt 4.2.4.

Nach dem erfolgreichen Abspeichern der Konfiguration und dem Überführen des Mikrochips in einen Zustand mit gesperrter *ConfigZone*, kann damit begonnen werden Anwendungsdaten in den Speicherbreich der *DataZone* zu schreiben. Zu diesem Zweck wurde eine Funktion implementiert, die in vier Byte oder 32 Byte Abschnitten Daten in den Speicherbereich schreibt. Die Adressierung der Speicheradressen der *DataZone* erfolgt mittels der in der Dokumentation verwendeten SlotID 4.2.1.

Das Auslesen der Werte der *DataZone* ist mittels der implementierten `readDataZone` Funktion, die als Parameter die entsprechende SlotID und einen Speicherbereich für die ausgelesenen Daten erwartet möglich. Der in den angegeben Speicherbreich zurückgegebene Wert enthält die kompletten 32 Byte eines gewählten Slotspeicherplatzes.

Sollte das Schreiben oder Auslesen eines Speicherslots, auf Grund von gewählten Zugriffbeschränkungen für den gewählten Speicherslot, nicht möglich sein, geben die vorgestellten Funktionen einen entsprechenden Fehlercode zurück.

Sind alle gewünschten Werte in der *DataZone* hinterlegt, kann diese mit der implementierten `lockDataZone` Methode gesperrt und der Chip somit in einen, für den Einsatz als externe Möglichkeit zur Berechnung von kryptographischen Funktionen nutzbaren, Zustand gebracht werden.

Befindet sich der Mikrochip in dem zuvor beschriebenen Zustand, kann dieser zur Berechnung der in Abschnitt 4.2.1 erläuterten *MAC* und *HMAC* Funktionalität genutzt werden.

Mittels der implementierten Funktion `calcMAC`, die eine SlotID sowie eine maximal 32 Byte langen Eingabewert erwartet, wird eine wie in Abschnitt 4.2.1 dargelegte SHA256 Hashsummenberechnung durchgeführt. Ein ebenfalls beim Funktionsaufruf angegebener Speicherbereich enthält die dann berechnete 32 lange Hashsumme.

Die zuvor genannte Möglichkeit zur Berechnung von *HMACs* kann durch die implementierte Funktion `calcHMAC` genutzt werden. Die erwarteten Parameter sind ebenfalls eine maximal 32 Byte lange Eingabe, die SlotID des zu verwendenen Schlüssel sowie ein Speicherbreich für den berechneten HMAC. Die Funktion nutzt das erläuterte *Nonce* Kommando um die übergebenen 32 Byte für den anschließenden Aufruf der *HMAC* Berechnung zu übertragen. Beide Funktionen geben, bei durch den Chip gemeldeten Fehlern in der Berechnung, einen Fehlercode zurück.

4.2.4 Besonderheiten und Probleme

Bei der Verwendung des Atmel AT-SHA204 wurde sich Eingangs insbesondere mit der vom Hersteller bereitgestellten Dokumentation auseinandergesetzt [13]. Diese ist in ihrem Aufbau und der Beschreibung mehrerer Eigenschaften und Funktionen des Mikrochips unklar.

Bei Angaben zu Speicheradressen zur Adressierung von Lese- und Schreibzugriffen ist eine missverständliche Darstellung gegeben, die teils falsche Adressen angibt. Auch die in dieser Arbeit deutlich gemachte Vorgehensweise zur initialen Konfiguration eines AT-SHA204 Mikrochips ist nicht klar verständlich beschrieben, so wird die Beschreibung der einzelnen Sperren an unterschiedlichen Stellen der Dokumentation vorgenommen. Eine zusammenfassende Ausführung über die Abhängigkeit der einzelnen Sperren von einander und ihre Auswirkungen auf die unterschiedlichen Lese- sowie Schreibrechte fehlt ebenso.

Beim Setzen der in Kapitel 4.1 beschriebenen Sperren wird in der Dokumentation spezifiziert, dass ein entsprechendes anschließend ausgelesenes Speicherfelder der ConfigZone einen sich vom ursprünglichen Auslieferungszustand mit $0x55$ gesetzten Wert unterscheidet. Eine genaue Erläuterung, welchen Wert das entsprechende Speicherfeld nach einem erfolgreichen Sperren haben sollte, fehlt. Im Zuge der Konfiguration von mehreren Mikrochips zeigte sich, dass bei anschließend funktionstüchtiger Konfiguration ein Speicherwert mit $0x00$ in der entsprechenden Speicherzelle vorhanden war.

Bei einem, mit einer auf zwei anderen Chips funktionstüchtigen Konfiguration, bespielten Mikrochip konnte nach dem Setzen der entsprechenden Konfigurationssperre ein Wert von $0x47$ ausgelesen werden. Ein direkter Grund für dieses abweichende Verhalten ließ sich nicht erkennen, es stellte sich jedoch heraus, dass dieser Mikrochip sich anschließend nicht für die Berechnung von kryptographischen Funktionen nutzen ließ. Dies legt die Vermutung eines Hardwaredefekts dieses einzelnen Mikrochips nahe.

Diese genannten Unklarheiten führten im Zuge dieser Arbeit dazu, dass zwei Mikrochips nun keiner Verwendung im Sinne dieser Arbeit mehr zugeführt werden können. Sie zeigen nur noch nicht dokumentiertes Verhalten, so werden Hashanfragen, die von einem funktionstüchtigen Mikrochip nachvollziehbar sinnvoll beantwortet werden, mit ständig wechselnden Ergebnissen beantwortet.

Zusätzlich zu den beschriebenen Problemen, welche sich aus teilweise Uneindeutigkeiten der vorhanden Dokumentation sowie der bei der Konfiguration aufgetretenen Probleme ergeben, schränkt die grundsätzlichen Funktionsweise des Atmel AT-SHA204 seine Verwendbarkeit als externer Koprozessor zur Berechnung kryptographischer Funktionen signifikant ein.

Die vom Mikrochip angebotenen Funktionen *MAC* und *HMAC* berechnen, entsprechend der genannten Standards aus Kapitel 2.1, Hashsummen zur Authentifizierung von Nachrichten. Diese Berechnung basiert aber zu Teilen auf nicht vollständig beeinflussbaren Eingabedaten; so fließt, wie in Kapitel 4.2.1 beschrieben, ein nicht geringer Anteil an für den verwendeten Mikrochip spezifischen Daten in diese Berechnung ein.

Ebenfalls undokumentiert ist die Art und Weise, wie diese Daten mit der vom Anwender übergebenen Nachricht und dem gegebenfalls verwendeten Schlüssel kombiniert werden. Es ist somit insbesondere einem Entwickler nicht möglich entsprechende Berechnungen nachzuvollziehen. Dieses nachvollziehen zu können ist jedoch notwendig, falls ein entworfenes System eine Nachricht erhält, die zur Authentifizierung von einem Atmel AT-SHA204 berechneten HMAC begleitetet wird. Zum Prüfen und Bestätigung eines solchen MAC/HMAC bedarf es daher zwingend eines weiteren identisch konfigurierten Atmel AT-SHA204 Mikrochips. Denn nur mittels der gleichen internen Berechnungen, basierend auf identisch gewähltem Schlüssen und der empfangen Nachricht, ist es möglich einen Hashwert zu berechnen, der dann zum Vergleich mit dem empfangenen Hashwert genutzt werden kann.

5 Evaluation

Im Folgenden Kapitel wird näher beschrieben, wie die bei der Implementierung der zuvor beschriebenen Lösungen zur Realisierung externer kryptographischer Funktionen für Arduino bewertet werden können. Beginnend mit einer Erläuterung zur Bewertung der Realisierung eines als Koprozessors genutzten weiteren Arduino Mega2560, wird anschließend die Verwendbarkeit des basierend auf einem Atmel AT-SHA204 entworfenen Systems für die beschriebene Aufgabe näher dargelegt.

5.1 Ardunio als Koprozessor

Die mit dem Koprozessor umgesetzte Funktionalität erlaubt es HMACs nach dem in FIPS180 definierten Standard zu generieren. Ebenso ist es möglich Hashsummen nach dem SHA256 Standard, definiert in [1], berechnen zu lassen.

Die korrekte Berechnung der SHA256 Hashsummen wurde durch das Testen der in dieser Arbeit erarbeiteten Funktionen mit den in [25] beschrieben Testwerten durchgeführt. Die im Standard beschriebenen Ergebnisse stimmen mit den durch den Koprozessor berechnenten Ergebnissen überein. Eine Ausnahme stellt der zuletzt im Standard beschriebene Test dar, dieser beschreibt die Berechnung einer SHA256 Hashsumme über eine Eingabe bestehend aus 1.000.000 mal dem Buchstaben "**a**". Diese Eingabe wurde auf die durch den Koprozessor maximal unterstützte Länge von 255 Zeichen gekürzt, wie in Kapitel 4.1.3 erläutert. Das so erhaltene Ergebnis wurde mit dem Ergebnis verglichen, welches durch das Kommandozeilenprogramm `SHA256sum` berechnet wird. Dieses Programm ist Teil der für die Erstellung dieser Arbeit verwendeten CentOS 6.5 Umgebung und wurde seinerseits entsprechend des FIPS180 Standard entwickelt und getestet [63].

Die in [49] beschriebenen Testeingaben für Schlüssel sowie Nachricht wurden genutzt, um die berechneten HMAC Hashsummen des entwickelten Koprozessor zu testen. Ein erster Testdurchlauf speicherte den beschrieben Schlüssel in den ersten Datenslot des Koprozessors ab und rief als nächstes die in Kapitel 4.1.3 beschriebene HMAC Funktion entsprechend auf. Ein zweiter Test nutzte die Möglichkeit einen Schlüssel nur für eine einzelne HMAC Berechnung zu definieren, die Berechnung erfolgte mittels der implementierten Funktion `HMACTempKey(msg)`. Bei beiden Testdurchläufen stimmten die vom Koprozessor berechneten Hashsummen mit den im Dokument beschriebenen `HMAC-SHA-256` bezeichneten Hashsummen überein.

Es wurde zur kurzen und effizienten Erstellung der beschriebenen HMAC Hashsummen eine Möglichkeit geschaffen entsprechend regelmäßig benötigte Schlüssel auf dem Koprozessor abzuspeichern. Dies erlaubt es, in einem produktiven Einsatz auf das regelmäßige Übertragen dieser redundanten Daten zu verzichten. Es ist möglich bis zu acht solcher maximal 255 Zeichen langer regelmäßig genutzter Schlüssel abzuspeichern. Verglichen mit dem alternativen Übertragen von zuerst einem 255 Zeichen langen Schlüssel sowie anschließend einer 255 Zeichen langen zu authentifizierenden Nachricht kann, durch das vorherige Abspeichern eines Schlüssels, die Kommunikation für alle dann folgenden Authentifizerungen fast halbiert werden. Dies ergibt sich, da nur noch die einzelne Nachricht begleitet mit der entsprechenden Instruktion, welcher abgespeicherte Schlüssel verwendet werden soll, notwendig ist.

Durch Modifikationen von existierenden genutzten Bibliotheken konnte die zuvor existierende Übertragungslänge einer einzelnen Nachricht von 32 fast verachtfacht werden, beschrieben in Kapitel 4.1.4. Das in dieser Arbeit entworfenen System beschränkt sich auf eine maximale Eingabelänge von 255 Zeichen, die mit dem externen Koprozessor verarbeitet werden können. Solch eine Nachrichtenlänge reicht insbesondere für die im Projekt ARINA beschrieben Kommunikation zur Übertragung von Infrarotsignalen zwischen zwei Arduino Mikrocontrollern aus [43].

Der entworfene Ablauf zur Kommunikation mit dem Koprozessor erlaubt es, zeitkritische Aufgaben des als Master agierenden Arduinos weiterhin durchzuführen. Der durch ein berechnetes Ergebnis des Koprozessors ausgelöste Interrupt unterbricht die Abarbeitung anderer Programmabschnitte des Masters in minimalem Umfang, wie in Kapitel 4.1.1 beschrieben. Diese gewählte Variante erlaubt es, permanente Anfragen, welche durch die Wahl eines Round Robin Verfahrens häufig stattfinden würden, zu vermeiden und so mögliche zeitkritische Aufgaben mit nur seltenen sehr kurzen Interrupts bearbeiten zu können. Die Kommunikation zwischen Master und Slave findet dazu in nicht für zeitkritische Aufgaben genutzten Zeiträumen statt.

Neben der beschriebenen Zielsetzung, Nachrichten authentifizieren zu können, wurde des Weiteren bedacht, dass weitere externe Komponenten Teil eines zukünftigen Systemaufbaus sein können. Die gewählte Kommunikationsvariante I²C ist ein weit verbreiteter Standard zur Kommunikation zwischen Mikrocontrollern und externen Komponenten, siehe hierzu Kapitel 2.4. Sie ermöglicht das in dieser Arbeit entworfene System in schon einen existierenden I²C Bus einzubinden.

Für die Verwendung als so genannter *Salt* (eng. Salz), der mit einem Klartext verknüpft werden kann, unterstützt der Koprozessor auch die Generierung von Zufallszahlen. Dies kann zur Erhöhung der Entropie von den aus dieser Verknüpfung berechneten Hashwerten genutzt werden.

5.2 Arduino und Atmel AT-SHA204

Zur Evaluation des auf dem Atmel AT-SHA204 basierenden Systems wurde der im Folgenden beschriebene Versuch durchgeführt.

Es wurden zwei Mikrochips mit einer identischen Konfiguration, wie in Abschnitt 4.2.1 beschrieben, konfiguriert. Der im vorherigen Kapitel beschriebenen Vorgehensweise zur Konfiguration folgend, wurden beide Mikrochips mittels der Funktion `lockConfigZone` in den nächsten Zustand überführt. Nach erfolgreichem Sperren der *ConfigZone* wurden die im Anhang ersichtlichen, zufällig generierten Werte in die jeweiligen Datenslots übertragen. Anschließend wurden die Mikrochips durch ein Sperren des Datenbereiches, mittels der Funktion `lockDataZone`, in einen für Berechnungen bereiten Zustand überführt.

Aufgerufen mit der gleichen 32 Byte langen Eingabe und bei Auswahl des gleichen internen Speicherslots hat die in Abschnitt 4.2.3 vorgestellten Funktion `calcMAC` auf beiden Mikrochips einen identischen Hashwert berechnet. Abbildung 15 zeigt die verwendete Eingabe und den berechneten Hashwert beider Mikrochips.

```
1  Input:
2  0x00 0x11 0x22 0x33 0x44 0x55 0x66 0x77
3  0x88 0x99 0xAA 0xBB 0xCC 0xDD 0xEE 0xFF
4  0x00 0x11 0x22 0x33 0x44 0x55 0x66 0x77
5  0x88 0x99 0xAA 0xBB 0xCC 0xDD 0xEE 0xFF
6
7  Result:
8  0x23 0x06 0x67 0x00 0x4F 0x28 0x4D 0x6E
9  0x98 0x62 0x04 0xF4 0x60 0xA3 0xE8 0x75
10 0x8A 0x59 0x85 0xA6 0x79 0x96 0xC4 0x8A
11 0x88 0x46 0x43 0x4E 0xB3 0xDB 0x58 0xA4
12 0xFB 0xE5 0x73
```

Abbildung 15: *Darstellung des für Testzwecke genutzten 32 Byte langen Eingabewertes sowie des jeweils durch `calcMAC` berechneten Hashwertes.*

Durch den beschriebenen Versuch konnte gezeigt werden, dass wenn eine Nachricht von einem mittels `calcMAC` berechneten Hashwert begleitet wird, es durch zwei identisch konfigurierte Atmel AT-SHA204 möglich ist, die Authentizität dieser Nachricht zu überprüfen.

6 Zusammenfassung

In dieser Bachelorarbeit wurden zwei Ansätze zur externen Berechnung von kryptographischen Funktionen unter Verwendung eines Arduino Mikrocontrollers realisiert.

Das in Kapitel 4.1 vorgestellte System erlaubt es, einen als Koprozessor programmierten Arduino 2560 Mikrocontroller SHA256 Hashsummen sowie HMACs über bis zu 255 Byte lange Nachrichten zu berechnen.

Die in dieser Arbeit getroffenen Entscheidungen zur Wahl des genutzten Kommunikationsweges zwischen zwei Arduino Mikrocontrollern wurden ebenso präsentiert, wie das entworfene System zum Übertragen von Anweisungen und Berechneten Hashsummen. Das entworfene und implementierte Softwaresystem zur Abarbeitung empfangener Kommandos wurde erklärt und die gewählte Art deren Bearbeitung erläutert. Der beschriebene Hardwareaufbau sowie die zur Kommunikation und Berechnung implementierten Funktionen ermöglichen es auch einem auf dem Gebiet der Mikrocontrollerprogrammierung unerfahrenen Leser, das entworfene System zu replizieren.

Ein in der Evaluation vorgestellter Versuch zeigt, dass das realisierte System den Anforderungen der in FIPS180 definierten Standards an HMACs sowie der in RFC4231 beschriebenen Tests für die Berechnung von SHA256 Summen genügt [25][49].

Ein zweiter in dieser Arbeit realisierte Lösungansatz verwendet einen Atmel AT-SHA204 Mikrochip zur Berechnung nach den in FIPS198 definierten Standard von HMACs [62]. Die Aufarbeitung der Dokumentation zur Verwendung des Mikrochips hat zur Implementierung von mehreren, in Abschnitt 4.2.3 vorgestellten Funktionen geführt. Diese Funktionen erlauben es, die initial notwendigen Konfigurationsschritte, beschrieben in Abschnitt 4.2.1, effizient durchzuführen und die bereit gestellten kryptographischen Funktionalitäten zu nutzen.

Die während der Ausarbeitung dieser Arbeit festgestellten Probleme, teilweise bedingt durch eine unklare Dokumentation des verwendeten Mikrochips, wurden vorgestellt und aufgearbeitet. Die aufgearbeitete Beschreibung der notwendigen Schritte zur Konfiguration des Mikrochips sowie der unterschiedlichen Bedeutungen von Sperren und Zugriffsbeschränkungen, erlaubt es einem unvertrauten Leser sich zügig mir der neuen Materie auseinanderzusetzen.

Der in der Evaluation vorgestellte Versuch zeigt, dass sich zwei identisch konfigurierte Atmel AT-SHA204 dazu eignen, eine Nachricht, durch Verwenden der kryptographischen Funktion dieser Mikrochips, zu authentifizieren.

Die beiden realisierten Systeme lassen sich auch nach Abschluss dieser Bachelorarbeit erweitern und verbessern. Eine direkte Verbesserung würde das Erweitern der aktuell maximal 255 Byte Zeichenbegrenzung bei der Kommunikation mit dem realisierten Arduino Koprozessor beziehungsweise der 32 Byte des Atmel AT-SHA204 darstellen. Zur Umsetzung einer nicht längenbegrenzten Kommunikation des Koprozessors, würde ein Kommunikationsprotokoll entworfen und der aktuelle Nachrichtenaustausch erweitert werden müssen. Das Kommunikationsprotokoll müsste, unabhängig von der maximalen einzelnen Nachrichtenlänge, einer Übertragung die Eingabe über mehrere Übertragungen aufteilen und anschließend die gesamte Eingabe zur Berechnung übergeben. Ein anspruchsvolleres aber eleganteres Vorgehen wäre es, einzelne übertragene Daten in die begonnene Berechnung der Hashsumme mit einfließen zu lassen und das intern vorgehaltene Ergebnis laufend zu aktualisieren. Ein solches Vorgehen erfordert zwar detailreiches Wissen zu den mathematischen Abläufen einer Hashsummenberechnung, bietet jedoch den Vorteil nicht vom begrenzten vorhandenen Arbeitsspeicher eines Arduino Mikrocontrollers beschränkt zu sein.

Ein Austausch des verwendeten Atmel AT-SHA204 durch seinen abwärtskompatiblen Nachfolger Atmel AT-SHA204A erlaubt es, das auf einem Mikrochip basierende System weiter zu nutzen. Im Gegensatz zu dem in dieser Arbeit verwendeten Vorgänger bietet der erst kurz vor Ende der Bearbeitungszeit dieser Bachelorarbeit, vorgestellte AT-SHA204A die Möglichkeit, SHA256 Hashsummen auf ausschließlich der übergebenen Eingabe zu berechnen (vgl. hier zu Kapitel 4.2.4). Diese Berechnung ist zudem in ihrer Eingabelänge explizit nicht begrenzt. Mit geringem Aufwand ließen sich damit die schon in dieser Arbeit beschriebenen kryptographischen Funktionen sinnvoll erweitern, da es nun möglich wäre, mittels des Mikrochips über vollständig beeinflussbare Eingaben beliebiger Länge SHA256 Hashsummen zu berechnen.[12]

Diese Arbeit konzentriert sich auf die Realisierung von HMACs und Hashsummen Berechnungen basierend auf dem SHA256 Algorithmus. Die vorgestellten allgemeinen Spezifikationen eines MAC und einer HMAC lassen jedoch auch weitere Algorithmen zur Authentifizierung von Nachrichten zu. Der realisierte Koprozessor als auch das mit einem Atmel AT-SHA204 arbeitende entworfene und implementierte System ließen sich um solche alternativen Algorithmen erweitern.

In der als *Atmel CryptoAuthentication* bezeichneten Modellreihe werden ebenfalls Mikrochips zur Berechnung nach *Elliptic Curve Digital Signature Algorithm* (ECDSA) sowie *Advanced Encryption Standard* (AES) angeboten [14]. Auch zur direkten Berechnung anderer Algorithmen auf einem Mikrocontroller existieren kryptographische Funktionen, die das schon bereits existierende System ergänzen können [15].

A Literatur

[1] D. Eastlake 3rd und T. Hansen. *US Secure Hash Algorithms (SHA and HMAC-SHA)*. http://tools.ietf.org/html/rfc4634; abgerufen am 20. August 2014. Juli 2006.

[2] Georg Becke et al. *Texas Instruments, Comparing Bus Solutions*. http://www.lammertbies.nl/download/ti-appl-slla067a.pdf; abgerufen am 10. August 2014. Feb. 2014.

[3] *Apple iPhone 5s Technische Daten*. https://www.apple.com/de/iphone-5s/specs/; abgerufen am 10. September 2014. Sep. 2014.

[4] *Arduino - Serial*. http://arduino.cc/en/pmwiki.php?n=Reference/Serial; abgerufen am 10. September 2014. Sep. 2014.

[5] *Arduino - Serial - begin()*. http://arduino.cc/en/Serial/Begin; abgerufen am 03. September 2014. Sep. 2014.

[6] *Arduino Ethernet Shield*. http://arduino.cc/en/Main/ArduinoEthernetShield; abgerufen am 16. September 2014. Sep. 2014.

[7] *Arduino Mega 2560*. http://arduino.cc/de/pmwiki.php?n=Main/ArduinoBoardMega2560; abgerufen am 10. September 2014. Sep. 2014.

[8] *Arduino.cc , Wire Bibliothek*. http://arduino.cc/en/pmwiki.php?n=Reference/Wire; abgerufen am 11. August 2014. Apr. 2014.

[9] *Arduino.cc, Arduino Mega 2560*. http://arduino.cc/en/Main/ArduinoBoardMega2560; abgerufen am 06. September 2014. Sep. 2014.

[10] *Arduino.cc, The Arduino Playground*. http://playground.arduino.cc/; abgerufen am 10. September 2014. Sep. 2014.

[11] *Arduiono.cc, attachInterrupt()*. http://arduino.cc/en/Reference/AttachInterrupt; abgerufen am 01. September 2014. Sep. 2014.

[12] *Atmel ATSHA204A, Datasheet*. http://www.atmel.com/Images/Atmel-8885-CryptoAuth-ATSHA204A-Datasheet.pdf; abgerufen am 20. September 2014. Sep. 2014.

[13] *Atmel Corporation, Atmel AT-SHA204 Datasheet*. http://www.atmel.com/Images/Atmel-8740-CryptoAuth-ATSHA204-Datasheet.pdf; abgerufen am 10. August 2014. Juli 2014.

[14] *Atmel CryptoAuthentication™, Family of Hardware Security Solutions ATSHA204, ATAES132 and ATECC108*. http://www.atmel.com/Images/Atmel-8756-CrytpoAuth-ATSHA204-ATAES132-ATECC108-Brochure.pdf; abgerufen am 20. September 2014. Sep. 2014.

[15] *AVR-Crypto-Lib*. http://www.das-labor.org/wiki/AVR-Crypto-Lib; abgerufen am 20. September 2014. Sep. 2014.

[16] Chris Barlow. *Further task scheduling with Arduino / AVR*. http://chrisbarlow.wordpress.com/2012/12/28/further-task-scheduling-with-arduino-avr/; abgerufen am 03. August 2014. Dez. 2012.

[17] Mihir Bellare. "New Proofs for NMAC and HMAC: Security without Collision-Resistance". In: *Advances in Cryptology – Crypto 2006 Proceedings*. Springer-Verlag, 2006, S. 602–619.

[18] Thomas Bode. *Vorlesung Relationale Datenbanken 2013, Kapitel 3 Transaktionen*. "Rheinische Friedrich-Wilhelms-Universität Bonn". Sep. 2013.

[19] Alexander Bogoslawski. *Vorgestellt: Multithreading mit Arduino*. http://blog.flarandr.de/blog/2013/04/01/vorgestellt-multithreading-mit-arduino/; abgerufen am 07. August 2014. Apr. 2013.

[20] Christian Büch. *Seminararbeit, SPI*. http://www.uni-koblenz.de/~physik/informatik/MCU/SPI.pdf; abgerufen am 10. August 2014. Juni 2006.

[21] *Die Zweidrahtbussysteme I²C-Bus und SPI-Bus*. http://gedankenlyrik.bplaced.net/www/studium/seminararbeit_2008.pdf; abgerufen am 03. August 2014. 2008.

[22] T. Dierks und E. Rescorla. *RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2*. http://tools.ietf.org/html/rfc5246; abgerufen am 01. August 2014. Aug. 2008.

[23] Frank Durda FreeBSD. *Serial and UART Tutorial*. https://www.freebsd.org/doc/en/articles/serial-uart/; abgerufen am 16. August 2014. Apr. 2014.

[24] *Fritzing*. http://fritzing.org/home/; abgerufen am 23. September 2014. Sep. 2014.

[25] Patrick Gallagher. *FIPS PUB 180-4: Secure Hash Standard (SHS)*. Techn. Ber. National Institute of Standards und Technology, Aug. 2012.

[26] *Github, ArduinoThread*. https://github.com/ivanseidel/ArduinoThread; abgerufen am 16. August 2014. Apr. 2014.

[27] André Helbig. *I²C als Feldbus (I²C-Bus mit langer Leitung)*. http://www.cc2net.de/Tips/tips.html; abgerufen am 12. August 2014. Sep. 2002.

[28] *HÄUFIG GESTELLTE FRAGEN (FAQ) ZU PHYSX*. http://www.nvidia.de/object/nvidia-physx-faq-de.html; abgerufen am 19. September 2014. Sep. 2014.

[29] *I2C Adress*. http://www.gammon.com.au/forum/?id=10896&reply=9#reply9; abgerufen am 10. August 2014. März 2013.

[30] *IndieGoGo , Arduissimo Multicore-Arduino*. https://www.indiegogo.com/projects/arduissimo-multicore-arduino-for-more-arduino-and-raspberry-pi-interfacing; abgerufen am 03. September 2014. Sep. 2014.

[31] *Jim Lindblom, ATSHA204 Arduino Library GitHub Repository*. https://github.com/jimblom/sha204-Breakout; abgerufen am 19. August 2014. Nov. 2012.

[32] *Kickstarter , Smartduino*. https://www.kickstarter.com/projects/fairduino/smartduino-open-system-by-former-arduinos-manufact; abgerufen am 05. September 2014. Sep. 2014.

[33] *Kickstarter , Smartduino Kommentare*. https://www.kickstarter.com/projects/fairduino/smartduino-open-system-by-former-arduinos-manufact/comments; abgerufen am 06. September 2014. Sep. 2014.

[34] Peter Knight. *Cryptosuite , Cryptography suite for Arduino*. https://code.google.com/p/cryptosuite/; abgerufen am 09. August 2014. Juli 2010.

[35] H. Krawczyk, M. Bellare und R. Canetti. *RFC 2104 - HMAC: Keyed-Hashing for Message Authentication*. https://tools.ietf.org/html/rfc2104; abgerufen am 01. August 2014. Feb. 1997.

[36] Thomas Kugelstadt. "Extending the SPI bus for long-distance communication". In: *Analog Applications Journal* (2011). http://www.ti.com.cn/cn/lit/an/slyt441/slyt441.pdf; abgerufen am 10. August 2014.

[37] Peter Mandl. *Grundkurs Betriebssysteme*. ISBN 3-8348-0392-8. Vieweg+Teubner Verlag, Jan. 2008.

[38] Michael Meier. *Intrusion Detection effektiv! Modellierung und Analyse von Angriffsmustern*. ISBN 3-5404-8251-2. Springer-Verlag, Feb. 2007.

[39] A. Menezes, P. van Oorschot und S. Vanstone. *Handbook of Applied Cryptography*. 5. http://cacr.uwaterloo.ca/hac/; abgerufen am 06. August 2014. CRC Press, Aug. 2001. ISBN: 0-8493-8523-7.

[40] *Microchip Technology Inc.: In-Circuit Serial Programming™ (ICSP™) Guide*. http://ww1.microchip.com/downloads/en/devicedoc/30277d.pdf; abgerufen am 06. August 2014. Mai 2013.

[41] *Mikrocontroller.net, Arduino*. http://www.mikrocontroller.net/articles/Arduino; abgerufen am 05. September 2014. Sep. 2014.

[42] Mindaugas. *ScienceProg, Microwire compared to SPI and I2C*. http://www.scienceprog.com/microwire-compared-to-spi-and-i2c/; abgerufen am 03. August 2014. Sep. 2007.

[43] René Neff, Thomas Trimborn und Matthias Wübbeling. "ARINA: Arduino Remote Infrared Network Adapter". In: *CSCUBS Conference 2014*. http://cscubs.cs.uni-bonn.de/2014/data/paper_7.pdf; abgerufen am 02. August 2014. Institute of Computer Science 4 - IT Security - Universität Bonn, Germany. Mai 2014.

[44] *NightShade Electronics, Kickstarter: Arduino energyShield*. https://www.kickstarter.com/projects/nightshadeelectric/the-energyshield-power-for-your-arduino-creativity; abgerufen am 02. September 2014. Sep. 2014.

[45] *NXP Semiconductors, I2C-bus specification and user manual*. http://www.nxp.com/documents/user_manual/UM10204.pdf; abgerufen am 05. August 2014. Apr. 2014.

[46] Robert W. Patterson und Kathleen M. Patterson. *Arduino for Teens, (For Teens (Course Technology))*. 1st. Cengage Learning PTR, Nov. 2013.

[47] *Reddit , SmartDuino - The Biggest Fraud in Kickstarter History is Currently Unfolding*. http://www.reddit.com/r/arduino/comments/1rmbr8/smartduino_the_biggest_fraud_in_kickstarter/; abgerufen am 04. September 2014. Sep. 2014.

[48] Christoph Redecker. *The AVR TWI (I²C) Interface*. http://www.avrbeginners.net/architecture/twi/twi.html; abgerufen am 01. August 2014. Apr. 2014.

[49] *RFC4231, Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512*. http://tools.ietf.org/html/rfc4231; abgerufen am 19. August 2014. Dez. 2005.

[50] Christian Rheinnecker. *ArduinoPrX: Arduino Boards*. http://arduinoprx.de/?pages=15; abgerufen am 02. September 2014. Sep. 2014.

[51] Frank Riemenschneider. *Neuer Apple A7 kommt mit zwei Cyclone-Cores*. http://www.elektroniknet.de/halbleiter/prozessoren/artikel/101298/; abgerufen am 12. September 2014. Sep. 2013.

[52] *RS232 Specifications and standard*. http://www.lammertbies.nl/comm/info/RS-232_specs.html; abgerufen am 20. September 2014. Sep. 2014.

[53] *Schaltplan des ATSHA204 Authentication Chip Breakout*. http://dlnmh9ip6v2uc.cloudfront.net/datasheets/BreakoutBoards/atsha204-breakout-v01.pdf; abgerufen am 03. August 2014. Feb. 2012.

[54] Ingo Schoendube. "Interrupt-Techniken und ihre Probleme aus Softwaresicht". https://homepages.thm.de/~hg6458/semsts/Interrupt-Techniken_Schoendube.pdf; abgerufen am 01. August 2014. Technische Hochschule Mittelhessen, Sep. 2011.

[55] Anand Lal Shimpi. *The iPhone 5s Review*. http://anandtech.com/show/7335/the-iphone-5s-review/8; abgerufen am 09. August 2014. Sep. 2013.

[56] *SMBus (system management bus)*. http://www.itwissen.info/definition/lexikon/system-management-bus-SMBus.html; abgerufen am 05. September 2014. Sep. 2014.

[57] *Sparkfun, Atmel AT-SHA204 Breakout*. https://www.sparkfun.com/products/11551; abgerufen am 13. September 2014. Sep. 2014.

[58] *SPI library*. http://arduino.cc/en/Reference/SPI; abgerufen am 14. September 2014. Sep. 2014.

[59] *SPI (serial peripheral interface)*. http://www.itwissen.info/definition/lexikon/serial-peripheral-interface-SPI.html; abgerufen am 13. September 2014. Sep. 2014.

[60] *Tastaturen*. http://www.robotrontechnik.de/index.htm?/html/zubehoer/tastaturen.htm; abgerufen am 20. August 2014. Aug. 2014.

[61] *Technische Daten des Hauscomputers: Die maximal zulässige Leitungslänge des verwendeten Hausbusses*. http://hauscomputer.gmxhome.de/Technik.htm; abgerufen am 11. September 2014. Sep. 2014.

[62] James M. Turner. *FIPS PUB 198-1: The Keyed-Hash Message Authentication Code (HMAC)*. Techn. Ber. National Institute of Standards und Technology, Juli 2008.

[63] Scott Miller und David Madore Ulrich Drepper. *Linux Man Page, sha256sum(1)*. http://linux.die.net/man/1/sha256sum; abgerufen am 20. August 2014. 2010.

[64] *UM10204: I2C-bus specification and user manual*. http://www.nxp.com/documents/user_manual/UM10204.pdf; abgerufen am 03. August 2014. Apr. 2014.

[65] *United States Mint, Quarter Dollar*. http://www.usmint.gov/mint_programs/circulatingCoins/index.cfm?action=CircQuarterDollar; abgerufen am 20. September 2014. Sep. 2014.

[66] T. Ylonen und Ed. C. Lonvick. *The Secure Shell (SSH) Transport Layer Protocol*. http://tools.ietf.org/html/rfc4253; abgerufen am 02. August 2014. Jan. 2006.

Anhang

Es folgt eine Übersicht, der in der DataZone des Atmel AT-SHA204 gespeicherten Daten.

Slot: 0
Address: 0
Value: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Slot: 1
Address: 8
Value: 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11

Slot: 2
Address: 10
Value: 4B 93 91 A9 55 63 4F CA 6D A3 DE F2 65 56 5E 4 83 EA B6 28 FE E1 10 AF 5E CC C6 7C 5F C0 92 84

Slot: 3
Address: 18
Value: A0 28 62 E1 3B 80 3B F7 15 7B 6 B7 8B 32 84 8A 36 BD FE 9D 5C 4D A D6 C1 D9 A6 A7 EE 41 09 A3

Slot: 4
Address: 20
Value: BE A7 45 A4 51 33 66 CD 02 18 FA 60 9D B6 CE 28 28 29 5A 50 CD FA 7D EE 71 C5 C1 96 09 60 74 32

Slot: 5
Address: 28
Value: 2A 9E CC F7 7D 46 7E F0 BE A0 E9 7F CD CF F6 C3 5C 65 47 EA 29 3 79 B0 8 38 A1 A0 C DA 31 CA

Slot: 6
Address: 30
Value: D9 8 C9 89 A3 CB 4B 83 6D DB DB BF 86 6 FB 2F A6 E0 16 00 16 66 3D 79 16 32 2E 49 F1 5C 1A 73

Slot: 7
Address: 38
Value: 24 F5 65 4E BE 15 EA 65 86 32 48 7D A3 CB B0 D 84 71 36 92 2F 13 8A 00 A0 64 9F 32 C1 A1 DB 64

Slot: 8
Address: 40
Value: A4 FD 49 7C 7E EC 43 57 2E 9F 7E C3 59 39 C9 74 38 7D 8E B7 A6 FA E9 6F 7E 67 F7 5F 21 2C 19 51

Slot: 9
Address: 48
Value: F5 A3 D1 32 E8 6C 6B 41 DC 78 70 94 F 25 4B E1 93 31 9A C9 DC 80 1B 30 5E 25 2C A8 90 02 72 D6

Slot: 10
Address: 50
Value: 68 1 AC 64 A8 2A FD 32 30 9E B C9 DD 55 1F F5 28 F5 D6 BE 4C E6 3C 93 2B 14 41 82 54 25 36 08

Slot: 11
Address: 58
Value: 45 7B A4 C1 DC 2D C3 11 6 AF E3 C9 57 8E 21 6C D DF 5F DB 20 88 DF A0 48 B2 AF 93 D7 F2 F8 F2

Slot: 12
Address: 60
Value: F5 A3 4 F8 C1 3C 46 7A 2 F9 32 42 56 3B 9C C5 42 FC 2 16 1F 34 95 C 6 BF 3E 5D DC 58 D6 0C

Slot: 13
Address: 68
Value: 6D AE 5D 90 5C CD 1E 3D AB E1 9 99 7F 38 75 DC AD 56 F9 4E CF 56 AB A1 B8 94 3 8E 31 A7 A9 09

Slot: 14
Address: 70
Value: 4C F8 BA 17 F2 B9 6D 89 9C 51 81 09 69 20 D4 E0 40 9B 29 1A F4 90 CF 33 86 3F 86 67 43 82 75 08

Slot: 15
Address: 78
Value: ED 79 A1 6D 1F F9 5C CC 74 B9 46 38 6E CC E0 44 5A E8 33 1A 16 37 71 21 F3 F7 EF 30 57 76 BD 43