Erklärung zur Abschlussarbeit

Hiermit versichere ich, dass die eingereichte Ausarbeitung von mir persönlich verfasst und meine eigene individuelle Prüfungsleistung ist.

Ich versichere, dass die Ausarbeitung und auch keine Teile davon durch künstliche Intelligenz (KI) dergestalt erstellt wurden, dass das KI-Werk bzw. KI-Werkteile meine eigene Prüfungsleistung ersetzen. Ich versichere, KI allenfalls eingesetzt zu haben, um einen von KI für meine Aufgabenstellung ausgearbeiteten Lösungsvorschlag kritisch zu beurteilen und/oder einen Überblick über Aspekte zu erhalten, die für die von mir in Eigenleistung zu erbringende Prüfungsleistung relevant sein könnten. Soweit durch die Aufgabenstellung bzw. Prüferhinweise der Einsatz von KI vorgegeben ist, sind die von KI erzeugten Werkteile von mir in der Arbeit entsprechend gekennzeichnet. Mir ist bewusst, dass die von KI erzeugten Werkteile auf ihre Validität zu überprüfen und nicht zitierfähig sind.

Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungsnachweise andernorts eingereicht wurden.

Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen in gedruckter oder elektronischer Form sind gekennzeichnet. Sämtliche Literatur und sonstige Quellen sind nachgewiesen und im Literatur- und Quellenverzeichnis aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.

Ich bin ferner damit einverstanden, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs in elektronischer Form anonymisiert versendet und gespeichert werden kann.

Wird die Erklärung nicht abgegeben, kann von der Korrektur der eingereichten Arbeit abgesehen werden; die eingereichte Arbeit gilt als mit der Note „nicht ausreichend“ bewertet.

Mir ist bewusst, dass Täuschungen nach der für mich gültigen Studien- und Prüfungsordnung / nach § 6 RaPO / § 48 BayVwVfG geahndet werden.

Die Zustimmung zur elektronischen Plagiatsprüfung wird erteilt.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ort, Datum Unterschrift des Verfassers / der Verfasserin

Der Veröffentlichung der Master-/Bachelorarbeit in der Bibliothek der Technischen Hochschule Aschaffenburg wird[nicht] zugestimmt.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ort, Datum Unterschrift des Verfassers / der Verfasserin

Implementing the SHA-1 on the ATmega328p

Timo Selzam   
*Ingeniuerswissenschaften und Informatik*  
*TH Aschaffenburg*Aschaffenburg, Germany  
s220864@th-ab.de

*Abstract*—Durch die immer größere Verbreitung von Microcontrollern in elektronischen Schaltungen steigt die Wahrscheinlichkeit, dass diese angegriffen oder die Daten verändert werden. Um diesem Umstand zu begegnen, soll eine mathematische Einwegfunktion (SHA-1) zum Einsatz kommen. In dieser Arbeit präsentiere ich, wie dieser Algorithmus genutzt werden kann, um die Sicherheit bei der Nutzung von Microcontrollern zu erhöhen. Zur Präsentation wird ein ATmega328P genutzt, der die zu verschlüsselnde Nachricht, sowie die Kommandos zur Verschlüsselung und zur Ausgabe des Hashs über eine USART Schnittstelle erhält bzw. ausgibt.

Keywords—ATmega328, SHA-1, security

# Introduction

In fast allen elektronischen Geräten werden heutzutage Microcontroller zur Steuerung oder Datenverarbeitung eingesetzt. Oftmals speichern bzw. verarbeiten sie viele Daten die sie von Sensoren oder über eine externe Texteingabe erhalten. Um diese Geräte vor Manipulation oder Datendiebstahl zu schützen, soll ein Verfahren angewendet werden, um diese Gefahren zu minimieren. Geläufige Methoden, um Daten oder Passwörter zu verschlüsseln bzw. auf ihre Integrität zu überprüfen sind zum Beispiel kryptographische Hashfunktionen wie “MD5” oder “SHA 1”, “SHA 2”, “SHA 3”. Diese funktionieren alle nach einem ähnlichen Prinzip, bei dem die ursprüngliche Nachricht in Blöcke aus fest definierten Größen aufgeteilt werden. Diese werden anschließend mit vorgeschriebenen Zahlenreihen aufgefüllt und anschließend mit festgelegten Konstanten, Hashs und Formeln verrechnet. Selbst die kleinste Abweichung von der ursprünglichen Nachricht führt zu einem komplett anderen Ergebnis. Das zurückrechnen eines Hashs, um die original Nachricht zu erhalten ist nicht möglich, da bei der Berechnung ein Überlaufen der Variablen absichtlich in Kauf genommen wird. Aufgrund der genannten Eigenschaften solcher Funktionen soll in diesem Paper aufgezeigt werden, wie der SHA 1 Algorithmus aufgebaut ist und wie man ihn auf einem ATmega328p implementieren kann.

# Theoretical questions

## Mathematical one way function

Die Definition einer mathematischen Einwegfunktion lautet: “ Eine Hashfunktion ist eine Ein-Weg-Funktion (one-way oder preimage resistant function), wenn es für einen gegebenen Fingerabdruck *z* berechnungsmäßig praktisch unmöglich ist, eine Nachricht *M* mit *h(M )* = z zu finden.” [1, p. 96]. Kurz gesagt bedeutet das, dass das Rekonstruieren der ursprünglichen Nachricht, auch mit der Kenntnis über den Hash sowie die verwendete Funktion, so schwer ist, dass diesen Umstand als praktisch unmöglich anzusehen ist. Ausnahmen hiervon sollen zu einem späteren Zeitpunkt behandelt werden.

## Typical applications

Eine typische Anwendung von Einwegfunktionen stellt die Kryptographie dar. Zum Beispiel wird ein Passwort nicht in reiner Textform in einer Datenbank abgelegt, sondern es wird der Hash des Passwortes gespeichert. Auf diese Weise kann bei einer Anmeldung der berechnete Hash des eingegebenen Passworts mit dem Abgelegten überprüft werden. Sollte die Datenbank kompromittiert werden, ist das vergebene Passwort nach wie vor unbekannt. Eine weitere Anwendung ist die Überprüfung auf Echtheit / Veränderung von Dateien. Wenn die ursprüngliche Datei bzw. deren Hash bekannt ist, kann die Erhaltene sehr einfach und schnell überprüft werden. Selbst eine minimale Änderung, wie zum Beispiel die Benutzung eines “A” anstatt eines “a” führt zu einem komplett unterschiedlichen Hash. Somit ist eine “sinnvolle” Veränderung einer Datei so gut wie ausgeschlossen.

## Preimage Resitance

Der Begriff “preimage resistance” bezieht sich auf die Kollisionsfreiheit der genutzten Hashfunktion.

First Preimage Resistance: Von einer first preimage resistance spricht man, wenn es “für einen gegebenen Fingerabdruck *z* berechnungsmäßig praktisch unmöglich ist, eine Nachricht *M* mit *h(M)* = *z* zu finden”. [1, p. 96]

Second Preimage Resistance: Die second preimage resistance beschreibt, “wenn es berechnungsmäßig praktisch unmöglich ist, eine Nachricht M′ ≠ M mit h(M) = h(M′) zu finden.” [1, p. 95]

## Kollision

Von einer Kollision spricht man, wenn zwei unterschiedliche Dokumente bzw. Passwörter zu ein und demselben Hash führen. Sollte eine Hashfunktion häufig zu Kollisionen führen bzw. leicht gefunden werden können, ist dieser Algorithmus nicht als sicher anzusehen.

Ist die Eingabe nicht beschränkt, wird es zwangsläufig zu Kollisionen kommen, da es mehr Eingabe- als Ausgabemöglichkeiten gibt.

## XOR-Funktion

Um die Integrität einer Nachricht zu überprüfen, ist eine reine XOR Funktion nicht geeignet, da, sobald der Schlüssel bekannt ist, jede Nachricht entschlüsselt werden kann. Des Weiteren sind Kollisionen bei einer linearen Funktion wesentlich wahrscheinlicher, als bei anderen Hash-Funktionen.

## Geburtstagsproblem

Das Geburtstagproblem beschreibt, dass ab einer Gruppengröße von 23 Personen die Wahrscheinlichkeit 50% beträgt, dass mindestens zwei Personen am gleichen Tag, ungeachtet des Jahrgangs, geboren wurden. [2, p. 36] Diese Wahrscheinlichkeit lässt sich anhand folgender Formel berechnen:

mit . Für eine Wahrscheinlichkeit von 50% sei

Somit ergibt sie die Schätzung:

Die Variable sei die Anzahl der Personen und sei die Anzahl der “Nichtgeburtstage”.

Bei dem sogenannten Geburtstagsangriff werden zufällige Dateien / Nachrichten 1 bis *k* ausgewählt und der dazugehörige Hash berechnet. Danach wird verglichen, ob der gleiche Hasch zwei Mal vorkommt. Durch das Nutzen des Geburtstagsproblems werden Versuche benötigt, um mit einer Wahrscheinlichkeit von 50% eine Kollision in dem von uns genutzten SHA 1 zu finden.

Zum Vergleich: Um eine Nachricht mit dem gleichen Hash wie eine zuvor festgelegte Nachricht zu erhalten werden Wiederholungen benötigt.

Bei der Nutzung eines Rechners, der 1 Mrd. Versuche pro Sekunde schafft, benötigt dieser ca. Jahre mit dem Ausnutzen des Geburtstagsproblems und ca. Jahre ohne die Nutzung.

## Blockchain

Mathematische Einwegfunktionen spielen im Kontext von Bitcoin eine große Rolle im Hinblick auf Sicherheit, Konsens und Integrität.

Bitcoin nutzt Hashfunktionen wie zum Beispiel SHA 256 um Transaktionen sowie ganze Blöcke zu verschlüsseln. Jeder Block enthält den Hash des Vorherigen, wodurch eine nachträgliche Veränderung ausgeschlossen wird, da dies den Hash aller Nachfolgenden ungültig machen würde.

Jede Transaktion wird durch eine mathematische Einwegfunktion verschlüsselt und abschließend mit einer Signatur, erstellt durch asymmetrische Kryptographie, versehen.

# Analysis/Discussion of the Firmware

In diesem Abschnitt soll die Implementierung der Firmware beschrieben und analysiert werden. Das Hauptaugenmerk soll dabei auf die Umsetzung des SHA 1 Verfahrens, sowie auf den USART Kommandointerpreter gelegt werden.

## Implementig USART

Aufgabe der Firmware ist es, eine Zeichenkette einzulesen, zu verschlüsseln und wieder auszugeben. Um diese Funktionen ausführen zu können muss zuerst eine Möglichkeit geschaffen werden, damit Nachricht und Kommandos ein- und ausgegeben werden können. In diesem Fall wird dazu die USART Schnittstelle des ATmega328p genutzt. USART steht für universal s*ynchronous* / a*synchronous* receiver transmitter.

Um eine Nachricht senden und empfangen zu können, werden die Leitungen RxD (receive), TxD (transfer) sowie ein gemeinsamer GND benötigt. Wenn die Datenübertragung synchron durchgeführt werden soll, wird noch eine weitere Leitung für das gemeinsame Taktsignal benötigt. Des Weiteren ist darauf zu achten, dass die Baudrate (Übertragungsgeschwindigkeit) von Sender und Empfänger identisch ist.

Damit beide Seiten die Übertragung richtig interpretieren ist es unabdingbar das gleiche Übertragungsprotokoll zu nutzen. Dieses setzt sich, wie in Abbildung 1 [3, p. 147] zu erkennen, aus der Nachrichtenlänge, Start- und Stoppbits und ggf. einem Paritätsbit zusammen. Die Nachrichtenlänge kann zwischen 5 und 9 Bits betragen, das Paritätsbit kann eine gerade oder ungerade Parität anzeigen oder deaktiviert werden und es besteht die Wahl zwischen einem und zwei Stoppbits.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

KI-generierte Inhalte können fehlerhaft sein.

Abbildung 1: Aufbau USART Protokoll

Das Protokoll, das für die Implementierung des SHA 1 genutzt wird, nutzt eine asynchrone Übertragung, eine Datenlänge von 8 Bit, ein Stoppbit und kein Paritätsbit. Die Baudrate wird in einem anderen Programmteil festgelegt. Des Weiteren wird das Senden und Empfangen, sowie ein Empfangsinterrupt aktiviert. Die Festlegung der Parameter erfolgt in den Registern UCSR0B, UCSR0C, UBRR0H und UBRR0L. Diese Einstellungen werden vorgenommen, sobald die Funktion *usart\_init*() aufgerufen wird.

Nachdem die grundlegenden Einstellungen vorgenommen wurden, werden die Funktionen zum Senden und Empfangen der Daten geschrieben.

Die Funktion *usart\_put\_byte*() wartet bis das UDREn Flag im Register UCSRnA durch eine 1 signalisiert, dass das UDRn (USART data register) leer und somit bereit zum übertragen ist. Diese Überprüfung ist notwendig, damit es zu keinen Kollisionen bei der Datenübertragung kommt. Danach wird das Byte, das an die Funktion übergeben wurde, in das UDRn geschrieben und somit gesendet.

An die Funktion *usart\_put\_string*() wird ein Pointer bzw. Array übergeben, dass eine Zeichenkette enthält. Zuerst wird geprüft, ob an der Adresse auf die der Pointer zeigt ein Nullterminator steht. Solange dies nicht der Fall ist, wird das Byte an der aktuellen Adresse an die Funktion *usart\_put\_byte*() übergeben. Anschließend wird die Adresse, auf die der Pointer zeigt um eine Variablengröße, in diesem Fall ein Byte, erhöht. Diese Schleife läuft solange, bis der Pointer auf eine Adresse zeigt, die den Nullterminator enthält und somit das Ende des Strings signalisiert.

Der Vollständigkeit halber wurde die Funktion *usart\_receive*() ebenfalls implementiert. Sie kann genutzt werden um im polling Verfahren aktiv darauf zu warten, dass das UDRn eine Nachricht erhält. Hierzu wird das RXCn Flag im UCSRnA Register beobachtet. Erst wenn es 1 wird, wird die *for* Schleife verlassen und das UDRn ausgelesen.

## Implementing SHA 1

Bevor der Code für den SHA 1 geschrieben werden kann, soll der Ablauf hier schematisch besprochen werden. Eine nähere Betrachtung findet im späteren Verlauf statt.

Der genutzte Algorithmus erzeugt aus einer beliebigen Zeichenkette einen Hash mit 160 Bit. Um dieses Ziel zu erreichen, wird die Nachricht in mindestens einen 512 Bit breiten Block aufgeteilt. Dieser ist wie folgt aufgebaut. Einer Nachricht mit der Länge *l* Bits wird eine einzelne 1 angehängt und mit einer Anzahl *k* Nullen aufgefüllt, wobei *k* die kleinste, nicht negative Lösung der Gleichung

sei. Die ursprüngliche Nachrichtenlänge in Bit wird in den letzten zwei 32 Bit breiten Worten des letzten Blocks geschrieben. [3, p. 13]

Um eine fehlerfreie Funktion der Firmware sicher zu stellen ist es essentiell, dass die empfange Nachricht mit der gesendeten Nachricht übereinstimmt. Dieser Faktor spielt eine große Rolle bei der Auswahl der Baudrate, da eine höhere Übertragungsgeschwindigkeit zwar die Übertragungszeit verkürzt, aber im gleichen Zuge auch die Robustheit verringert.

Zur Bestimmung der Baudrate muss die Bytelänge (*l*), die maximal zu übertragenden Bytes (*n*) und die möglichen Baudraten (*Bd*) betrachtet werden. Zur Beurteilung wird die folgende Formel genutzt

|  |  |  |
| --- | --- | --- |
| Baudrate | Byte Zeit (1/8/1) | Übertragungszeit (T) |
| 1200 | 8,33 ms | 8,33 s |
| 2400 | 4,17 ms | 4,17 s |
| 4800 | 2,08 ms | 2,08 s |
| 9600 | 1,04 ms | 1,04 s |

Da, wie Eingangs schon erwähnt, die Robustheit oberste Priorität hat und eine Übertragungszeit von ca. 1,04 Sekunden für die maximal mögliche Zeichenanzahl ausreichend für diese Applikation ist, wählen wir eine Baudrate von 9600 Bd. Um diese später in das Register des Microcontrollers schreiben zu können, muss sie zuvor noch in das passende Format umgerechnet werden. Dazu definieren wir die Makros *F\_CPU*, *BAUDRATE* und *Teiler* und nutzen die Formel

Der Compiler wird jedes Vorkommen der Zeichenketten *F\_CPU*, *BAUDRATE* und *Teiler*() mit den definierten bzw. berechneten Werten ersetzen. Es ist zu beachten, dass große Werte mit UL (unsigned long) versehen werden müssen, damit diese Zahlen nicht falsch interpretiert werden.

Die Datentypen *uint32\_t* und *uint64\_t* wurden für eine bessere Lesbarkeit des Programms erstellt. Wenn der Header für einen anderen Microcontroller genutzt werden sollte ist davor sicherzustellen, dass Datentypen und längen identisch sind.

Um die Berechnung des Hashs übersichtlich gestalten zu können wurde ein Funktionspointer erstellt, auf den im späteren Verlauf eingegangen werden soll.

Die zuvor erstellten Daten *SHA\_1.h* und *usart.c* müssen in die *SHA\_1.c* per include eingebunden werden, damit sie verwendet werden können. Weiterhin werden Header zur Nutzung von Interrupts (avr/interrupt.h), zur Speicherverwaltung (stdlib.h) und zur Vereinfachung der Arbeit mit den Registern des Microcontrollers (avr/io.h) eingebunden. In *main*.c werden in der *main* Schleife die Funktionen *usart\_init()* und *sei()* aufgerufen um die USART Schnittstelle zu initialisieren und externe Interrupts zu aktivieren. Die pointer *M* und *W* werden erzeugt. Diese werden im späteren Verlauf genutzt um die eingegebene Nachricht in 512 Bit Blöcke aufzuteilen bzw. die für den SHA 1 benötigten 80 Rechenschritte durchzuführen. Damit nicht unnötig Speicher reserviert werden muss erfolgt die Speicherallokation dynamisch bei der Berechnung des Hashs, Da erst zu diesem Zeitpunkt abschließend bekannt ist, wie viel Speicher tatsächlich benötigt wird. Der Speicher des Puffers *msg*, der bereits in der *SHA\_1.h* erzeugt wurde, wird über die Funktion *malloc()* mit der Breite von 8 Bit erzeugt und einer Adresse zugewiesen.

Damit das Programm nicht nur einmal durchlaufen und danach beendet wird, findet die Abarbeitung in einer *while* Schleife statt. Die Bedingung der Schleife wird mit “1” angegeben, damit sie immer erfüllt ist.

Um den Speicher das ATmega328p nicht über zu strapazieren wurde die maximale Nachrichtenlänge auf 1000 Zeichen festgelegt. In jedem Durchlauf des Programms wird geprüft, ob diese Restriktion eingehalten wurde. Ist dies nicht der Fall, wird eine Fehlerroutine aufgerufen und der Programmablauf unterbrochen.

Über eine Verbindung zu einem PC kann die Nachricht über einen seriellen Monitor an den ATmega328p geschickt werden. In dem Programmcode des Microcontrollers wird die eingehende Nachricht über eine ISR (interrupt service routine) in den Puffer *msg* geschrieben. Bei jedem Aufruf der ISR wird der Speicher des Puffers um zwei weitere Bytes durch die Funktion *realloc* erweitert. Dies ist nötig um das neu eingegangen Byte der Nachricht ablegen und den Nullterminator anhängen zu können. Sollte die Speicherreallokation fehlschlagen springt das Programm in eine Fehlerroutine und die weitere Abarbeitung wird unterbrochen. Bei einer erfolgreichen Reallokation wird das empfangene Byte im Puffer abgelegt und der Längenindex *len* um eins erhöht. Es können solange Daten eingegeben werden bist entweder die Nachrichtenlänge 1000 überschreitet oder bis das Kommando zur Berechnung des Hash eingegeben wird.

Die Eingabe des Kommandos wird mit einer *if* Abfrage fortlaufend kontrolliert. Wenn erkannt wird, dass *failsafe “0”,* das vorletzte Zeichen “#” und das letzte Zeichen “!” ist, wird die Verarbeitung gestartet. Aufgrund der Vorgaben ist es somit nicht möglich die Zeichen “#” und “!” hintereinander in der zu verschlüsselnden Nachricht zu nutzen. Die Variable *failsafe* wird genutzt, um den Programmablauf zu steuern und Kollisionen zu verhindern.

Im ersten Schritt der Bearbeitungsrutine wird die benötigte Blockanzahl ermittelt. Hierzu wird der Umstand genutzt, dass bei einer Division von integer Variablen die Nachkommastellen entfallen. In diesem speziellen Fall wird der Längenindex der Nachricht mit 8 Multipliziert und das Ergebnis durch 448 geteilt. Anschließend wird eine eins addiert, um die Anzahl *n* der benötigten 512 Bit Blöcke zu erhalten. Die Berechnung in Bits wurde gewählt um eine einheitliche Datengrundlage zu schaffen.

Nach der Berechnung werden die letzten zwei Stellen des Puffers, die nur das Kommando darstellen, mit dem Nullterminator beschrieben.

Im weiteren Verlauf wird mit der Funktion *calloc* Speicher für “M” und “W” allokiert. Das Besondere an der genutzten Funktion ist, dass der allokierte Speicher gleich zu Beginn mit “0” beschrieben wird. Somit muss der Block nicht mit *k* Nullen aufgefüllt werden. *M* stellt die einzelnen 512 Bit Blöcke dar ()und *W* wird zur Berechnung der Durchläufe genutzt. Auch hier wird überprüft, ob die Speicherallokation erfolgreich war. Falls nicht wird eine Fehlerroutine aufgerufen und der Programmablauf gesperrt.

Bei dem Aufruf der Funktion *pad()* werden die Pointer *msg* und *M* übergeben. Hierbei handelt es sich um einen *call by reference* Aufruf, der es ermöglicht, die Puffer in der Funktion zu bearbeiten. Die Funktion *pad()* wird genutzt um die einzelnen Bytes des Puffers *msg* in die Wortorganisierten Blöcke *M* zu schreiben, nach dem letzten Zeichen der Nachricht eine 1 anzuhängen. Außerdem wird die ursprüngliche Nachrichtenlänge in Bit in den letzten beiden Wörtern des letzten Blocks geschrieben.

Zur Umsetzung werden zwei *if* Schleifen genutzt. Die erste Schleife wird zur Indizierung der 16 *n* Worte *M* genutzt. In der Zweiten wird bei jedem Durchgang mittels der zuvor erstellten temporären Zählvariable *count* geprüft, ob der Puffer *msg[count]* den Nullterminator enthält. Wenn dies nicht der Fall ist wird das enthaltene Zeichen mittels der Zählvariablen *k* der Schleife und einer mathematischen Berechnung auf die vorgesehene Position geschoben. Damit bei dem Schiebevorgang keine Daten verloren gehen muss ein Typecast des 8 Bit breiten Zeichens auf einen 32 Bit breiten Datentypen erfolgen. Anschließend wird die Zählvariable  *count* um 1 inkrementiert.

Wenn die Zeichenkette aus *msg* vollständig übertragen wurde wird über *count* berechnet wo die Nachricht endet und eine einzelne 1 an diese Position angehängt. Es wird eine temporäre Variable *bits* angelegt, in der die Länge *l* der ursprünglichen Nachricht in Bits gespeichert wird. Das vorletzte Wort des letzten Blocks wird, mit Zuhilfenahme des Schiebeoperators rechts, mit den oberen 32 Bit der Variable beschrieben, das letzte Wort des letzten Blocks wird mit den unteren 32 Bit der Variable beschrieben.

Das Hauptaugenmerk wurde auf die zur Berechnung des Hashs benötigte Funktion *process()* gelegt, da sie die komplexeste des ganzen Programms ist. Sie sollte vollumfänglich nutzbar sein, was bedeutet, dass auch Zeichenkette mit mehr als 55 Symbolen eingegeben und mehrere Hashs hintereinander erzeugt werden können. Außerdem sollte sie übersichtlich und ressourcenschonen sein. Um diese Anforderungen zu erfüllen wird ein Funktionspointer genutzt. Der Vorteil dabei ist, dass der Aufruf der zum Berechnungsvorgang benötigten Formel über eine Indizierung erfolgt. Somit wird nicht jede Formel zur Berechnung in eine eigene Schleife eingebettet sondern der Index zur passende Formel anhand des aktuellen Berechnungsschritts berechnet. Die vorgegebenen Konstanten wie auch die initialen Hashs wurden jeweils in einem Array abgelegt und können somit auf die gleiche Art und Weise mit dem berechneten Index ausgewählt werden.

Aufgrund der genutzten Pointer, Arrays und Funktionen ist es möglich die Berechnung mit zwei *for* Schleifen durchzuführen und den Code dafür schmal, überschaubar und gut lesbar zu gestalten.

Zu Beginn wird die temporäre Variable *temp* erstellt. Diese wird benötigt um einen berechneten Wert zwischen zu speichern und zu einem späteren Zeitpunkt in einen der fünf Puffer (*A* bis *E)* zu speichern. Für den Fall, dass die Berechnung mit mehr als einem 512 Bit breiten Block erfolgen soll wird die erste Schleife benötigt. In ihr wird zu Beginn der aktuellste Hashwert in die Puffer *A* bis *E* gespeichert. Bei dem ersten Durchlauf entspricht dieser dem Initialwert, bei allen Weiteren dem Hash der vorherigen Berechnung.

Für die 80 Berechnungsdurchläufe wird die zweite Schleife und für die Indizierung die Zählvariable *t* genutzt. Während der ersten 16 Durchläufe gilt

wobei *i* die Zählvariable der ersten Schleife darstellt. Anhand der Berechnung des Indexes kann man erkennen, dass mit jedem weiteren Durchlauf der ersten Schleife der Index des Puffers *M* um *i* 16 erhöht wird. Nach dem 16. Schritt wird W[t] wie folgt berechnet

Die Funktion *rotl(x, y)* führt eine zyklische Verschiebung des übergebenen Wertes *x* um die Anzahl *y* Bits nach links durch. Der Unterschied zu einer reinen Schiebeoparation liegt darin, dass die herausgeschobenen Bits nicht verloren gehen, sondern von rechts wieder eingeschoben werden.

Zur Berechnung der Variablen *temp* wird folgende Formel verwendet

Sie enthält, neben den bereits bekannten Puffern *W* und *E* und Funktionen, das Array *K* in dem die Konstanten gespeichert sind und den Funktionspointer *funktion*. Für die Indizierung der Beiden wird die Zählvariable *t* der zweiten Schleife durch den Wert 20 dividiert. Durch diese einfache Berechnung ist es möglich alle 80 Durchgänge in nur einer Schleife zu bearbeiten. Die Puffer *A* bis *E* werden in jedem Durchgang nach den Vorgaben neu beschrieben.

Die Puffer *A* bis *E* werden zu den Hashs *H[0]* bis *H[4]* aufaddiert sobald alle 80 Berechnungen pro Block abgeschlossen wurden.

Nachdem alle 512 Bit breiten Blöcke berechnet wurden enthalten die Hashpuffer *H[0]* bis *H[4]* die finalen Werte.

Nach der Berechnung muss der Speicher der Puffer *M* und *W* wieder freigegeben werden, da durch die Berechnung eines neuen Hashs wieder Speicher allokiert wird. Dies passiert, bis kein Speicher mehr vorhanden ist, was zu unvorhersehbaren Auswirkungen führen kann. Für die Freigabe des Speichers wird in der Funktion *de\_init()* die Funktion *free* genutzt und der Pointer auf den NULL Vektor gerichtet.

Damit der Programmfluss gesteuert werden kann wird die Variable *failsafe* auf 1 gesetzt.

Als weitere Anforderung wurde festgelegt, dass nach der Eingabe des Steuerkommandos “#$“ die Ausgabe des berechneten Hash erfolgen soll. Eine Abfrage der Variablen *failsafe* wurde hinzugefügt. Durch diese Maßnahme soll sichergestellt werden, dass zwischen Berechnung und Ausgabe die Werte des Hash nicht geändert werden können.

Zur Ausgabe über die USART Schnittstelle wurde die Funktion *print\_char()* geschrieben, in der zwei *for* Schleifen dafür sorgen, dass der Puffer *H[n]* indiziert und die 8 Bit breiten Datenblöcke aus dem 32 Bit breiten Wort *H[n]* über den Schiebeoperator an die bereits bekannte Funktion *usart\_put\_byte()* übergeben werden. Nachdem alle 160 Bit übertragen wurden wird der Puffer *H[]* durch das Array *Ho[]* auf die initialen Werte zurückgesetzt. Dies ist nötig, um im direkten Anschluss einen neuen Hash aus einer Zeichenkette berechnen zu können, ohne den Microcontroller neustarten zu müssen.

Abschließend wird der Längenindex *len* und *failsafe* auf 0 gesetzt, der allokierte Speicher des Puffers *msg* freigegeben, ein Byte Speicher für *msg* allokiert und geprüft, ob dies fehlgeschlagen ist.

Die Fehlerbehandlungsroutine dient in diesem Programm nur zur Veranschaulichung, wie eine solche aufgebaut und funktionieren könnte. Aus diesem Grund soll sie nicht näher betrachtet werden.

# Summary

Die Arbeit beschreibt die Entwicklung einer Firmware für den Atmega328p Mikrocontroller, die eine Zeichenkette über die USART-Schnittstelle empfängt, mit dem SHA-1 Algorithmus verschlüsselt und den Hash-Wert ausgibt. Die Kommunikation erfolgt asynchron mit einer Baudrate von 9600 Bd und einer 8-Bit-Datenlänge.

Der SHA-1 Algorithmus wird implementiert, um aus einer Nachricht einen 160-Bit langen Hash zu erzeugen, wobei die Nachricht in 512-Bit-Blöcke aufgeteilt und entsprechend dem Standard mit zusätzlichen Bits und Längenangaben versehen wird. Die Berechnung erfolgt in 80 Schritten, wobei zyklische Verschiebungen und logische Operationen genutzt werden.

Die Firmware verwendet dynamische Speicherallokation und begrenzt die Nachrichtenlänge auf 1000 Zeichen. Nach der Eingabe eines Steuerkommandos wird der berechnete Hash über USART ausgegeben. Fehler wie fehlgeschlagene Speicheroperationen werden in einer Fehlerbehandlungsroutine erkannt.

Die Arbeit bietet eine effiziente Implementierung des SHA-1 Hash-Generators auf einem Mikrocontroller, mit Fokus auf ressourcenschonende und fehlerfreie Ausführung.

# Conclussion

## Ergebnisanalyse

Die Implementierung des SHA 1 auf dem Atmega328p hat die erwarteten Funktionen erfüllt. Sowohl die Kommunikation über die USART Schnittstelle wie auch die Berechnung und Ausgabe des Hash erfolgten nach den Vorgaben und wurden effizient in dem Microcontroller umgesetzt.

Durch die Nutzung von Interrupts wurde die Reaktionsgeschwindigkeit erhöht und die dynamischer Speicherzuweisung trägt zur Flexibilität bei.

Die Berechnung wurde durch optimierte Schleifennutzung und Funktionspointer effizient gehalten und die Aufteilung der Nachricht in 512 Bit Blöcke erfolgreich implementiert.

## Schlussfolgerung

Die Implementierung des SHA 1 auf einem Atmega328p wurde erfolgreich durchgeführt und getestet. Alle Anforderungen an den Code wurden getreu der Vorgaben erfüllt.

Der Code ist durch die Kommentare und die Aufteilung in .c und .h Dateien gut les- und wartbar.

Durch den begrenzten Speicher des Microcontrollers ist es nicht möglich einen Hash für sehr lange Zeichenketten oder große Datenmengen erzeugen.

## Empfehlungen

Für eine reelle Verwendung sollte auf neuere Versionen der SHA-Familie zurückgegriffen werden, da SHA 1 als nicht mehr sicher gilt.

Bei der Verschlüsselung von sehr langen Zeichenketten müsste eine höhere Baudrate verwendet werden, um die Übertragungszeit gering zu halten. Des Weiteren könnte ein externes Speichermedium wie eine SD-Karte genutzt werden, um den internen Speicher zu kompensieren.

Je nach Anwendung könnte die dynamische Speicherverwaltung gegen eine statische ersetzt werden, um Speicherprobleme zu minimieren.

Für eine reelle Nutzung müsste die Fehlerbearbeitung final implementiert werden.

Insgesamt wurde die Firmware erfolgreich implementiert und bietet eine zuverlässige Funktionalität. Durch die genannten Verbesserungen könnte sie weiter verbessert bzw. geändert werden.

# Literaturverzeichnis

|  |  |
| --- | --- |
| [1] | D. Wätjen, Kryptographie - Grundlagen, Algorithmen, Protokolle, Wiesbaden : Springer Vieweg, 2018 . |
| [2] | N. Henze, Stochastik: Eine Einführung mit Grundzügen der Maßtheorie, Berlin: Springer Spektrum, 2019. |
| [3] | N. I. o. S. a. Technologie, „FIPS PUB 180-4,“ Gaithersburg, 2015. |

# Abbildungsverzeichnis

[Abbildung 1: Aufbau USART Protokoll 2](#_Toc191715143)