Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jannik Beyerstedt

Sichere und robuste Firmware-Updates von IoT-Geräten

Fakultät Technik und Informatik

Faculty of Engineering and Computer Science

Jannik Beyerstedt

Sichere und robuste Firmware-Updates von IoT-Geräten

Bachelorarbeit eingereicht im Rahmen der Prüfungsordnung Mechatronik 2008

im Studiengang Bachelor of Science Mechatronik
an der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Lehmann
Zweitgutachter: Prof. Dr.-Ing. Martin Hübner

Eingereicht am: 01.06.2017

Zusamenfassung

Jannik Beyerstedt

Thema der Arbeit
Sichere und robuste Firmware-Updates von IoT-Geräten

Stichworte
IoT, Firmwareaktualisierung, Sicherheit, Robustheit, Over the Air, FOTA, RIOT, CoAP

Kurzzusammenfassung
In dieser Arbeit wird ein sicherer und robuster Firmware-Update-Prozess für IoT-Geräte entworfen. Zum Transport der Firmware-Updates wird ein eigenes Datenformat verwendet, um die Authentizität, Integrität und Vertraulichkeit der Update-Daten zu gewährleisten. Für die Durchführung der Updates werden jedoch auch Werkzeuge benötigt, um die Update-Dateien zu erzeugen und den Geräten bereitzustellen. Diese sollen im Entwurf mit einbezogen werden. Die Prozesskette wird außerdem exemplarisch implementiert, um die Anforderungen an den Update-Prozess durch Tests verifizieren zu können.

Jannik Beyerstedt

Title of the paper
Secure and Robust Firmware Updates of IoT Devices

Keywords
IoT, Firmware Update, secure, robust, Over the Air, FOTA, RIOT, CoAP

Abstract
Within this study, a firmware update process should be designed, which is robust and secure, but also suitable for low power IoT devices. The update procedure will use a special file format for delivering the firmware, which is secured by a digital signature verifying its origin and encrypted to ensure the confidality of the firmware. Updating a device also needs tools for creating and distributing these files, like an update server and special build scripts, which should be part of the design as well. For conducting experiments and tests, the update infrastructure will be implemented and deployed at a small scale. These tests should verify the robustness and combined safety and security requirements of the system.

Inhaltsverzeichnis

1. Einleitung

Computer und das Internet sind, zumindest in einem großen Teil der Welt, allgegenwärtig. Viele Menschen tragen mindestens ein Gerät bei sich, welches dauerhaft mit dem Internet verbunden ist. In den letzten Jahren wurde jedoch ein Begriff für eine neue Kategorie von elektronischen Geräten geprägt, die ebenfalls dauerhaft mit dem Internet verbunden sind: Das Internet of Things (IoT).

Alle IoT-Geräte haben gemeinsam, dass sie einen Mikrocontroller enthalten, der eine Firmware ausführt, und dass sie über ein zumeist drahtloses Netzwerk mit dem Internet verbunden sind. Da Mikrocontroller jedoch auch nur eine spezielle Art von Computern sind, sind diese prinzipiell den gleichen Gefahren ausgesetzt, wie alle anderen mit dem Internet verbundenen Geräte. Auch wenn die Firmware sorgfältig entwickelt und getestet wurde, sind Programmierfehler und damit Sicherheitslücken nicht auszuschließen. Zusätzlich muss vermutet werden, dass in einem Massenmarkt von günstigen IoT-Geräten nicht alle Hersteller mit höchster Sorgfalt vorgehen und die Sicherheit ihrer Geräte ausführlich überprüfen lassen.

Im Oktober 2016 kam es beispielsweise zu großflächigen Ausfällen des Internets in den USA, da von Malware befallene IP-Überwachungskameras und Videorecorder massive Attacken auf DNS-Server durchführten. Diese Geräte waren von der Malware „Mirai“ befallen, die unter anderem fest eingestellte Passwörter in den Geräten ausnutzte.[1] Angriffe auf IoT-Geräte sind also schon heute ein Problem und automatisierte Firmware-Updates sind die einzige effektive Möglichkeit, mit der ein Hersteller ihm bekannte Fehler beheben kann.

In dieser Bachelorarbeit wird der Frage nachgegangen, wie ein robustes und sicheres Firmware-Update auf einem Mikrocontroller eines IoT-Geräts durchgeführt werden kann. Dafür müssen jedoch zunächst die Begriffe Sicherheit und Robustheit definiert

[1] https://krebsonsecurity.com/2016/10/hacked-cameras-dvrs-powered-todays-massive-internet-outage/

werden. Außerdem muss beachtet werden, dass der Update-Prozess nicht nur Komponenten in dem IoT-Gerät erfordert, sondern auch Programme benötigt werden, um die Updates den Geräten bereitzustellen. Zu diesen Programmen wird ein Update-Server gehören, vom dem die Geräte Auskunft erhalten, ob eine Firmwareaktualisierung für dieses Gerät bereitsteht.

Für den Entwurf der Prozesskette werden jedoch zuerst grundlegende Anforderungen benötigt, auf Basis derer der Prozess spezifiziert werden kann. Dabei muss auch auf die Einschränkungen eines IoT-Geräts Rücksicht genommen werden. Außerdem wird der Frage nachgegangen, ob die Firmware-Updates als inkrementelle Aktualisierungen oder kompletter Austausch der Firmware durchgeführt werden sollen. Von dieser Frage hängt auch der Entwurf eines geeigneten Dateiformats zum Transport der Update-Daten ab.

Aus dem konkreten Update-Prozess ergeben sich weitere Anforderungen an eine Implementierung, die dazu genutzt werden die Softwarearchitektur für einen Prototypen aufzubauen. An einer exemplarischen Implementierung können Tests durchgeführt werden, die überprüfen, ob die Anforderungen korrekt umgesetzt wurden. Für die Implementierung wurde ein ARM Cortex M4-Prozessor in Form eines Nucleo-F411-Evaluationsboards von STMicroelectronics ausgewählt. Als embedded OS ist die Wahl auf RIOT gefallen, da dieses frei verfügbar und auf das IoT spezialisiert ist. Außerdem ist die HAW Hamburg an der Entwicklung von RIOT beteiligt. Der im Prototypen eingesetzte Update-Server wird nur als Minimalbeispiel implementiert, da die Verwaltung der Update-Dateien für viele unterschiedliche Geräte, sowie die effiziente und skalierbare Programmierung des Servers, Probleme sind, die im Rahmen dieser Arbeit nicht hinreichend betrachtet werden können.

Da in dieser Arbeit nicht alle Aspekte eines Firmware-Update-Prozesses betrachtet werden können, müssen einige Annahmen getroffen werden. Dazu zählt, dass die Datenübertragung durch das Internet nicht im Detail betrachtet wird. Daher werden die verwendeten Übertragungsprotokolle als Blackbox betrachtet, die dafür sorgt, dass die Informationen vollständig und ohne Fehler beim Empfänger ankommen.

Der Firmware-Update-Prozess behandelt ebenfalls nicht, wie eine Firmware ihre Konfigurationsdaten zwischen den Versionen migriert, da es für dieses Problem keine universelle Lösung gibt.

2. Anforderungen an den Update-Prozess

Bevor ein konkreter Prozess für ein Firmware-Update entworfen werden kann, müssen die Anforderungen an diesen zusammengetragen werden. Die Fragestellung, wie Firmware-Updates über eine drahtlose Schnittstelle, also Over the Air (OTA), durchgeführt werden können ist jedoch nicht neu. Daher haben einige der gängigen Hersteller von Mikrocontrollern Application Notes oder Präsentationen veröffentlicht, die sich mit dem Thema eines sicheren Firmware-Updates auseinandersetzen.

Wenn in den Publikationen der Chip-Hersteller ein konkretes Verfahren beschrieben wird, ist jedoch meist eine serielle Kommunikationsschnittstelle notwendig, sodass sich eine Person in nächster Nähe zum Gerät befinden muss, um ein Firmware-Update durchzuführen (vgl. [STM12, TI15]). Da diese Arbeit jedoch OTA-Updates behandeln soll, können aus den Publikationen nur die allgemeinen Anforderungen an einen sicheren Firmware-Update-Prozess übernommen werden.

In den folgenden Abschnitten werden die Anforderungen an den Update-Prozess, sowie die zur Implementierung notwendige Hardware, ausgearbeitet. Im letzten Abschnitt werden einige Lösungen analysiert, die in der Industrie eingesetzt werden.

2.1. Grundlegendes Systemmodell für Firmware-Updates

In den folgenden Abschnitten werden die grundlegende Anforderungen an einen Firmware-Update-Prozess spezifiziert. Dafür wird zunächst ein verallgemeinertes System aus nur zwei Teilnehmern, die über das Internet miteinander kommunizieren können, betrachtet.

Wie in Abbildung 2.1 dargestellt, ist die Kommunikation bidirektional und die ausgetauschten Daten werden zunächst generisch „Update-Daten“ genannt. Die Teilnehmer werden als „Server“ und „IoT-Gerät“ bezeichnet.

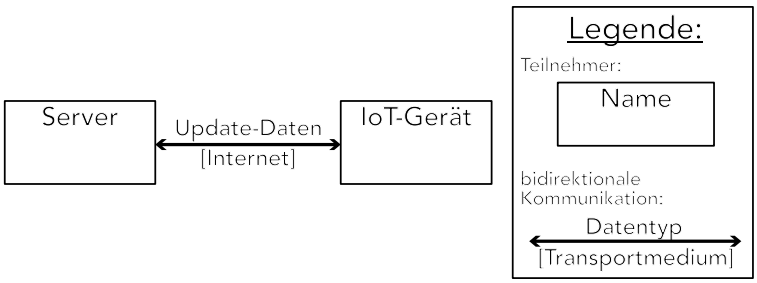

Abbildung 2.1.: grundlegendes Systemmodell für ein Firmware-Update

2.2. Charakterisierung eines IoT-Geräts

Das Internet of Things wird von der International Telecommunications Union (ITU) als ein Netzwerk aus physikalischen und virtuellen Dingen bezeichnet, welches fortgeschrittene Dienste für die Informationsgesellschaft ermöglicht. Ein Ding im Sinne des IoT kann dabei sowohl ein physikalisches, als auch ein virtuelles Objekt sein, sofern dieses dazu fähig ist, in Kommunikationsnetze eingebunden zu werden. Dabei werden physikalische Objekte auf eines oder mehrere virtuelle Objekte in der Informationswelt abgebildet.(vgl. [ITU12])

Diese Definition eines Dings im Sinne des IoT ist jedoch sehr allgemein, sodass sich daraus kaum Anforderungen an ein IoT-Gerät schließen lassen. Andererseits ist diese Unsicherheit auch eine Eigenschaft des IoT, da noch nicht bekannt ist, wie sich dieser Bereich des Internets entwickeln wird. Unbestritten ist jedoch, dass das Internet of Things dazu führen wird, dass in Zukunft deutlich mehr Geräte mit dem Internet verbunden sein werden. Die ITU spricht davon, dass die Anzahl der IoT-Geräte mindestens um eine Größenordnung höher sein wird, als die Zahl der Geräte, die zur Zeit mit dem Internet verbunden sind. Einzig und allein durch die Menge an Geräten, wird das IoT zu großen Herausforderungen im Bereich der IT-Sicherheit führen.

Im Gegensatz zum heutigen Internet, in dem der Großteil der Kommunikation von Menschen ausgelöst wird, zeichnet sich das IoT dadurch aus, dass hauptsächlich die verbundenen Geräte automatisch untereinander kommunizieren. Dafür ist es natürlich erforderlich, dass alle Geräte miteinander verbunden und auch global erreichbar sind. Die große Menge an Geräten führt hierbei zu einer größeren Diversität der mit dem Internet verbundenen Geräte, da diese ganz unterschiedliche Fähigkeiten haben werden.(vgl. [ITU12])

IoT-Anwendungen können beispielsweise intelligente Transportsysteme, neue intelligente Stromnetze (Smart Grid), E-Health oder die Automatisierung und Vernetzung der eigenen Wohnung (Smart Home) sein. In diesen Bereichen sind zwei grundsätzlich verschiedene Gerätekategorien zu unterscheiden: Einerseits werden generische Geräte benötigt, die Aufgaben aus dem IoT-Bereich neben ihren anderen Aufgaben ausführen. Dies sind beispielsweise Smartphones oder Heimelektronik, aber auch Industriemaschinen. Andererseits werden viele spezialisierte Sensoren und Aktoren benötigt, um beispielsweise ein Smart Home zu realisieren. Da diese Geräte auf eine spezifische Aufgabe spezialisiert sind, haben sie auch gänzlich andere Anforderungen, als die generischen IoT-Geräte.(vgl. [ITU12])

Der Firmware-Update-Prozess, der in dieser Arbeit entwickelt werden soll, wird vor allem auf die Sensor- und Aktor-Geräte Bezug nehmen, da diese sich nicht so aktualisieren lassen, wie beispielsweise ein Smartphone. In RFC 7228 [BEK14] wird diese besondere Klasse von IoT-Geräten beispielsweise als eingeschränkte Knoten (constrained nodes) bezeichnet, die besondere Kommunikationsnetze erfordern. Ein Gerät ist dabei nach RFC 7228 eingeschränkt, wenn aufgrund von physikalischen Beschränkungen oder Kostenbeschränkungen nicht alle zur Zeit als normal geltenden Anforderungen an einen Netzwerkknoten von dem eingeschränkten Knoten erfüllt werden können.

Beschränkungen ergeben sich dabei dadurch, dass mindestens einer der folgenden Punkte zutrifft:

1. Die Programmkomplexität ist durch den begrenzten Speicherplatz (ROM) stark eingeschränkt.
2. Die Größe von Zustands- und Pufferspeichern ist stark begrenzt (RAM).
3. Die zur Verfügung stehende Rechenleistung ist relativ gering.
4. Die zur Verfügung stehende Leistung ist stark begrenzt.
5. Die zur Verfügung stehende Energie ist stark begrenzt.

Zusätzlich ist zu beachten, dass IoT-Geräte, die als Aktoren oder Sensoren fungieren, im eingebauten Zustand in vielen Fällen nicht einfach erreichbar sind. Dies führt dazu, dass diese Geräte bis zum Lebensende möglichst keine Wartung erfordern sollten, sodass Updates vollautomatisch vorgenommen werden müssen, da diese sonst nie durchgeführt werden würden.

Ein weiterer Grund für vollautomatische Updates ist, dass IoT-Geräte meist keine eigene Benutzerschnittstelle haben, die komplexe Einstellungen, wie ein Update ermöglicht.

Auf der Basis dieser besonderen Merkmale von IoT-Geräten werden im nächsten Abschnitt die grundsätzlichen Anforderungen an den Update-Prozess zusammengetragen.

2.3. Grundsätzliche Anforderungen

In den vorherigen Abschnitten wurde ein grundlegendes System für Firmware-Update beschrieben und das Internet of Things charakterisiert. In den nächsten Abschnitten werden die Anforderungen an den Update-Prozess definiert, die anhand des grundlegendes Systems gestellt werden können.

Zunächst sollen jedoch die zwei grundlegendsten Anforderungen festgehalten werden:

R1 Der Update-Prozess muss die Möglichkeit bieten, die Firmware des IoT-Geräts zu aktualisieren.

R2 Das Firmware-Update soll über die in dem IoT-Gerät vorhandene Netzwerkschnittstelle übertragen werden.

Die weiteren Anforderungen sind in die Bereiche Robustheit, Sicherheit und Hardware-Anforderungen unterteilt.

2.3.1. Robustheit

Für den Kunden, bzw. Endanwender des IoT-Geräts ist es wichtig, dass durch ein Firmware-Update keine Nachteile für ihn entstehen. Dadurch ergeben sich die folgenden Anforderungen an die Robustheit des Update-Prozesses:

R3 Automatische Updates:
Ein Firmware-Update muss automatisch durchgeführt werden (vgl. [Tom14]).

R4 Verfügbarkeit des Geräts:
Ein Firmware-Update darf die Verfügbarkeit des Geräts nur minimal beeinträchtigen (vgl. [TF16, Tom14]).

R5 Einschränkung des Funktionsumfangs:
Der Funktionsumfang des Geräts darf durch ein Firmware-Update nicht beeinträchtigt werden (vgl. [TF16, Tom14]).

R6 Unzuverlässige Datenverbindung:
Ein Firmware-Update darf bei einer unzuverlässigen Datenverbindung keine inkonsistente Firmware auf dem Gerät hinterlassen (vgl. [Atm06, Atm15, TI15, KHV+16, STHS16]).

R7 Firmware-Updates passen zur Hardware:
Der Update-Prozess muss sicherstellen, dass ein Firmware-Update nur installiert wird, wenn dieses Update für das Gerät bestimmt ist (vgl. [Atm15, TI15, KHV+16, Min12]).

Vor allem bei IoT-Geräten ohne eigenes Administrationsinterface ist es wichtig, dass die Updates automatisch durchgeführt werden, damit sichergestellt ist, dass alle Geräte jeweils zeitnah die neueste Firmware installieren.

2.3.2. Sicherheit

Der Begriff Sicherheit wird im Deutschen für zwei Themen verwendet, die im Englischen mit unterschiedlichen Wörtern bezeichnet werden. Die Betriebssicherheit (im Englischen Safety) befasst sich mit dem Schutz der Umgebung vor dem betrachteten Objekt während die Angriffssicherheit (im Englischen Security) das Objekt vor der Umgebung schützen muss. Zur Betriebssicherheit können jedoch ohne ein konkretes Gerät keine Aussagen gemacht werden, weshalb dieser Bereich in dieser Arbeit nicht weiter betrachtet wird. Andererseits muss aber auch beachtet werden, dass Angriffe auf das Gerät häufig auch Auswirkungen auf die Betriebssicherheit haben. Daher können die beiden Aspekte der Sicherheit nie komplett getrennt voneinander betrachtet werden.

Trotzdem wird im Folgenden mit dem Begriff Sicherheit immer nur die Angriffssicherheit bezeichnet. Um die Anforderungen an die Sicherheit formulieren zu können, müssen zunächst die Schutzziele und die möglichen Angreifer charakterisiert werden.

2.4. Schutzziele und Charakterisierung des Angreifers

Die Schutzziele, die bei der Durchführung eines Firmware-Updates beachtet werden müssen, sind hauptsächlich die grundlegenden Schutzziele der IT-Sicherheit: Vertraulichkeit, Integrität und Verfügbarkeit, aber auch Authentizität. Die Verfügbarkeit des IoT-Geräts wurde schon im Kapitel zu den Anforderungen hinsichtlich der Robustheit als eine für den Nutzer besonders wichtige Eigenschaft herausgestellt. Um dies sicherzustellen muss der Update-Prozess so gestaltet sein, dass die Integrität und Authentizität der Update-Daten gewährleistet ist, damit keine schadhafte oder schädliche Software auf dem Geräte installiert werden kann.

In den Publikationen der Industrie, wie den Application Notes und Präsentationen der Chip-Hersteller, ist jedoch auffällig, dass zusätzlich besonderer Wert auf den Schutz des geistigen Eigentums, also auf die Vertraulichkeit der Daten, gelegt wird (vgl. [TI15, Ren12]). Dadurch ergibt sich die erste Angreifergruppe: Die Konkurrenz, die sich einen Wettbewerbsvorteil durch Kenntnis der Firmware verschaffen könnte.

Mit Kenntnis der Firmware können aber auch Plagiate erstellt werden, die dieselbe Firmware verwenden, wie das Originalgerät. Daher sollte der Update-Prozess verhindern, dass fremde Geräte Updates vom Originalhersteller erhalten.

Eine weitere Angreifergruppe wird aus verschiedenen Arten von böswilligen Hackern gebildet. In diesem Fall müssen hauptsächlich die Kunden, aber auch der Ruf des Herstellers, geschützt werden. Als Angriffsvektor auf den Update-Prozess bietet sich nur die Kommunikation zwischen dem Gerät und dem Update-Server an, um die Installation einer fremden Firmware zu erzielen. Eine manipulierte Firmware kann z. B. dazu genutzt werden ein Botnetz aufzubauen oder die vom Gerät erfassten Daten zu sammeln.

Zu der Gruppe der Hacker müssen auch Geheimdienste und andere staatliche Institutionen gezählt werden. Da diese jedoch nahezu unbegrenzte Ressourcen zur Verfügung haben, können ganz andere Angriffe durchgeführt werden, als von einer Firma oder Privatperson. Daher wird auf Geheimdienste in der weiteren Betrachtung nicht weiter eingegangen. Außerdem kann der Update-Prozess nicht gegen Angriffe schützen, die durch physikalischen Zugriff auf das Gerät möglich sind, da entsprechende Maßnahmen nur vom Herstellers des konkreten IoT-Geräts sinnvoll vorgenommen werden können. Dennoch werden einige Anforderungen zum Schutz vor Angriffen durch Zugriff auf die Hardware im nächsten Abschnitt (Kap. 2.5) mit aufgeführt.

Zusammenfassend ergeben sich folgende Anforderungen an den Update-Prozess, um die Sicherheit zu gewährleisten:

R8 Authentizität des Firmware-Updates:
Der Update-Prozess muss sicherstellen, dass nur Firmware-Updates installiert werden, die vom Originalhersteller ausgestellt wurden (vgl. [Atm06, Atm15, TI15, KHV+16, Sha11, Tom14, TF16]).

R9 Integrität des Firmware-Updates:
Der Update-Prozess muss sicherstellen, dass die Firmware-Daten bei der Übertragung nicht manipuliert oder verändert wurden (vgl. [Atm06, Atm15, TI15, KHV+16, Sha11, Tom14, TF16, STHS16]).

R10 Vertraulichkeit der Firmware:
Der Update-Prozess muss sicherstellen, dass die Firmware-Daten nicht von Dritten ausgelesen werden können (vgl. [Atm06, Atm15, TI15, KHV+16, TF16]).

R11 Firmware-Updates pro einzelnem Gerät:
Der Update-Prozess muss die Möglichkeit bieten Firmware-Updates auszustellen, die nur für ein Gerät gültig sind. Die Updates sollten in diesem Fall nur von diesem einen Gerät lesbar sein. (vgl. [Atm06, TI15, Tom14]).

R12 Integrität der installierten Firmware:
Der Update-Prozess muss sicherstellen, dass die installierte Firmware nicht manipuliert werden kann (vgl. [Min12]).

2.5. Anforderungen an die Hardware und Umwelt

Um ein hohes Maß an Sicherheit zu gewährleisten, muss der eingesetzte Mikrocontroller gewisse Anforderungen erfüllen:

R13 Debugging-Schnittstellen abgeschaltet:
Im Mikrocontroller müssen alle Debugging-Schnittstellen abgeschaltet sein (vgl. [KHV+16, Ren12]).

R14 Schutz vor Auslesen des Flashspeichers:
Der Flashspeicher des Mikrocontrollers darf über keine Schnittstelle lesbar sein. Davon ausgeschlossen ist der auf dem Flashspeicher vorliegende Programmcode selbst. Auch der Bootloader des Chipherstellers, sofern einer vorhanden ist, darf den Inhalt des Flashspeichers nicht lesen können. Eine Veränderung des Inhalts des Flashspeichers muss dazu führen, dass der komplette Flashspeicher gelöscht wird (vgl. [KHV+16, Ren12]).

R15 Watchdog:
Der Mikrocontroller muss dem Update-Prozess einen Watchdog-Timer zur Verfügung stellen (vgl. [Min12]).

R16 Interrupt-Vektortabelle verschieben:
Der Mikrocontroller muss dem Update-Prozess die Möglichkeit bieten die Position der Interrupt-Vektortabelle einzustellen.

Außerdem müssen die folgenden weiteren Anforderungen erfüllt werden, die nicht der Gerätehardware zuzuordnen sind:

R17 Update-Server:
Es muss einen im Netzwerk von allen IoT-Geräten erreichbaren Server geben, der den IoT-Geräten Auskunft gibt, ob ein Update für dieses Gerät verfügbar ist. Die Antwort des Servers muss, sofern ein Update für das anfragende Gerät verfügbar ist, angeben, an welcher Stelle das Update zum Download verfügbar ist. Dieser Server wird Update-Server genannt (vgl. [Tom14, TF16, Sha11]).

Die in den vorausgegangenen Kapiteln aufgestellten Anforderungen werden mit der Vorstellung des konkreten Update-Prozesses in Kapitel 3 noch ergänzt.

2.6. Lösungen der Industrie

Um einen Einblick in bestehende Lösungen der Industrie zu erlangen, können Patente, aber auch die Informationsmaterialien von kommerziell erhältlichen Lösungen analysiert werden. Die Marketing-Materialien bieten jedoch nur einen begrenzten Einblick in die Details der angebotenen Lösung und die von den Chip-Herstellern bereitgestellten Bootloader benötigen meistens eine serielle Kommunikationsschnittstelle.

Von STMicroelectronics gibt es beispielsweise eine Application Note mit dem vielversprechenden Titel „Secure Firmware Upgrade“ [STM12], welche jedoch die UART-Schnittstelle benutzt. Mocana[1] und SevenStax[2] bieten auf ihrer Webseite zwar Lösungen zum Firmware-Update an, Details sind jedoch nicht erhältlich ohne Kontakt mit dem Vertrieb aufzunehmen.

In Patenten gibt es recht wenige Verfahren, die auf embedded Systeme oder sogar IoT-Geräte spezialisiert sind. Von Bosch gibt es ein Patent zum Aktualisieren von Steuergeräten, das jedoch ein Aktualisierungsgateway voraussetzt, welches den Download, sowie den Update-Prozess verwaltet. Diese Methode ist sehr stark auf ein typisches System im Automobil angepasst.(vgl. [SH16])

Ein zweites Patent beschreibt ein Verfahren ohne Gateway und nennt explizit den Begriff Firmware Over the Air (FOTA) (vgl. [SSD16]). Da in diesem Verfahren aber zwingend ein externer Flashspeicher vorhanden sein muss, der über das verwendete Bussystem angeschlossen ist, kann auch dieses Patent nicht zur Hilfe genommen werden, um ein OTA-Update für IoT-Geräte zu entwickeln.

Ein Patent für ein Kabelmodem von Lakestar Semi Inc. verwendet wenigstens ein Gerät, welches recht nah am Anwendungsfall IoT ist. Dieses Verfahren ist jedoch dadurch eingeschränkt, dass es strenge Richtlinien gibt, wie Kabelmodems mit dem Kabelnetzbetreiber kommunizieren müssen.(vgl. [EM04])

Der Sensorikhersteller SICK hat ein Verfahren angemeldet, mit dem Firmware-Updates in komplexen Industrieanlagen durchgeführt werden können, ohne dass das Zusammenspiel der verschiedenen Komponenten gefährdet wird. Die Lösung besteht darin, dass die Sensorhardware und -Software auf zwei trennbaren Modulen realisiert werden. Die Anwendungssoftware und Konfigurationsdaten werden auf einem auswechselbaren Modul gespeichert. Mit einer PC-Software und einem entsprechenden Schreib- und Lesegerät können die Konfigurationsdaten in einem Modul angepasst werden oder es kann ein Update eingespielt werden. Dabei wird von der PC-Software sichergestellt, dass die Konfigurationsdaten immer zum verwendeten Softwarestand passen. Das Migrieren der Konfigurationsdaten muss also nicht von der Software im Sensor durchgeführt werden.(vgl. [GK08])

[1] https://www.mocana.com/iot-security/nanoupdate

[2] http://www.sevenstax.de/en/company/news/article/sicheres-firmware-update-fuer-embedded-systeme/?cHash=db9ba5c16b605a8ca09bf5910c86d3bb

Offensichtlich sind also schon ein paar Lösungen auf dem Markt verfügbar, diese stehen jedoch nicht alle offen zur Verfügung. Einige Lösungen erfordern auch spezielle Hardware, um beispielsweise den Schreibzugriff auf den Flashspeicher einzuschränken (vgl. [HP00, FIG. 1]).

In dieser Arbeit soll jedoch ein Prozess entwickelt werden, der prinzipiell auf vielen verschiedenen Mikrocontrollern eingesetzt werden kann, ohne dass diese sehr spezielle Anforderungen erfüllen müssen. Dieses Prinzip soll es vielen Herstellern ermöglichen ein sicheres Firmware-Update mit möglichst geringen Mehrkosten zu integrieren. Es muss im Vergleich zu einem IoT-Gerät ohne Update-Möglichkeit nur ein Mikrocontroller mit einem größeren Flashspeicher zur Verfügung stehen. Für das IoT ist Sicherheit sehr wichtig und obwohl ein Update-Mechanismus natürlich eine weitere potenzielle Schwachstelle ist, wird durch die Möglichkeit Updates durchzuführen ein größeres Problem gelöst, als neu entsteht. Wie in der Fachpresse zu lesen ist, sollten Geräte mit bekannten Schwachstellen nämlich nicht mehr verwendet und an das Internet angeschlossen werden, sofern keine Updates zur Verfügung stehen. Durch das open-source Betriebssystem RIOT als Basis für die Implementierung in Kapitel 4 kann der Update-Prozess auch ohne Budget für Lizenzen integriert werden.

3. Entwurf des Updateprozesses

In diesem Kapitel wird ein Firmware-Update-Prozess entwickelt, der die Anforderungen aus dem vorherigen Kapitel erfüllt. Dafür wird zunächst die Datenverarbeitungskette aufgebaut, die von den Bestandteilen des Update-Prozesses abgedeckt werden muss.

Um die Update-Daten speichern und transportieren zu können, muss außerdem ein Dateiformat entworfen werden, das in Abschnitt 3.3 spezifiziert wird.

Im darauffolgenden Abschnitt werden die Abläufe auf dem Gerät noch einmal genauer dargestellt und die Prozessschritte werden auf den Bootloader und die Anwendung aufgeteilt. Außerdem werden in diesem Abschnitt die Schnittstellen der verwendeten Software-Module entworfen.

Aus der Spezifikation der Update-Prozesskette ergeben sich weitere Anforderungen an die Implementierung, die im letzten Abschnitt des Entwurfs (Kap. 3.6) beschreiben werden.

3.1. Entwurf der Prozesskette

Um einen Update-Prozess für ein IoT-Gerät entwerfen zu können, muss zunächst der grundlegende Datenfluss ermittelt werden. Die Prozesskette beginnt mit dem Quellcode auf dem Computer des Entwicklers und endet mit der Ausführung des neuen Anwendungscodes. Jeder Prozessschritt erhält bestimmte Daten als Eingabe und generiert Daten am Ausgang, die zum Teil ein anderes Datenformat haben, als die Eingabedaten.

Die Anforderung, dass nur zum Gerät passende Firmware-Updates installiert werden dürfen (**R7**), kann nur erfüllt werden, indem zusätzlich zum Maschinencode (Binärdaten) noch Metadaten transportiert werden. Welche Informationen diese Metadaten enthalten müssen wird, im nächsten Abschnitt (Kap. 3.3) hergeleitet.

Die Metadaten allein sind jedoch nicht ausreichend, um die Anforderungen an die Sicherheit zu befriedigen. Daher wird zum Transport der Update-Daten ein spezielles Dateiformat verwendet, welches ebenfalls im nächsten Abschnitt spezifiziert wird. Für die weitere Betrachtung des Updateprozesses ist es jedoch ausreichend zu wissen, dass zur Erstellung der Update-Datei ein weiterer Prozessschritt notwendig ist, der Signieren genannt wird.

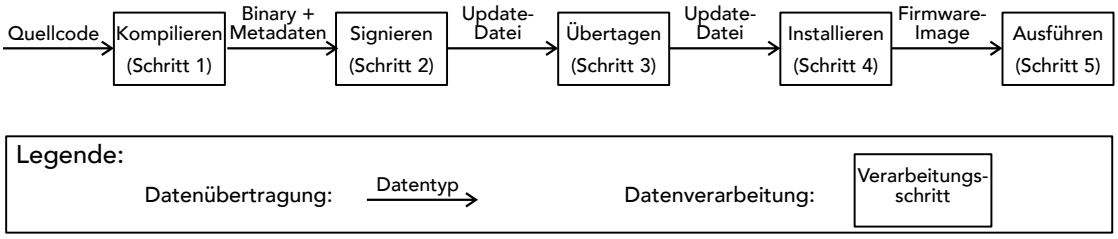

Abbildung 3.1.: Datenfluss der Prozesskette

Wie in Abbildung 3.1 dargestellt, wird der Quellcode der Anwendung zunächst kompiliert (Schritt 1). Dies geschieht in der Regel auf dem Computer des Entwicklers. Bei einem komplexen Prozess, wie es die Erstellung von Update-Dateien ist, sollte dafür ein Build-Script verwendet werden, in dem auch weitere Elemente, wie der Inhalt einzelner Metadaten-Felder, angegeben werden können.

Im nächsten Schritt wird die Update-Datei erstellt, indem die Binärdaten und die Metadaten signiert werden. Dies kann ebenfalls auf dem Rechner des Entwicklers ausgeführt werden. Für Updates, die an Geräte im Feld ausgeliefert werden sollen, ist ein zentraler Signaturserver jedoch vorzuziehen, da der Signaturprozess geheime kryptographische Schlüssel verwendet, die sensibel behandelt werden müssen. Aus ebendiesem Grund ist das Signieren in einem gesonderten Schritt aufgeführt und nicht mit dem Kompilieren zusammengefasst.

Schritt 3 ist dafür zuständig die Update-Datei an das Gerät zu übertragen, sodass in Schritt 4 die Datei installiert werden kann. Nach der Installation liegt im Flashspeicher des Geräts ein sogenanntes Firmware-Image vor, welches nach einem Neustart ausgeführt wird.

Welche Schritte die Installation umfasst, wird in Kapitel 3.4 behandelt. Dafür ist jedoch zunächst die Spezifikation des Update-Dateiformats notwendig, die auch das Firmware-Image definiert.

3.2. Vollständige und inkrementelle Updates

Bevor das Update-Dateiformat spezifiziert werden kann, muss evaluiert werden, ob inkrementelle Updates möglich sein sollen, oder jeweils nur die vollständige Firmware ausgetauscht werden kann.

Inkrementelle Updates zeichnen sich dadurch aus, dass nur die Änderungen am bestehenden Programm übermittelt werden. Der Vorteil dieser Methode besteht darin, dass in der Regel deutlich weniger Daten übertragen werden müssen. Für einen Betriebssystem-Patch in Windows oder Linux muss beispielsweise auch nicht das gesamte Betriebssystem noch einmal heruntergeladen werden und für einige Aktualisierungen ist noch nicht einmal ein Neustart notwendig.

Dieses Prinzip lässt sich auch auf einzelne Programme anwenden, ist dafür aber auch mit einem größeren Aufwand im Updater verbunden, da ein Dateiformat interpretiert werden muss, dass differentielle Änderungen darstellt. Der Quellcode eines Programms, welches in einer interpretierten Sprache geschrieben wurde, lässt sich dafür sogar im laufenden Betrieb inkrementell aktualisieren.

Bei kompiliertem Code müssen die Patches jedoch im Maschinencode eingefügt werden, sodass inkrementelle Updates zwar möglich sind, jedoch einen großen Aufwand erfordern. Da der Maschinencode auf Mikrocontrollern meist in einem Flashspeicher abgelegt ist, können aufgrund der Funktionsweise der Flash-Technologie nicht einfach Zeilen in den bestehenden Code eingeschoben werden. Außerdem können einzelne Bits jeweils nur in einer Richtung, beispielsweise von 1 auf 0, geschrieben werden. Für die umgekehrte Richtung muss ein großer Flash-Speicherblock auf einmal gelöscht werden. Änderungen an bestehenden Daten im Flashspeicher sind also nur sehr eingeschränkt möglich.

Eine Möglichkeit einen Patch einzuspielen wäre jedoch, dass eine Sprunganweisung so modifiziert wird, dass in einen Speicherbereich gesprungen wird, der in einem noch unbeschriebenen Bereich des Flashspeichers liegt. In diesem Bereich kann dann auch nachträglich Code auf den Flashspeicher geschrieben werden, ohne einen ganzen Speicherblock löschen zu müssen. Dieses Verfahren ist jedoch nicht mit einem einfachen Compiler möglich, sondern erfordert wahrscheinlich eine manuelle Programmierung in Assembler.

Inkrementelle Updates sind also für kompilierte Programme zwar möglich, jedoch sehr aufwendig in der Umsetzung. Wenn dann noch beachtet wird, dass IoT-Geräte vermutlich ein embedded Betriebssystem einsetzen, kann ein inkrementelles Update nach

dem zuvor beschriebenen Schema als unmöglich angesehen werden, da schon eine neue Betriebssystem-Version zu viele Änderungen hervorrufen wird.

Obwohl kompilierte Sprachen vor allem auf Mikrocontrollern, die für Sensornetzwerke eingesetzt werden, den Vorrang haben, wurde in einer Veröffentlichung aus 2009 [SC09] betrachtet, ob der Einsatz interpretierter Sprachen in diesen Geräten energetisch sinnvoll ist. Drahtlose Kommunikation benötigt im Vergleich zum Betrieb des Mikrocontrollers nämlich sehr viel Energie. Der Energiebedarf für die Übertragung eines Updates könnte jedoch eingeschränkt werden, wenn nicht jedes Mal die gesamte Firmware übertragen werden müsste. Andererseits benötigt eine interpretierte Sprache eine Laufzeitumgebung, die wiederum ineffizienter arbeitet, als kompilierter Maschinencode. In der Publikation wurde der Overhead des Java-Bytecode Interpreters Maté für eine einfache Schleife, sowie das Senden eines Netzwerkpakets gemessen. Der Faktor, um den die Ausführung des Bytecodes länger als der Maschinencode braucht, entspricht auch dem zusätzlichen Energiebedarf. Weitere Messungen der Autoren haben ergeben, dass für die Übertragung eines Bytes ausgeführten Codes ca. 10.000 Mal so viel Energie benötigt wird, wie für die Ausführung des Codes selbst. Deshalb sollte die Größe eines Updates besonders klein sein.

Der in der Publikation [SC09] verwendete Bytecode ist um einen Faktor einhundert bis vierhundert kleiner, als der vergleichbare Maschinencode, hat aber auch einen ca. 30-fachen Overhead bei der Ausführung. Um zu berechnen, ab welchem Zeitpunkt der Bytecode effizienter ist, als der Maschinencode, ist die absolute Größe des Codes nicht entscheidend. Der Zeitpunkt ist nur abhängig von dem Overhead des Bytecode bei der Ausführung und dem höheren Energiebedarf des Maschinencodes bei der Übertragung. Bei einer Reduktion der Code-Größe auf ein Zehntel und einem Overhead durch die Laufzeitumgebung von Faktor 30 kann der Bytecode beispielsweise 300 Mal ausgeführt werden bis der Maschinencode energieeffizienter ist. Dies bedeutet, dass der Einsatz interpretierter Sprachen in diesem Beispiel energetisch nur sinnvoll ist, wenn sehr häufig Updates durchgeführt werden. Vermutlich werden IoT-Geräte jedoch deutlich weniger häufig aktualisiert, sodass ohne weitere Bedenken jeweils die gesamte Firmware aktualisiert werden. Vor allem auch, da interpretierte Sprachen auf IoT-Geräten noch nicht weit verbreitet sind.

3.3. Format der Firmware-Update-Datei

Zum Transport der Firmware-Update-Daten wird ein spezielles Dateiformat benötigt, damit die Sicherheitsanforderungen an den Update-Prozess eingehalten werden können.

Wie schon im letzten Abschnitt erwähnt, müssen für ein sicheres und robustes Firmware-Update neben den Binärdaten noch Metadaten übertragen werden. Um die Authentizität und Integrität der Update-Daten sicherzustellen (Requirements **R8** und **R9**) werden außerdem digitale Signaturen benötigt. Die Vertraulichkeit der Daten (Req. **R10**) wird dadurch sichergestellt, dass diese verschlüsselt übertragen werden.

Da dieses Problem jedoch nicht neu ist, wurde schon vor über zehn Jahren ein Dateiformat unter RFC 4108 [Hou05] veröffentlicht, welches ein Firmware-Paket sicher übertragen soll. Eine Verschlüsselung der Daten zum Schutz vor Veröffentlichung ist optional ebenfalls vorgesehen. Das verwendete Dateischema basiert dabei auf der Cryptographic Message Syntax aus RFC 3852 [Hou04], die eine ASN.1-formatierte Datei mithilfe von digitalen Signaturen und Zertifikaten absichert.

Durch die ASN.1-Syntax ist dieses Dateiformat zwar in Bezug auf die transportierten Informationen sehr flexibel, dafür ist das Parsen der Informationen jedoch aufwendiger, als eine fixe Datenstruktur. Außerdem müssen für die Signaturen Zertifikatsketten überprüft werden, wofür unter Umständen weitere Zertifikate aus Quellen im Netzwerk nachgeladen werden müssen. Daher wurde dieses Dateiformat als zu komplex eingestuft.

Da es noch keinen Industriestandard für die Übertragung von OTA-Updates gibt, wird ein eigenes Dateiformat entwickelt, das einfach und dadurch auch schnell und energiesparend verarbeitet werden kann, dafür aber auch nur eine spezielle Aufgabe erfüllt.

Die Datei muss die Firmware in Binärform, sowie einige Metadaten transportieren, die mit einer Signatur abgesichert und durch Verschlüsselung vor unberechtigtem Auslesen geschützt sind. Die Metadaten müssen jedoch auch dem Bootloader zur Verfügung stehen, sodass diese bei der Installation zusammen mit der Firmware im Flashspeicher abgelegt werden müssen.

Des Weiteren ist zu beachten, dass sich allein durch die Verwendung eines eigenen Bootloaders die Startadresse der Anwendung verschiebt. Um eine Anwendung an einer anderen Stelle im Flashspeicher als dessen Basisadresse starten zu können, muss dem Interruptcontroller die neue Adresse der Vektortabelle mitgeteilt werden. In der ARM-v6-M-

und ARM-v7-M-Architektur ist dafür das Vector Table Offset Register (VTOR) vorgesehen. Die Verschiebung darf dabei aber nicht beliebig sein, sondern muss einer Staffelung folgen. Für den Cortex-M0+ sind dies beispielsweise Vielfache von 256 Byte [STM14, Kap. 2.3.4] und für die Mikrocontroller-Familien Cortex-M3, -M4 und -M7 muss die Vektortabelle in einem Raster von 512 Byte [STM16, STM13, STM17] ausgerichtet sein. Dies bedeutet jedoch auch, dass für den gesamten Header mit den Metadaten ebendieser Platz mindestens zur Verfügung steht. Wenn die Informationen weniger Speicherplatz benötigen, muss der Rest entsprechend aufgefüllt werden.

Damit die Metadaten an einer bekannten Speicheradresse zur Verfügung stehen, sollten diese vor dem eigentlichen Anwendungscode abgelegt werden. Dadurch verschiebt sich die Startadresse der Anwendung zwar ein weiteres Mal, dies ist jedoch im Rahmen der oben genannten Staffelung ohne große Probleme möglich.

3.3.1. Schematischer Aufbau der Datei

Die Update-Datei besteht aus mehreren Schichten, die jeweils Informationen zum Anfang der Datei hinzufügen. Dadurch, dass die Größenangaben sich jeweils auf die Startadresse der Binärdaten beziehen, ist das Dateiformat größtenteils unabhängig von der konkreten Größe der VTOR-Staffelung.

Um für zukünftige Erweiterungen, vor allem im Bereich der Metadaten, vorbereitet zu sein, wurde der Platz für die Daten der einzelnen Schichten jeweils auf Vielfache von 32 Byte aufgerundet. Im Folgenden werden immer die reservierten Speicherbereiche beschrieben, während die konkreten Datenstrukturen in den nachfolgenden Kapiteln beschrieben werden.

Wie in Abbildung 3.2 dargestellt, besteht der Kern der Update-Datei aus den Binärdaten und den Metadaten. Die Länge der Binärdaten ist selbstverständlich variabel während für die Metadaten 64 Byte zur Verfügung stehen.

Die Datei wird in einem zweiten Schritt um die Firmware-(Image-)Signatur ergänzt, welche die noch unverschlüsselten Binärdaten und die Metadaten absichert. Hierfür stehen 64 Byte zur Verfügung. Die Metadaten und die Firmware-Signatur bilden den Firmware-(Image-)Header, der vom Bootloader benötigt wird, um die Integrität und Authentizität der installierten Firmware zu überprüfen.

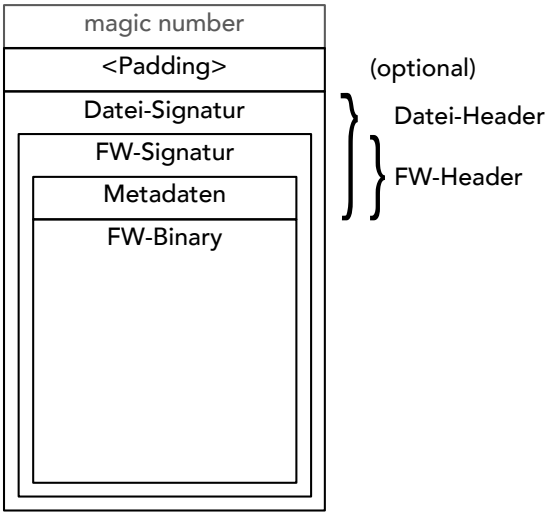

Abbildung 3.2.: schematischer Aufbau des Dateiformats

Firmware-Header und Binärdaten bilden das Firmware-Image, welches nach der Installation des Updates im Flashspeicher des Mikrocontrollers vorliegt.

Mit diesen Maßnahmen sind jedoch nur die Authentizität und Integrität gesichert (Requirements **R8** und **R9**). Damit die Binärdaten vertraulich bleiben (Req. **R10**), muss diese Sektion verschlüsselt werden. Dafür werden die Binärdaten in der Update-Datei gegen eine verschlüsselte Version getauscht. Je nach verwendeter Verschlüsselungsmethode kann sich die Länge der Binärdaten erhöhen, dies ist jedoch für das Dateiformat nicht weiter relevant.

Um die Authentizität und Integrität der Datei vor dem Entschlüsseln der Binärdaten prüfen zu können, wird eine zweite Signatur gebildet, die die Firmware-Signatur, Metadaten und Binärdaten absichert. Für diese Dateisignatur stehen 128 Byte zur Verfügung, sodass der Datei-Header insgesamt eine Größe von 256 Byte hat.

Da die Größe des Headers der im Firmware-Slot abgelegten Firmware immer ein ganzzahliges Vielfaches der VTOR-Staffelung sein muss, wird gegebenenfalls ein Padding am Anfang des Firmware-Images benötigt. Wenn der Installationsprozess das benötigte Padding selbstständig hinzufügt, muss das Padding nicht in der Datei enthalten sein. Für eine simplere Installationsroutine, kann die Datei jedoch das Padding schon enthalten. Die konkrete Bitfolge, die zum Auffüllen verwendet wird, ist dabei nicht relevant und wird daher an nicht spezifiziert.

Damit die Update-Datei identifiziert werden kann, wird eine Magic Number an den Anfang der Datei geschrieben. Diese wird nicht von der Länge des eventuell vorhandenen Paddings abgezogen und beeinflusst alle weiteren Längen und Ausrichtungen nicht. Der Wert der Magic Number entspricht der Repräsentation des ASCII-Strings `RIOTFW01` (`0x52 0x49 0x4f 0x54 0x46 0x57 0x30 0x31`). Über den Zähler am Ende der Magic Number können in Zukunft bei Bedarf verschiedene Versionen und Revisionen der Update-Datei unterschieden werden. Die ersten sechs Stellen wurden so gewählt, dass keine Kollisionen mit anderen Dateiformaten auftreten.

Die Metadaten werden in der Update-Datei nicht verschlüsselt, damit diese an jeder Stelle der Prozesskette lesbar sind. Da das Signieren und Bereitstellen der Update-Dateien nicht unbedingt auf demselben Computer durchgeführt wird, müssen die Update-Dateien zwischen diesen beiden Computern ausgetauscht werden. Da der Update-Server zum korrekten Ausliefern der Updates jedoch Informationen aus den Metadaten benötigt, wäre ein begleitender Datensatz notwendig, wenn die Metadaten ebenfalls verschlüsselt wären. Dieser Datensatz wäre jedoch eine weitere Fehlerquelle, sodass frei lesbare Metadaten bevorzugt wurden.

Die zweite Signatur wurde gewählt, damit das IoT-Gerät die Authentizität und Integrität der Datei überprüfen kann, ohne alle Daten vorher zu entschlüsseln, da die Entschlüsselung in der Regel aufwendiger zu berechnen ist, als das Prüfen einer Signatur. Außerdem können in der Dateisignatur weitere Informationen sicher transportiert werden, die je nach Algorithmus für die Entschlüsselung benötigt werden.

3.3.2. Firmware-Metadaten

In den Metadaten werden folgende Informationen gespeichert:

Beschreibung	**ID**	**Länge**	**Datenformat oder Inhalt**
Magic Number	`magic`	4 Byte	ASCII `OTA1` (`0x4f 0x54 0x41 0x31`)
Chip Seriennummer	`chip_id`	16 Byte	freies Datenformat
Firmware-Version	`fw_vers`	2 Byte	16 Bit unsigned integer
Firmware-Basisadresse	`fw_base_addr`	4 Byte	32 Bit unsigned integer
Länge der Binärdaten	`size`	4 Byte	32 Bit unsigned integer

Tabelle 3.1.: Firmware-Metadaten: Datenformat der Metadatenfelder

Die Magic-Number wird benötigt, um bei einer auf dem Flashspeicher installierten Firmware zu erkennen, dass die Metadaten belegt sind. Auch in diesem Fall ist, wie schon bei der Update-Datei, eine Ziffer dafür vorgesehen zukünftige inkompatible Versionen unterscheiden zu können.

Die Hardware-ID kann vom Hersteller des Geräts frei vergeben werden und soll die Hardwarekonfiguration des Geräts identifizieren. Damit soll sichergestellt werden, dass die Firmware auch mit der durch die Hardware-ID identifizierten Hardware kompatibel und damit lauffähig ist (Req. **R7**). Diese Nummer kann beispielsweise eine Artikelnummer mit einer Revisionsnummer sein, die geändert wird, sofern durch Änderungen an der Hardware eine andere Software erforderlich ist.

In einigen Branchen kann es erforderlich sein, dass dem Hersteller bekannt sein muss, auf welchem spezifischen Gerät welche Firmware installiert ist (Req. **R11**). Zu diesem Zweck kann in dem Metadatenfeld `chip_id` die Seriennummer des Geräts eingetragen werden auf dem das Update installiert werden darf. Da viele Mikrocontroller bei ihrer Produktion schon eine Seriennummer zugewiesen bekommen, die in einem spezifischen Register verfügbar ist, sollte diese Chip-Seriennummer verwendet werden. Bei den STM32-Mikrocontrollern ist die Seriennummer 96 Bit lang, das Metadatenfeld wurde jedoch vorsorglich auf 128 Bit ausgelegt.

Um zu erkennen, ob die Firmware in der Update-Datei aktueller ist, als die Installierte, wird die Firmware-Versionsnummer ebenfalls in den Metadaten angegeben. Dafür muss jedoch beim Erstellen der Update-Dateien darauf geachtet werden, dass die Firmware-Versionsnummer immer streng aufsteigend vergeben wird.

Des Weiteren ist es wichtig anzugeben für welche Basisadresse (`fw_base_addr`) die Anwendung kompiliert (bzw. gelinkt) wurde. Dies ist gleichzeitig die Basisadresse der Vektortabelle und damit, wie im vorherigen Abschnitt (Kap. 3.3.1) beschrieben, der Fixpunkt, um den das Dateiformat aufgebaut ist. Diese Information wird verwendet, um das Update an die korrekte Stelle im Flashspeicher zu schreiben.

Zuletzt wird die Länge der unverschlüsselten Binärdaten angegeben, da auf dem Flashspeicher des Mikrocontrollers kein Dateisystem vorhanden ist, welches die Gesamtlänge der Datei bereitstellen würde. Die Länge wird dabei sowohl für das korrekte Entschlüsseln der Daten, als auch für die Überprüfung der Signaturen benötigt.

Im Codesegment 3.1 ist eine beispielhafte Implementierung der benötigten Datenstruktur in C dargestellt. Hierbei ist zu beachten, dass das `struct` 2 Byte mehr Speicherbedarf

haben kann, sofern eine 4 Byte Speicherausrichtung im Linkerscript vorgegeben wurde. Um diese Lücken zu vermeiden, kann ein `packed struct` verwendet werden, dies ist jedoch nicht zwingend notwendig, wenn alle Programme, die die Metadaten schreiben oder lesen, dieselbe Speicherausrichtung verwenden.

```
typedef struct OTA_FW_metadata_t {
    uint32_t magic;
    uint8_t  hw_id[8];
    uint8_t  chip_id[16];
    uint16_t fw_vers;
    uint32_t fw_base_addr;
    uint32_t size;
} OTA_FW_metadata_t;
```

Code 3.1: C-Implementierung der Metadaten (Beispiel)

3.3.3. Verschlüsselung

Wie zuvor erwähnt, müssen die Daten der Binärdaten-Sektion verschlüsselt werden, um die Vertraulichkeit zu gewährleisten. Dafür wird ein symmetrisches Verschlüsselungsverfahren eingesetzt.

In der Kryptographie wird zwischen symmetrischen und asymmetrischen Verschlüsselungsverfahren unterschieden. Bei symmetrischen Verfahren wird derselbe Schlüssel zum Entschlüsseln verwendet, wie zum Verschlüsseln. Je nachdem, welche Menge von Daten auf ein Mal verschlüsselt werden kann, wird zwischen Block- und Stromchiffren unterschieden. Während Stromchiffren einen Datenstrom beliebiger Länge verschlüsseln können, muss in eine Blockchiffre immer ein Datenblock mit einer vorgegebenen Länge eingegeben werden.

Asymmetrische Verfahren verwenden ein Schlüsselpaar bestehend aus einem privaten und einem öffentlichen Schlüssel. Dadurch lassen sich vertrauliche Informationen an nur einen Empfänger übertragen, indem mit dem öffentlichen Schlüssel des Empfängers verschlüsselt wird. Umgekehrt kann bei Verschlüsselung mit dem privaten Schlüssel jeder Dritte sicher sein, dass die Nachricht von einem bestimmten Absender verschlüsselt wurde. Natürlich muss dazu der private Schlüssel geheim gehalten werden.

Da asymmetrische Verschlüsselungsverfahren deutlich langsamer, als symmetrische Verfahren sind, wird zur Verschlüsselung der Binärdaten ein symmetrisches Verfahren ein-

gesetzt. Welches Verschlüsselungsverfahren konkret in der Update-Datei eingesetzt wird ist für das Dateiformat an sich nicht relevant. Für die weitere Betrachtung wurde jedoch die Blockchiffre AES-128 im Cipher Block Chaining (CBC) Betriebsmodus ausgewählt, da dieses Verfahren bewährt ist, häufig eingesetzt wird und damit auch vergleichsweise gut getestet ist. Außerdem gibt es in einigen Mikrocontrollern ein Hardware-Modul zur AES-Entschlüsselung, welches deutlich effizienter arbeiten kann, als eine Software-Implementierung.

Für den CBC-Modus wird neben dem Schlüssel noch ein sogenannter Initialisierungsvektor (IV) benötigt. Dieser muss für jede Verschlüsselung zufällig generiert werden und wird auch vom Empfänger zum Entschlüsseln benötigt. Da die verwendeten Signaturen die Möglichkeit bieten beliebige Daten sicher zum Empfänger zu transportieren, können Initialisierungsvektor und Schlüssel in der Signatur sicher übertragen werden. Der AES-Schlüssel wird jedes Mal neu generiert, da es keine Vorteile gibt diesen neben den Schlüsseln für die Signaturen ebenfalls vorab auf dem Gerät zu hinterlegen. Falls jedoch ein AES-Schlüssel bekannt werden sollte, könnten bei vorinstallierten Schlüsseln vergangene und zukünftige Update-Dateien ebenfalls entschlüsselt werden. Neu generierte Schlüssel haben außerdem den Vorteil, dass diese Schlüssel nicht zusätzlich verwaltet werden müssen.

Die Betriebsart CBC und eine Schlüssellänge von 128 Bit werden in der Technischen Richtlinie „Kryptografische Algorithmen und Schlüssellängen“ des BSI [BSI17] unter den empfohlenen Verfahren aufgelistet. Auch wenn die verwendete Schlüssellänge nur die Mindestanforderung erfüllt, wird dies als ausreichend angesehen, da keine hochsensiblen Informationen geschützt werden. Außerdem ist der Wert eines geknackten Schlüssels relativ gering, wenn für jede Update-Datei ein neuer Schlüssel verwendet wird.

3.3.4. Signaturen

Für die Signaturen wird ein asymmetrisches Verschlüsselungsverfahren eingesetzt. Um eine Signatur zu bilden, wird ein kryptografischer Hash über die zu signierenden Daten berechnet, der mit dem privaten Schlüssel des Senders verschlüsselt wird. Der Empfänger kann daraufhin die Signatur mit dem öffentlichen Schlüssel des Senders entschlüsseln und ebenfalls den Hashwert der zu signierenden Daten berechnen. Wenn der berechnete und der entschlüsselte Hashwert übereinstimmen, wurde die Nachricht nicht manipuliert und der Absender wurde gleichzeitig authentifiziert.

Zur Berechnung des Hashwerts wird SHA-256 eingesetzt. Dieses Verfahren ist weit verbreitet und gehört zu den empfohlenen Hashfunktionen des BSI (vgl. [BSI17]). Wie für AES gibt es Mikrocontroller mit Kryptografie-Peripherie, die einen SHA-256 Hash deutlich effizienter und damit schneller als eine Software-Implementierung berechnen kann. Außerdem ist dieses Verfahren in RIOT enthalten.

Microsoft beschreibt in einem Patent [Chi06] ein Verfahren zur Erkennung von inkonsistenter oder alter Firmware und wie die Firmware dann ersetzt werden kann. Dafür soll der sogenannte Boot-Code die Firmware teilweise oder vollständig durch eine Austausch-Firmware ersetzen, die auf dem Gerät selbst oder auf einem Speicher im selben Netzwerk gespeichert ist. Eine inkonsistente Firmware wird dadurch erkannt, dass eine Prüfsumme über die Firmware errechnet und mit einem vorher gespeicherten Wert verglichen wird. Für die Prüfsumme wird explizit ein CRC-Wert erwähnt, es sind jedoch auch jegliche andere Gültigkeitsprüfungen erlaubt. Ein CRC-Wert ist jedoch keine kryptografisch sichere Hashfunktion und sollte daher nicht zur Sicherung der Datenintegrität, insbesondere bei größeren Datenmengen, verwendet werden.

Um die Signatur zu vervollständigen, muss der Hashwert verschlüsselt werden. Ein bekannter asymmetrischer Verschlüsselungsalgorithmus ist RSA. Bei dessen Verwendung wird vom BSI eine Schlüssellänge von mindestens 2000 Bit empfohlen (vgl. [BSI17, Tab. 3.1]). Dies bedeutet jedoch auch, dass die Signatur deutlich länger ist, als die Daten, die diese schützen soll.

Als Alternative bietet sich die NaCl-Bibliothek[1] an, die Verfahren speziell für Systeme mit kleinem Speicher bereitstellt. Die in dieser Bibliothek verwendeten kryptografischen Verfahren wurden nach aktuellen Empfehlungen und in Hinblick auf hohen Datendurchsatz ausgewählt [BLS12]. Dies bedeutet auch, dass die Verfahren sehr effizient arbeiten.

Da NaCl zum Teil mehrere verschiedene Implementierungen desselben Algorithmus enthält, um die effizienteste Implementierung für die beim Installieren der Bibliothek verwendete Prozessorarchitektur auszuwählen, lässt sich diese Bibliothek nicht einfach in ein Projekt integrieren. Für diesen Fall wird jedoch die TweetNaCl-Bibliothek[2] bereitgestellt, die den Inhalt von NaCl auf eine Header- und eine Source-Datei reduziert. Diese Bibliothek ist außerdem als offizielles Paket in RIOT enthalten.

[1] https://nacl.cr.yp.to
[2] https://tweetnacl.cr.yp.to

Als Signatur für die Update-Datei wird die „Public-key authenticated encryption“ mit dem Namen `crypto_box` verwendet. Die Besonderheit an der `crypto_box` ist, dass sowohl der private Schlüssel des Senders, als auch der öffentliche Schlüssel des Empfängers zum Verschlüsseln verwendet werden. Sofern jedes Gerät bei der Produktion seinen eigenen privaten Schlüssel erhält, können also Signaturen gebildet werden, die von nur einem Gerät entschlüsselt werden können. Dies ermöglicht eine weitere Sicherheitsstufe neben der Angabe der Chip-Seriennummer, wenn Updates jeweils nur auf einem Gerät installierbar sein sollen.

Die Schlüssel sind jeweils 32 Byte lang, also deutlich kürzer als bei RSA. Da jedoch elliptische Kurven die Basis der kryptografischen Algorithmen bilden, ist für das vom BSI empfohlene Sicherheitsniveau nur eine Schlüssellänge von mindestens 250 Bit erforderlich (vgl. [BSI17, Tab. 3.1, S. 29]). Dies ist durch eine Schlüssellänge von 32 Byte gegeben. Ein weiterer Vorteil der `crypto_box` gegenüber RSA ist, dass die Signatur nur 32 Byte länger ist, als die zu verschlüsselnden Daten, und damit in den meisten Fällen einen deutlich geringeren Overhead als RSA aufweist.

Die Firmware-Signatur füllt den vorgesehenen Platz von 64 Byte aus, da die Signatur nur einen SHA-256-Wert mit einer Länge von 32 Byte transportiert.

Für die Dateisignatur wurde mehr Platz, nämlich 128 Byte, vorgesehen, damit diese Signatur noch weitere Informationen, wie einen Initialisierungsvektor oder Nonce für die symmetrische Verschlüsselung aufnehmen kann. Bei Verwendung von AES-128 und SHA-256 werden jeweils 16 Byte für den AES-Schlüssel und Initialisierungsvektor, sowie 32 Byte für den Hashwert benötigt, sodass die Signatur 96 Byte lang ist.

3.4. Prozesskette auf dem Gerät

In Kapitel 3.1 wurde die Update-Prozesskette beschrieben ohne auf die erforderlichen Schritte auf dem IoT-Gerät genauer einzugehen. In diesem Kapitel wird nun der Installationsprozess auf dem Gerät genauer ausgestaltet. Damit wird vor allem der Prozessschritt 4 aus Abbildung 3.1 genauer definiert.

In der Einleitung wurde schon erwähnt, dass ein Bootloader notwendig ist, um den Update-Prozess realisieren zu können. Im folgenden Abschnitt wird zunächst erläutert, wie ein Cortex-M-Mikrocontroller startet, warum sich daraus ergibt, dass ein Bootloader benötigt wird und wie die Anwendung im Flashspeicher abgelegt wird. Daraus ergibt

sich ein erster Ansatz für den Update-Prozess auf dem Gerät. Im darauffolgenden Abschnitt wird dieser Ansatz in Hinblick auf die Aufgabenteilung zwischen Anwendung und Bootloader überarbeitet und detaillierter beschrieben.

3.4.1. Bootloader, Anwendung und Firmware-Slots

Ein ARM Cortex-M Mikrocontroller startet aus dem Reset, indem der Reset Handler aus der Interruptvektortabelle ausgeführt wird. Die Speicheradresse des Reset Handlers steht dabei immer an einem Offset von 4 Byte hinter der Basisadresse der Vektortabelle, die im Standardzustand am Anfang des Flashspeichers liegt. Die Position der Vektortabelle kann bei einigen Mikrocontrollern über den Pegel an einem speziellen Pin auch in den RAM verschoben werden. In den meisten Cortex-M-Architekturen ist es jedoch auch möglich die Basisadresse der Vektortabelle mithilfe des sogenannten VTOR-Registers aus dem Programmcode heraus zu verschieben (siehe Req. **R16**). Dadurch kann die Anwendung prinzipiell an jeder Stelle des Flashspeichers beginnen.

Im Entwurf des Update-Dateiformats (Kap. 3.3) wurde bereits darauf eingegangen, dass die Vektortabelle nicht beliebig verschoben werden kann, sondern nur ganzzahlige Vielfache einer gewissen Ausrichtung zulässig sind. Für den in der folgenden Implementierung eingesetzten Cortex-M4-Prozessor muss die Vektortabelle an einem Raster von 512 Byte ausgerichtet sein.

Am Anfang des Flashspeichers wird also ein Programm, welches Bootloader genannt wird, benötigt, das die Anwendung startet, die an einer anderen Stelle im Flashspeicher liegt. Die Anwendung bezeichnet dabei den Code, der in der Firmware-Update-Datei, bzw. im Firmware-Image (siehe Kap. 3.3.1), transportiert wird. Um die Anwendung nun auszuführen, muss ein Sprung in den Reset Handler der Anwendung durchgefürt werden. Damit jedoch die Vektortabelle der Anwendung für alle folgenden Interrupts genutzt wird, muss das VTOR-Register zuvor angepasst werden.

Erster Ansatz: Der Bootloader installiert das Firmware-Update

Um die Firmware zu aktualisieren, muss der Code der Anwendung ausgetauscht werden. Da die Anwendung sich nicht selbst ersetzen kann, muss diese Aufgabe der Bootloader übernehmen.

Die Update-Datei sollte dafür zuvor von der Anwendung heruntergeladen und in einem separaten Bereich des Flashspeicher abgelegt worden sein. Somit muss kein Netzwerkstack im Bootloader vorhanden sein, sodass dieser möglichst wenig Speicherplatz benötigt.

Der Nachteil dieses Ansatzes ist jedoch, dass es kein Backup für den Fall gibt, dass die neue Firmware nicht startet oder Fehler beim Überschreiben der alten Firmware aufgetreten sind. Für einen robusten Update-Prozess sollte also mindestens eine lauffähige Firmware-Version zusätzlich auf dem Gerät vorhanden sein.

Robustes Update: mehrere Firmware-Slots

Um eine hohe Robustheit des Update-Prozesses zu erzielen, werden zwei unabhängige Firmware-Slots im Flashspeicher verwendet. Somit wird bei einem Update nicht die aktuell verwendete Firmware überschrieben, sondern jeweils der Slot mit der vorletzten Version.

Die Firmware-Slots sind dabei möglichst gleich große Speicherbereiche im Flashspeicher, in denen die Firmware installiert werden kann. Prinzipiell ist zwar die Verwendung von mehr als zwei Slots möglich, dies führt jedoch auch zu höheren Kosten für einen größeren Flashspeicher. Außerdem sind für einen robusten Updateprozess zwei Slots ausreichend, weshalb alle weiteren Betrachtungen genau zwei Slots voraussetzen.

Diese Firmware-Slots müssen an den Sektoren des Flashspeichers ausgerichtet sein, damit das Löschen eines Firmware-Slots nicht den anderen Slot oder den Bootloader beeinflusst.

Der Bootloader muss also bei jedem Start des Geräts die Versionsnummern der in den beiden Slots installierten Firmware-Versionen erkennen und die aktuellste Version ausführen.

Da der Compiler zumindest die Vektortabelle mit absoluten Speicheradressen angibt, ist ein kompiliertes Programm immer nur an einer Stelle im Flashspeicher ausführbar. Daher müssen immer zwei Varianten der Firmware für die entsprechenden Startadressen der zwei Firmware-Slots erstellt werden, die als zwei separate Update-Dateien den Geräten zum Download bereitgestellt werden.

Das Update-Modul

In der Einleitung dieses Kapitels wurde erwähnt, dass die Anwendung einen Teil des Updateprozesses auf dem Gerät übernimmt. Dafür wird ein C-Modul benötigt, das alle für den Update-Prozess relevanten Prozessschritte bereitstellt, die während der Anwendungslaufzeit ausgeführt werden sollen.

Da der Anwendungscode eines IoT-Geräts schon einen Netzwerkstack für die Aufgaben außerhalb des Updates benötigt, bietet es sich an die Kommunikation mit dem Update-Server vollständig während der Anwendungslaufzeit durchzuführen. Dadurch wird Komplexität aus dem Bootloader entfernt, sodass dieser weniger fehleranfällig, aber auch kleiner ist.

Die vom Update-Modul bereitgestellten Funktionen können vom Hersteller des IoT-Gerät dazu verwendet werden ein Update durchzuführen. Welche Funktionen jedoch genau benötigt werden, wird im folgenden Kapitel erläutert.

3.4.2. Aufgabenteilung zwischen Bootloader und Update-Modul

Um ein Update zu installieren sind die folgenden Prozessschritte notwendig:

1. Update-Server anfragen, ob ein Update verfügbar ist
2. Update-Datei vom Update-Server herunterladen
3. Update-Datei überprüfen
4. Update installieren
5. Gerät neu starten, damit der nun aktuelle Firmware-Slot ausgeführt wird.

Auch wenn im ersten Ansatz angenommen wurde, dass der Bootloader das Update installiert, soll diese Entscheidung noch einmal neu evaluiert werden.

Vertrauenswürdigkeit von Bootloader und Update-Modul

Um das Update aus der Update-Datei zu installieren, muss zunächst die Dateisignatur überprüft werden, bevor die Binärdaten entschlüsselt werden können. Diese Prozesse sind sicherheitsrelevant und müssen deshalb besonders vertrauenswürdig sein.

Um zu bestimmen, ob der Bootloader vertrauenswürdiger, als das Update-Modul ist, müssen einige Annahmen getroffen werden:

1. Der Auslieferungszustand des Geräts ist vertrauenswürdig. Das bedeutet auch, dass die Firmware, die in der Fabrik aufgespielt wurde, den Angreifern nicht bekannt ist.
2. Der Flashspeicher ist vor Lese- und Schreiboperationen, die nicht von der Anwendung oder dem Bootloader selbst ausgeführt werden, geschützt. (Siehe Req. **R14**).
3. Die Kryptografie-Bibliothek und die verwendeten kryptografischen Algorithmen haben keine bekannten Schwachstellen.
4. Die Anwendung manipuliert den Inhalt des Flashspeichers oder die kryptografischen Schlüssel nicht selbst. Ausgenommen ist selbstverständlich das Update-Modul.

Wenn diese Annahmen alle wahr sind, ergibt sich als logische Schlussfolgerung, dass der Bootloader im Auslieferungszustand genauso vertrauenswürdig ist, wie die Anwendung und das darin enthaltene Update-Modul. Dies bedeutet jedoch auch, dass das Update-Modul die Update-Datei genauso sicher installieren kann, wie der Bootloader.

Wenn der Bootloader das Update installiert, kann dieser nie Over-the-Air ausgetauscht werden, sodass eventuell vorhandene Programmierfehler nie behoben werden können. Wenn jedoch das Update-Modul auch die Installation von Updates vornimmt, ist es theoretisch möglich den Bootloader zu aktualisieren. Hierbei ist jedoch zu beachten, dass es kein Backup für den Bootloader gibt. Das Update des Bootloaders erfüllt also nicht die hohen Anforderungen an die Robustheit.

Es ist außerdem anzumerken, dass Mikrocontroller mit einer Memory Protection Unit, die spezielle Speicherbereiche des Flashspeichers gesondert absichert, eine andere Betrachtung erfordern. Die oben genannten Annahmen beziehen sich auf den verwendeten STM32F4-Mikrocontroller und ähnliche Controller ohne spezielle Sicherheitsmechanismen. Als Gegenbeispiel ist jedoch die Renesas Synergy Serie zu nennen [Ing17].

Resultierender Ablauf

Aus den in diesem Kapitel ausgeführten Erkenntnissen ergibt sich also der in Abbildung 3.3 dargestellt Ablauf für ein Firmware-Update. Das Aktivitätsdiagramm stellt dabei einen vereinfachten Ablauf ohne die möglichen Fehlerfälle dar.

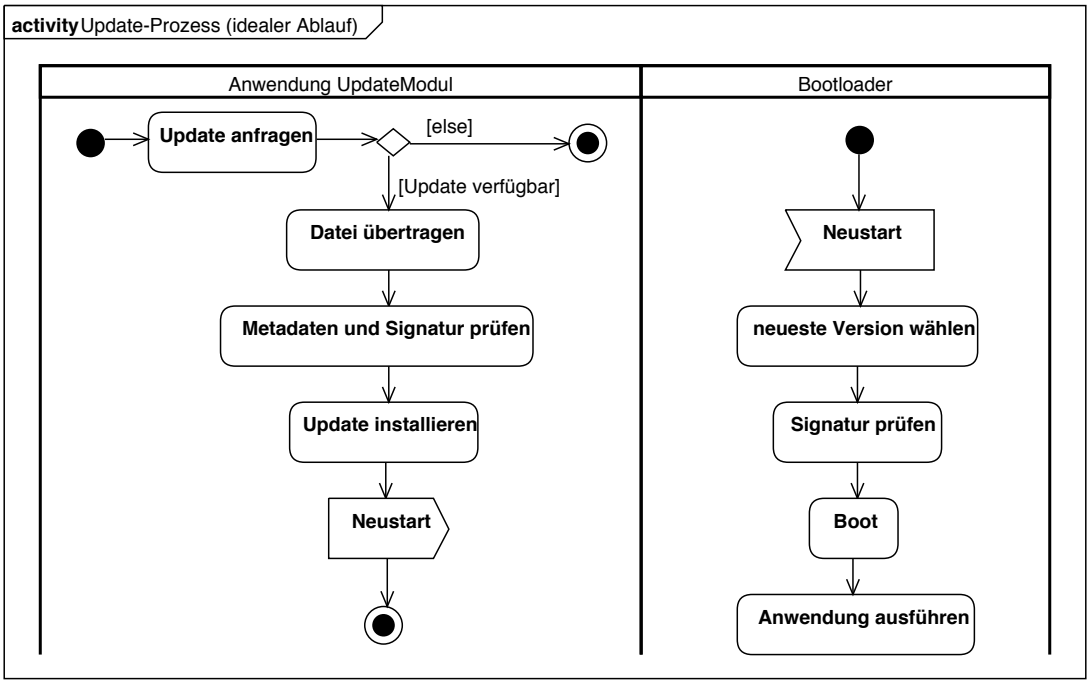

Abbildung 3.3.: Aktivitätsdiagramm: Prozessschritte auf Gerät, ohne Fehlerbehandlung

Um die Anforderung zu erfüllen, dass Updates automatisch durchgeführt werden sollen (Req. **R3**), muss die Anwendung regelmäßig beim Updateserver anfragen, ob ein Update für dieses Gerät verfügbar ist. Sofern dies der Fall ist, muss die Update-Datei heruntergeladen und wird in einem speziell dafür vorgesehenen Bereich auf dem Flashspeicher gespeichert werden. Wenn die Metadaten und Signatur erfolgreich überprüft wurden, kann die Installation beginnen. Dieser Schritt sollte nicht unterbrochen werden.

In der bisherigen Darstellung wird davon ausgegangen, dass das Gerät selbst die Anfragen an den Update-Server sendet. Wenn die IoT-Geräte jedoch nicht per W-LAN, sondern über ein IoT-spezifisches Protokoll, wie 6LoWPAN, mit dem Internet verbunden sind, wird ein Gateway benötigt. Da dieses Gateway in der Regel nicht aus einer Batterie gespeist werden wird, wäre es denkbar, dass das Gateway stellvertretend für

die Geräte die Updates anfragt und vom Update-Server herunterlädt. Dies würde jedoch auch bedeutet, dass für jeden Hersteller ein eigenes Gateway benötigt wird, sodass die Vorteile eines standardisierten Funkprotokolls zunichte gemacht werden. Daher wird diese Möglichkeit nicht weiter betrachtet.

Ein Neustart des Geräts nach einer erfolgreichen Installation führt dazu, dass der Bootloader ausgeführt wird. Dieser durchsucht die Firmware-Slots nach der neuesten Version, überprüft dessen Signatur und führt die enthaltene Anwendung aus.

Die einzelnen Schritte müssen dabei nicht direkt nacheinander ausgeführt werden, sondern sollen in einzelnen Funktionen implementiert werden, sodass die Anwendung den Update-Prozess genauer beeinflussen kann. Da beispielsweise der Download der Update-Datei und das Installieren eine relativ lange Zeit in Anspruch nehmen, muss die Anwendung bestimmen können, wann ein geeigneter Zeitpunkt für ein Update ist, da diese Entscheidung das Update-Modul nicht selbst leisten kann. Ebenfalls soll vor dem Neustart des Geräts die Möglichkeit bleiben Datenübertragungen zu beenden und eventuell Vorkehrungen zu treffen, um die Datenstrukturen mit Konfigurationsinformationen auf eine neue Version migrieren zu können.

3.5. Risikoanalyse

Im Laufe der Prozesskette treten verschiedene Risiken auf. Diese Risiken können unterteilt werden in Sicherheitsrisiken, sowie Risiken und Fehlerquellen, die die Robustheit beeinflussen. Daher werden diese Bereiche in den folgenden Abschnitten getrennt betrachtet.

3.5.1. Sicherheitsrisiken

In allen Prozessschritten, die kryptografische Schlüssel verwendeten besteht die Gefahr, dass diese Schlüssel entwendet oder manipuliert werden. Hierfür müssen vor allem auf dem Signaturserver entsprechende IT-Sicherheitsmaßnahmen vorgenommen werden. Dies liegt jedoch nicht in Verantwortungsbereich des Update-Prozesses.

Die Übertragung der Update-Datei durch das Internet erfolgt über einen prinzipiell unsicheren Kanal. Daher sind Man-in-the-middle-Angriffe, aber auch Replay- oder Denial-of-Service-Angriffe möglich. Angreifer können die Anfragen an der Update-Server mani-

pulieren und sowohl die Antwort des Update-Server und die Update-Datei mitlesen, als auch diese Daten manipulieren. Außerdem könnten eine Update-Datei oder auch jegliche andere Daten an das Gerät übertragen werden, ohne dass sich das Gerät zwangsläufig in einem Zustand befindet, in dem es diese Daten erwartet.

Das Mitlesen und Manipulieren der Kommunikation zwischen Gerät und Update-Server kann durch Verwendung einer Transportverschlüsselung verhindert werden. Für die Sicherheit des Update-Prozesses ist dies jedoch nicht erforderlich, da die Update-Datei selbst gegen diese Angriffe gesichert ist.

Auf dem Gerät selbst wäre es möglich die kryptografischen Schlüssel auszulesen und zu manipulieren. Durch Zugriff auf den Flashspeicher über die Debugging-Schnittstellen, können außerdem Teile der Firmware oder des Bootloaders manipuliert werden. Wenn jedoch die Requirements **R14** und **R13** eingehalten werden, sind diese Angriffe nicht möglich.

Diese möglichen Angriffsvektoren sind in Abbildung 3.4 entlang der Prozesskette dargestellt.

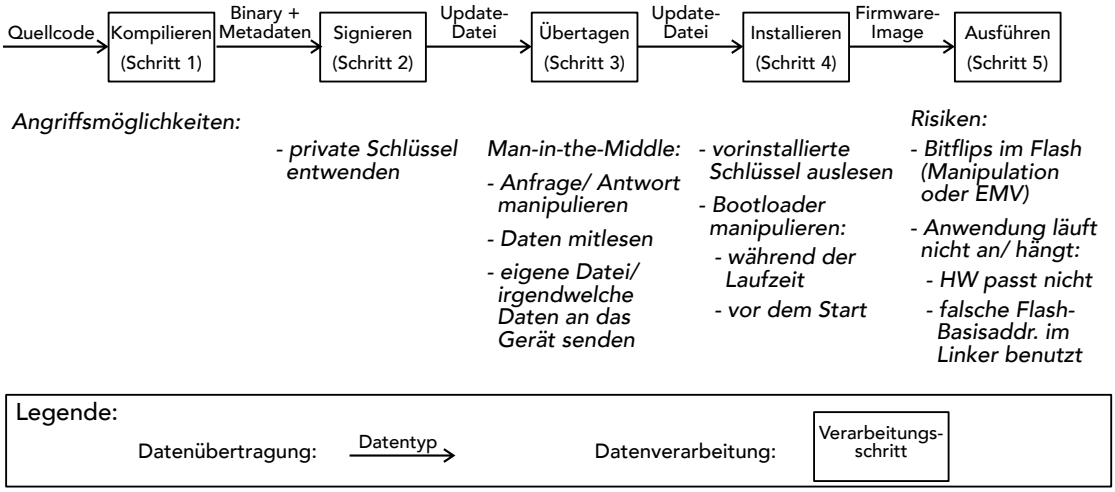

Abbildung 3.4.: Risiken im Verlauf der Prozesskette

3.5.2. Risiken für die Robustheit

Die Robustheit wird vor allem durch Fehler in der Hardware oder menschliche Fehler beeinflusst. Im laufenden Betrieb kann es beispielsweise vorkommen, dass Bits im Flash-

speicher kippen. Dies führt dazu, dass eine korrekte Ausführung der Firmware nicht mehr garantiert werden kann. Ein inkonsistentes Programm wird jedoch durch die nicht mehr passende Signatur erkannt und dementsprechend nicht mehr ausgeführt. Wenn keine weitere Firmware im zweiten Firmware-Slot vorhanden ist führt dieser Umstand zu einem irreparablen Gerät. Dies muss jedoch in Kauf genommen werden, da unbeabsichtigte Bitflips nicht von Manipulationen unterschieden werden können.

Ein weiteres Risiko besteht darin, dass eine Anwendung nicht korrekt ausgeführt werden könnte, wenn diese nicht zur Hardware passt. Dies kann nur erkannt werden, wenn die Anwendung abstürzt und das Gerät wieder in den Bootloader startet oder, indem ein Watchdog verwendet wird, um hängende Software zu erkennen. Im letzteren Fall kann der Bootloader so programmiert werden, dass ein Reset aufgrund des Watchdogs gesondert behandelt wird und danach wieder die alte Firmware gestartet wird. Da eine Firmware, die nicht korrekt startet, nur nach einen Update auftreten kann, muss die alte Firmware noch im anderen Firmware-Slot vorhanden sein.

3.6. Ergänzende Anforderungen

Durch die Spezifikation des konkreten Updateprozesses ergeben sich weitere Anforderungen an das Update-Modul, den Bootloader, die Anwendung und den Update-Server.

Anforderungen Update-Modul

R18 Update-Modul: Funktion für Update-Anfrage:
Das Update-Modul muss eine Funktion bereitstellen, die beim Update-Server anfragt, ob neue Updates verfügbar sind.

R19 Update-Modul: Funktion zum Download:
Das Update-Modul muss eine Funktion bereitstellen, die ein verfügbares Update vom Update-Server herunterlädt und in einem explizit für die Update-Datei reservierten Flash-Speicherbereich speichert.

R20 Update-Modul: Signatur und Metadaten prüfen:
Das Update-Modul darf eine Update-Datei nur installieren, wenn die Signatur gültig ist und die Metadaten zum Gerät passen.

R21 Update-Modul: Funktion für Installation:
Das Update-Modul muss eine Funktion bereitstellen, die die Update-Datei installiert.

R22 Update-Modul: Installation:
Die Installation der Update-Datei erfolgt, indem der Inhalt gemäß der Spezifikation der Update-Datei (Kap. 3.3) verifiziert, entschlüsselt und in den in der Update-Datei angegebenen Firmware-Slot gespeichert wird. Bei einer Installation muss Requirement **R20** eingehalten werden.

R23 Update-Modul: Integritätsprüfung:
Das Update-Modul darf eine Update-Datei nur installieren, wenn diese vollständig und ohne Fehler übertragen wurde.

R24 Update-Modul: passende Metadaten:
Das Update-Modul darf eine Update-Datei nur installieren, wenn die in den Metadaten der Update-Datei angegebene Hardware-ID mit der Hardware-ID der zum Zeitpunkt der Installation ausgeführten Firmware übereinstimmt. Außerdem darf eine Update-Datei nicht installiert werden, wenn die Firmware-Versionsnummer der Update-Datei kleiner ist, als die Firmware-Versionsnummer der zum Zeitpunkt der Installation ausgeführten Firmware.

R25 Update-Modul: Selbstschutz:
Das Update-Modul darf nicht den Firmware-Slot beschreiben, in dem sich die zu diesem Zeitpunkt ausgeführte Firmware befindet.

R26 Update-Modul: Unterbrochene Installation erkennen:
Das Update-Modul muss eine unterbrochene Installation erkennen und darf in diesem Fall den Download der Update-Datei überspringen. Eine unterbrochene Installation liegt in Verbindung mit Requirement **R30** vor, wenn eine gültige Update-Datei auf dem Gerät vorhanden ist und deren Firmware-Versionsnummer größer ist, als die Versionsnummer der aktuell ausgeführten Firmware.

R27 Update-Modul: Funktion für Neustart:
Das Update-Modul muss eine Funktion bereitstellen, die einen Neustart auslöst.

Anforderungen Bootloader

R28 Bootloader: höchste Versionsnummer starten:
Der Bootloader muss diejenige Firmware starten, die die höchste Firmware-Versionsnummer hat. Ausnahmen von dieser Anforderung werden durch andere Requirements definiert.

R29 Bootloader: Signaturprüfung vor jedem Start:
Der Bootloader darf eine Firmware nur starten, wenn dessen Signatur korrekt ist. Wenn von keiner der installierten Firmware-Versionen die Signatur korrekt ist, darf auch keine Firmware gestartet werden.

R30 Bootloader: Stromausfall während einer Installation:
Der Bootloader muss eine unterbrochene Installation erkennen und in diesem Fall den Firmware-Slot, der eine unvollständig installierte Firmware enthält, löschen. Nur so kann das Requirement **R26** ebenfalls die unterbrochene Installation erkennen und eine schnelle Neuinstallation beginnen. Für das Löschen eines Firmware-Slots ist es ausreichend, wenn die erste Flashpage des Firmware-Slots gelöscht wird, sofern diese mindestens so groß ist, wie der Firmware-Image-Header.

R31 Bootloader: Erkennung einer Boot-Schleife:
Der Bootloader muss erkennen, wenn die Firmware sich während des Starts aufhängt. Außerdem muss der Bootloader erkennen, ob die Firmware in kurzer Zeit mehrere Male hintereinander neu gestartet wurde. In beiden Fällen darf diese Firmware dann nicht weiter ausgeführt werden. Wenn ein weiterer Firmware-Slot belegt ist, muss der fehlerhafte Slot gelöscht werden und die alte Firmware gestartet werden. Für den Start einer Firmware gilt Req. **R29**.

Anforderungen Anwendung

R32 Anwendung: periodisches Anfragen von Updates:
Die Anwendung muss das Update-Modul gemäß Abb. 4.2 periodisch aufrufen. Nach einem Neustart, muss zuerst ein Update angefragt werden um eine eventuell unterbrochene Installation fertigstellen zu können.

Anforderungen Update-Server

R33 Server: Update-Anfrage:
Der Update-Server muss auf CoAP-GET-Anfragen, die an den Uniform Resource Identifier (URI) `/update` gesendet wurden, antworten, ob ein Updates für das anfragende Gerät verfügbar ist. Dafür muss die Anfrage die Hardware-ID des Geräts, die aktuelle Firmware-Version des Geräts, die benötigte Firmware-Slot-Nummer der Update-Datei und optional die Seriennummer des Geräts enthalten. Die CoAP-Antwort hat keinen Inhalt, wenn kein Update verfügbar ist, oder enthält den Download-URI, wenn ein Update verfügbar ist.

R34 Server: Bereitstellung der Update-Dateien:
Der Update-Server muss die Update-Dateien zum Download für die IoT-Geräte bereitstellen.

4. Exemplarische Implementierung

Um den in Kapitel 3 entworfenen Update-Prozess testen zu können, muss ein Prototyp implementiert werden. Als Mikrocontroller wurde ein STM32F4 von STMicroelectronics ausgewählt. Auch wenn vielen IoT-Geräte nur wenig Energie zur Verfügung steht, soll der Prototyp unabhängig von speziellen Energieanforderungen entwickelt werden. Daher kann ein relativ leistungsstarker Mikrocontroller, wie ein Cortex-M4, eingesetzt werden. Ein weiterer Vorteil der STM32F4-Serie ist, dass diese auch Varianten enthält, die ein Hardware-Kryptografiemodul haben. Dieses ist jedoch nicht auf dem in der Entwicklung eingesetzten Nucleo-F411 Evaluationsboard verfügbar.

4.1. Anforderungen an die Implementierung

In Kapitel 2 wurden zunächst allgemeine Anforderungen an den Update-Prozess gestellt, die nach dem Entwurf des Prozesses um Anforderungen an die Implementierung ergänzt wurden. Daher werden in Tabelle 4.1 diejenigen Anforderungen aufgelistet, die auf die Implementierung zutreffen und für jede Anforderung soll angegeben werden, ob diese implementiert werden soll.

Die automatische Abfrage und Installation von Updates (**R3**) wird nicht implementiert, da auch der Download der Dateien (**R19**) nicht implementiert wird. Der Prototyp soll zeigen, dass der Update-Prozess umsetzbar ist. Dafür ist es ausreichend, dass die Beispielanwendung ein manuell gestartetes Update ermöglicht. Das regelmäßige automatische Anfragen des Update-Servers, ob eine neue Firmware-Version verfügbar ist, muss nämlich von der Anwendung durchgeführt werden und ist nicht Teil des Update-Moduls.

Der Download der Update-Datei wird nicht implementiert, da die in RIOT zur Verfügung stehenden Bibliotheken es nicht mit den implementierten Standardprotokollen zulassen große Datenmengen zu transportieren. Es könnte alternativ ein TFTP-Server

Req.	Titel des Requirements	implementiert
R3	Automatische Updates	nein
R7	Firmware-Updates passen zur Hardware	ja
R8	Authentizität des Firmware-Updates	ja
R9	Integrität des Firmware-Updates	ja
R10	Vertraulichkeit der Firmware	ja
R11	Firmware-Updates pro einzelnem Gerät	nein
R12	Integrität der installierten Firmware sicherstellen	ja
R18	Update-Modul: Funktion für Update-Anfrage	ja
R19	Update-Modul: Funktion zum Download	nein
R20	Update-Modul: Signatur und Metadaten prüfen	ja
R21	Update-Modul: Funktion für Installation	ja
R22	Update-Modul: Installation	ja
R23	Update-Modul: Integritätsprüfung	ja
R24	Update-Modul: passende Metadaten	ja
R26	Update-Modul: Unterbrochene Installation erkennen	ja
R27	Update-Modul: Funktion für Neustart	ja
R28	Bootloader: höchste Versionsnummer starten	ja
R29	Bootloader: Signaturprüfung vor jedem Start	ja
R30	Bootloader: unterbrochene Installation erkennen	ja
R31	Bootloader: Erkennung einer Boot-Schleife	ja
R33	Server: Update-Anfrage	ja
R34	Server: Bereitstellung der Update-Dateien	nein

Tabelle 4.1.: Implementierungsstatus der Requirements

auf dem Gerät installiert werden, um zu demonstrieren, dass eine Datei über das Internet übertragen werden kann. Dies ist jedoch auch ohne eine konkrete Implementierung in diesem Kontext bekannt. Als Alternative wird die Update-Datei im Prototypen mithilfe eines Make-Skriptes direkt auf das Gerät übertragen.

Update-Dateien, die nur für ein Gerät gültig sind (**R11**), werden in der exemplarischen Implementierung ebenfalls nicht verwendet, da die Anforderung nur optional ist. Außerdem erfordert diese Option, wenn sie konsequent implementiert werden soll, einen

erheblichen Aufwand im Update-Server, da die Update-Dateien für jedes Gerät zum Zeitpunkt der Anfrage signiert werden müssen. Daher kann angenommen werden, dass gerätespezifische Updates nur von Geräteherstellern verwendet werden, die durch gesetzliche Bestimmungen oder spezielle Zertifizierungsprozesse dazu verpflichtet sind. Dies weicht jedoch von dem grundlegenden Update-Prozess ab, der durch die exemplarische Implementierung dargestellt werden soll.

4.2. Architektur-Entwurf des Update-Moduls

Zu Beginn dieser Arbeit wurde im Github-Repository von RIOT von einem Mitglied der RIOT-Community ein Pull-Request[1] erstellt. Dieser soll RIOT dahingehend erweitern, dass durch einen Bootloader verschiedene Firmware-Versionen gestartet werden können, die in Firmware-Slots auf dem Flashspeicher abgelegt sind. Da in diesem Code schon Funktionen angelegt wurden, um auf die Firmware in den Slots zugreifen zu können, wurde dieses Interface übernommen und auf das eigene Datenformat des Firmware-Images angepasst.

In RIOT können Parameter sowohl für die verwendete CPU, als auch das verwendete Board in eigenen Header-Dateien festgelegt werden. Da die Konfiguration der Firmware-Slots vom Flashspeicher des Mikrocontrollers abhängig ist, müssen diese Parameter und ein paar grundlegende Funktionen in der CPU-Konfiguration festgelegt werden.

In den Anforderungen in Kapitel 3.6 wurde spezifiziert, welche Funktionen vom Update-Modul bereitgestellt werden müssen. Dadurch ergibt sich direkt das Interface des entsprechenden C-Moduls, welches `ota_updater` genannt wird.

Um größtmögliche Flexibilität bei der Handhabung der Update-Datei zu erhalten, wird ein weiteres Modul benötigt, das die entsprechenden Funktionen und Datenstrukturen definiert, die für die Handhabung der Update-Dateien benötigt werden.

Aus diesen Überlegungen, sowie den Namenskonventionen von RIOT, ergibt sich das in Abbildung 4.1 dargestellt Klassendiagramm. Dabei ist zu beachten, dass die UML-Symbole für Klassen dazu verwendet wurden C-Module zu beschreiben.

Die Header-Dateien dieser Module werden nicht nur von den RIOT-Anwendungen, sondern auch von den Hilfsprogrammen benutzt, die die Update-Dateien und Metadaten

[1] https://github.com/RIOT-OS/RIOT/pull/6450

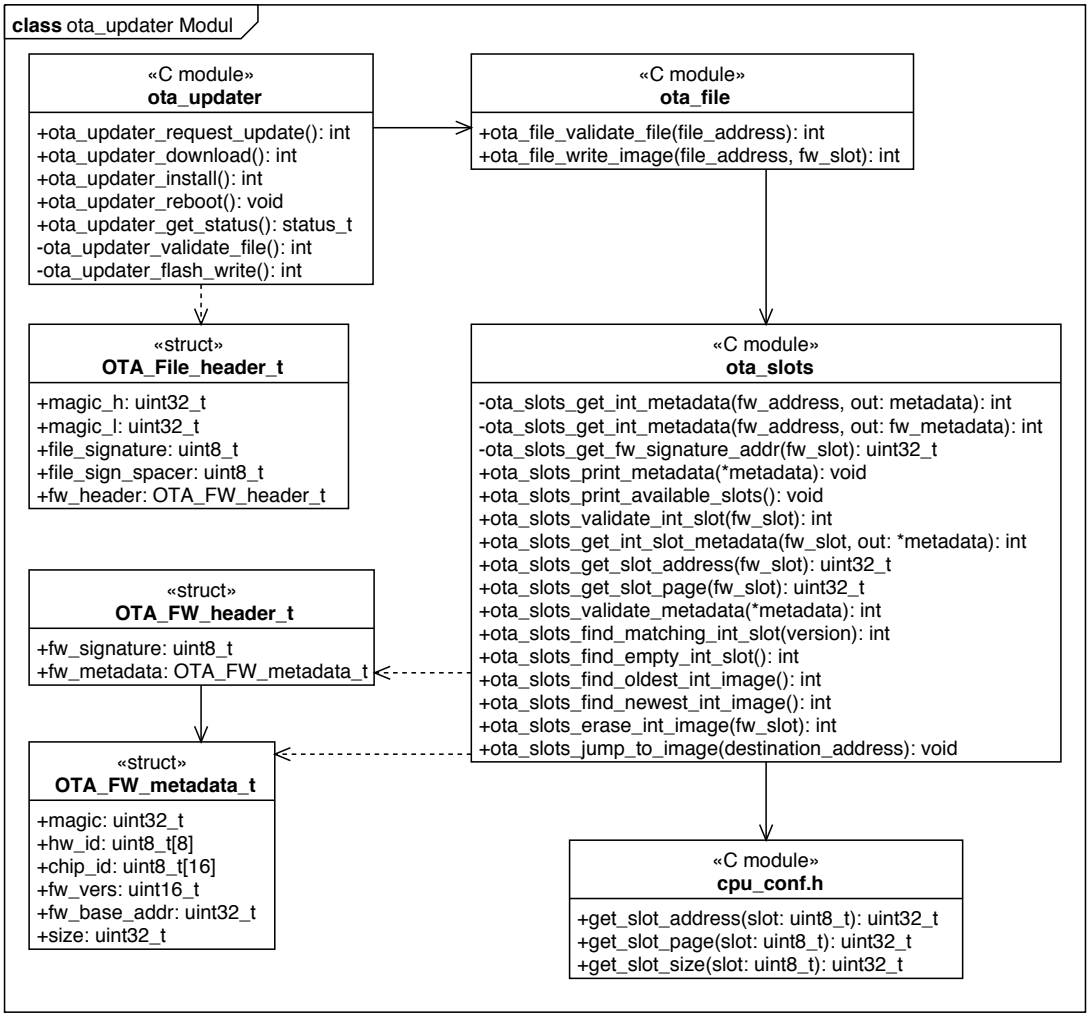

Abbildung 4.1.: Klassendiagramm: C-Module

erstellen. Somit ist sichergestellt, dass in der gesamten Prozesskette dieselben Datenstrukturen und Konstanten verwendet werden, auch wenn die Hilfsprogramme nicht RIOT benutzen, sondern reine C-Programme sind.

Der Ablauf, wie das `ota_updater`-Interface zu benutzen ist, wird im Aktivitätsdiagramm in Abbildung 4.2 dargestellt. Dieser Ablauf ist von der Anwendung einzuhalten, kann jedoch nach jedem Schritt unterbrochen werden.

Die Anwendung sollte regelmäßig nach Updates suchen, indem eine Anfrage an den Update-Server gesendet wird. Das Zeitintervall kann dabei jedoch nur in Zusammenhang

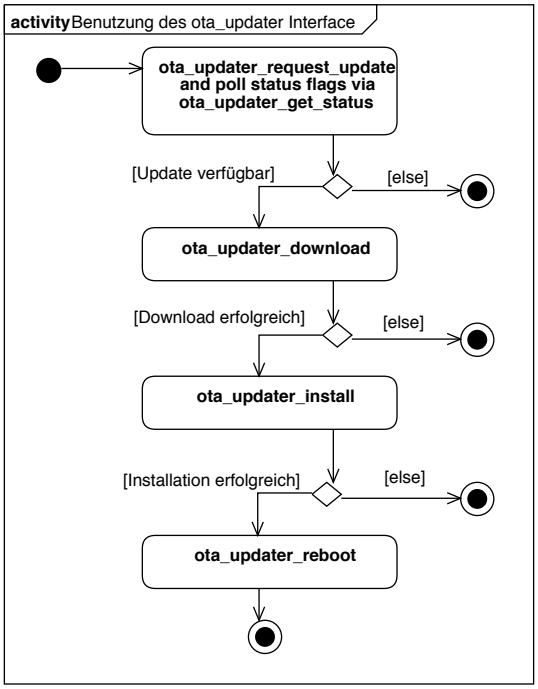

Abbildung 4.2.: Aktivitätsdiagramm: Benutzung des `ota_updater`-Moduls

mit dem konkreten Anwendungsfall sinnvoll festgelegt werden. Für die Anfrage wird das Protokoll CoAP verwendet, welches in RIOT nur ein asynchrones Interface hat. Daher kann die Funktion `ota_update_request_update()` nicht direkt zurückgeben, ob ein Update verfügbar ist (Req. **R18**). Dafür muss einige Zeit später der Status des Updates über die Funktion `ota_update_get_status()` abgefragt werden. Die Funktion zur Anfrage eines Updates kann jedoch auch erkennen, ob ein Update beispielsweise durch einen Stromausfall oder Reset unterbrochen wurde. In diesem Fall muss erst das unvollständige Update fortgeführt werden und es wird keine Anfrage an den Update-Server gesendet.

Wenn festgestellt wurde, dass ein Update verfügbar ist, kann mit der Funktion `ota_update_download()` der Download der Update-Datei gestartet werden (Req. **R19**). Der Download wird jedoch im Prototypen nicht implementiert, da RIOT noch eine Unterstützung für einen CoAP Block-Transfer hat und damit keine großen Datenmengen übertragen werden können.

Bei einem erfolgreichen Download kann die Update-Datei mit der Funktion `ota_update_install()` installiert werden (Req. **R22**). Da die neue Firmware

erst durch einen Neustart vom Bootloader ausgewählt werden kann, muss mit `ota_update_reboot()` ein System-Reset ausgelöst werden (Req. **R27**).

Ein Firmware-Update führt zwangsläufig zu einem Ausfall oder einer Beeinträchtigung der Funktion des IoT-Geräts. Der Funktionsausfall tritt dabei zumindest während der Installation, je nach Implementierung aber auch während des Downloads der Update-Datei, auf. Daher sind die Schritte des Updateprozesses in einzelnen Funktionen ausgeführt, damit im Kontext einer konkreten Anwendung bestimmt werden kann, zu welchem Zeitpunkt welche Beeinträchtigung der Funktionalität verträglich ist.

4.3. Struktur des Prototypen

Der Prototyp besteht aus einer Beispielanwendung, dem Bootloader, mehreren Hilfswerkzeugen und einem Update-Server.

Die Beispielanwendung wird über das sogenannte Ethernet over Serial (ethos) Protokoll an ein virtuelles Netzwerkinterface eines Computers angeschlossen und kann gleichzeitig über die serielle Konsole bedient werden. Zur Demonstration wird keine automatische Installation der Updates implementiert, sondern das Update-Modul kann nur über die serielle Schnittstelle manuell bedient werden.

Um den gesamten Update-Prozess von einer zentralen Stelle steuern zu können gibt es zusätzlich einen Ordner in den RIOT-Beispielen, der ein Makefile enthält, das alle benötigten Aktionen als Makefile-Targets bereitstellt.

Der Update-Server dient als einfache Gegenstelle für das Update-Modul.

4.4. Beispielanwendung

Die Beispielanwendung hat zunächst die einfache Funktion irgendeine RIOT-Anwendung zu sein, die vom Bootloader gestartet werden kann. Andererseits kann die Anwendung dazu genutzt werden das Update-Modul explorativ zu testen.

4.4.1. Funktionen der Beispielanwendung

Um das Update-Modul manuell bedienen zu können, werden entsprechende Befehle in der RIOT-Shell zur Verfügung gestellt. Um die Komplexität gering zu halten und weil das verwendete Nucleo-Board keine Ethernet-Schnittstelle hat, wird die Anwendung per Ethernet-over-Seriel (ethos) an das Netzwerk eines Computers angeschlossen.

Ethos ermöglicht es durch Multiplexing sowohl die normalen Nachrichten der seriellen Schnittstelle, als auch eine Ethernet-Verbindung über eine einzige serielle Schnittstelle zu führen. Dafür muss jedoch auf dem PC ein spezielles Programm verwendet werden, welches die Ethernet-Pakete an ein virtuelles Netzwerkinterface weiterleitet und gleichzeitig ein interaktives Terminal zur Verfügung stellt. Über das zentrale Makefile (siehe Kap. 4.5.3) kann das Ethos-Programm per `make ethos` ebenso komfortabel aufgerufen werden, wie das in den Standard-Makefiles von RIOT mitgelieferte Terminal (`make term`).

Für Demonstrationszwecke ist es ausreichend, dass die Beispielanwendung mit einem Server auf demselben PC kommunizieren kann. Daher sind Link-lokale IPv6-Adressen ausreichend und es wird kein Routing benötigt.

In der RIOT-Shell können die vom `ota_updater`-Modul bereitgestellten Funktionen über einzelne Shell-Befehle gestartet werden und es wird der Rückgabewert auf der Konsole ausgegeben. Für den Prototypen reicht es aus, manuell den Update-Server anzufragen, insbesondere da der Download der Update-Datei nicht implementiert ist (siehe Kap. 4.2). Die Update-Dateien können jedoch über das zentrale Makefile direkt in den Update-Slot des Mikrocontrollers geflasht werden. Je nachdem, in welchen Zeitintervallen nach Updates gesucht wird, wäre es außerdem relativ zeitaufwändig auf das automatische Installieren eines Updates zu warten.

4.4.2. Anmerkungen zur Implementierung

Die Überprüfung der in der Update-Datei angegebenen Seriennummer ist im Update-Modul nicht implementiert, da die Prototypen-Infrastruktur keine gerätespezifisch signierten Updates unterstützt. Die Update-Datei-Generatoren sind jedoch schon mit der entsprechenden Kommandozeilenoption implementiert, sodass eine Chip-Seriennummer angegeben werden könnte. In einem automatisierten Prozess muss der Update-Server den Update-Datei-Generator für jedes Gerät aufrufen, um eine Update-Datei mit der

angefragten Seriennummer zu erstellen. Dies geht jedoch über die Anforderung einen Proof-of-concept zu erstellen hinaus und wurde deshalb nicht implementiert.

Eine weitere offene Fragestellung ist, wie die kryptografischen Schlüssel sicher auf dem Gerät abgelegt werden sollen. Da die STM32-Mikrocontroller einen sogenannten One-Time-Programmable (OTP) Bereich im Flashspeicher haben, der nur beschreibbar, aber nicht löschbar ist, könnte dieser Bereich für die Ablage der Schlüssel verwendet werden. Sobald der Bereich nach Speicherung der Schlüssel gegen weiteres Beschreiben gesichert wird, können die Schlüssel auch nicht verändert werden. Dies bedeutet jedoch auch, dass die Schlüssel nicht aktualisiert werden können.

Der entscheidende Nachteil des OTP-Bereichs als sicherer Schlüsselspeicher ist jedoch, dass ein Angreifer, der Zugriff auf die Hardware hat, diesen mit einem Debugger auslesen kann, wenn nicht jegliche Debugging-Schnittstellen deaktiviert wurden. Dies trifft auch zu, wenn die Flash Read Protection des STM32 verwendet wurde, um den Flashspeicher komplett zu löschen, bevor eine andere Firmware über den Debugger ausgeführt werden kann (Req. **R14**). Daher ist der OTP-Bereich als sicherer Schlüsselspeicher nicht geeignet.

Da die Möglichkeiten der sicheren Schlüsselablage jedoch stark von der verwendeten Hardware abhängen, kann es keine universelle Lösung geben, weshalb dieser Aspekt in der Implementierung des Prototypen ausgelassen wurde.

4.5. Werkzeuge zur Erstellung der Update-Dateien

In Kapitel 3 wurde die Prozesskette so aufgeteilt, dass die Metadaten beim Kompilieren direkt an die Binärdaten angehängt werden und diese kombinierten Daten erst in einem weiteren Schritt signiert und damit zu einer Update-Datei umgewandelt werden. Sowohl für das Anhängen der Metadaten, als auch das Signieren werden weitere Werkzeuge benötigt, die in den nächsten Abschnitten beschrieben werden. Im letzten Abschnitt dieses Kapitels werden die Bestandteile des Makefiles erläutert, mit dem der Update-Prozess bedient werden kann.

Die folgenden Abschnitte erklären jeweils die grundsätzliche Funktion der Werkzeuge. Dies soll jedoch keine vollständige Dokumentation darstellen, da diese in den Readme-Dateien in den entsprechenden Dateipfaden der einzelnen Werkzeuge nachgeschlagen werden kann.

4.5.1. Hilfswerkzeug: Metadaten-Generator

Der Metadaten-Generator (`ota_update_filemeta`) erzeugt die Metadatenstruktur, die in Kapitel 3.3.2 festgelegt wurde, und speichert diese als Binärdatei. Der Inhalt der einzelnen Metadatenfelder wird dabei in den Kommandozeilenparametern angegeben und die Ausgabe wird im Arbeitsverzeichnis angelegt.

Dieses Werkzeug wird in der Regel als Teil des Kompilierens automatisch aufgerufen, da im zentralen Makefile von RIOT (`Makefile.include`) die entsprechenden Anweisungen integriert wurden.

4.5.2. Hilfswerkzeug: Datei-Signierer

Der Datei-Signierer (`ota_update_filesign`) wandelt eine Binärdatei, die aus den Metadaten und dem Maschinencode der Anwendung besteht, zu einer Update-Datei um, indem die Daten signiert werden. Dafür müssen die Dateipfade zur Quelldatei und den Schlüsseldateien in den Kommandozeilenparametern angegeben werden. Die Update-Datei wird im Arbeitsverzeichnis angelegt.

Da die Möglichkeit bestehen soll während der Entwicklung Firmware-Images, also entschlüsselte Update-Dateien, direkt auf dem Flashspeicher anzulegen, ist dieses Werkzeug in zwei Varianten aufgeteilt. Beim Kompilieren des Werkzeugs werden sowohl ein Generator für Update-Dateien, als auch ein Generator für Flash-Images erstellt.

4.5.3. Hilfswerkzeug: Makefile

RIOT verwendet das GNU-Make Build-System um Abhängigkeiten zwischen Modulen zu definieren und den Quellcode zu kompilieren. Es werden aber auch Makefile-Targets bereitgestellt, um die Software auf das verwendete Board zu flashen. Um eine Anwendung, die den Update-Prozess nutzt, entwickeln und testen zu können werden jedoch andere Makefile-Targets benötigt, als normalerweise von RIOT zur Verfügung gestellt werden.

Für erste Experimente ist es ausreichend die Dateien auf demselben Computer zu signieren, auf dem auch kompiliert wird. Dafür steht das Makefile-Target `sign-updatefiles` bereit.

Wenn jedoch Updates über einen Update-Server an Geräte im Feld verteilt werden sollen, sollten die Binärdaten, die nach dem Kompilieren erzeugt wurden, zum Signieren an einen separaten Signatur-Server übertragen werden. Dieser sollte die von ihm erstellten Update-Dateien dann erst an den Update-Server weiterreichen, sodass letztendlich eine neue Firmware-Version bereitsteht.

Alle weiteren Funktionen, die über das Makefile ausgelöst werden können, sind im dazugehörigen Readme aufgelistet und im Detail erklärt.

4.5.4. Hilfswerkzeug: Schlüsselgenerator

Um die kryptografischen Schlüssel für die Update-Datei zu erstellen, wird ein weiteres Werkzeug unter `/dist/tools/nacl_key_generator` bereitgestellt. Dieses Kommandozeilentool nimmt keine Argumente entgegen und erstellt Binärdateien im Arbeitsverzeichnis, die die Schlüssel enthalten.

4.6. Bootloader-Architektur

Die grundlegende Aufgabe des Bootloaders ist es diejenige Firmware zu starten, die die höchste Versionsnummer hat (Req. **R28**). Dabei können jedoch verschiedene Fehler auftreten, die entsprechend behandelt werden müssen, um in möglichst vielen Fällen das Gerät funktionsfähig zu halten. Da jedoch nicht jeder Fehler behoben werden kann, muss das Gerät in letzter Instanz in einen für die Umwelt sicheren Zustand versetzt werden.

4.6.1. Verhaltensmodell des Bootloaders

Um alle Anforderungen in Zusammenarbeit mit dem Update-Modul erfüllen zu können, implementiert der Bootloader das in Abbildung 4.3 dargestellte Aktivitätsdiagramm.

Damit die Sicherheit des Geräts gewährleisten werden kann, muss vor jedem Start einer Firmware dessen Signatur überprüft werden (Req. **R29**). Wenn diese Überprüfung fehlschlägt, darf die zugehörige Firmware nicht gestartet werden.

Der Bootloader muss außerdem das Requirement **R31** erfüllen, das fordert, dass eine Firmware nicht mehr ausgeführt werden darf, wenn diese sich aufhängt oder einen Reset

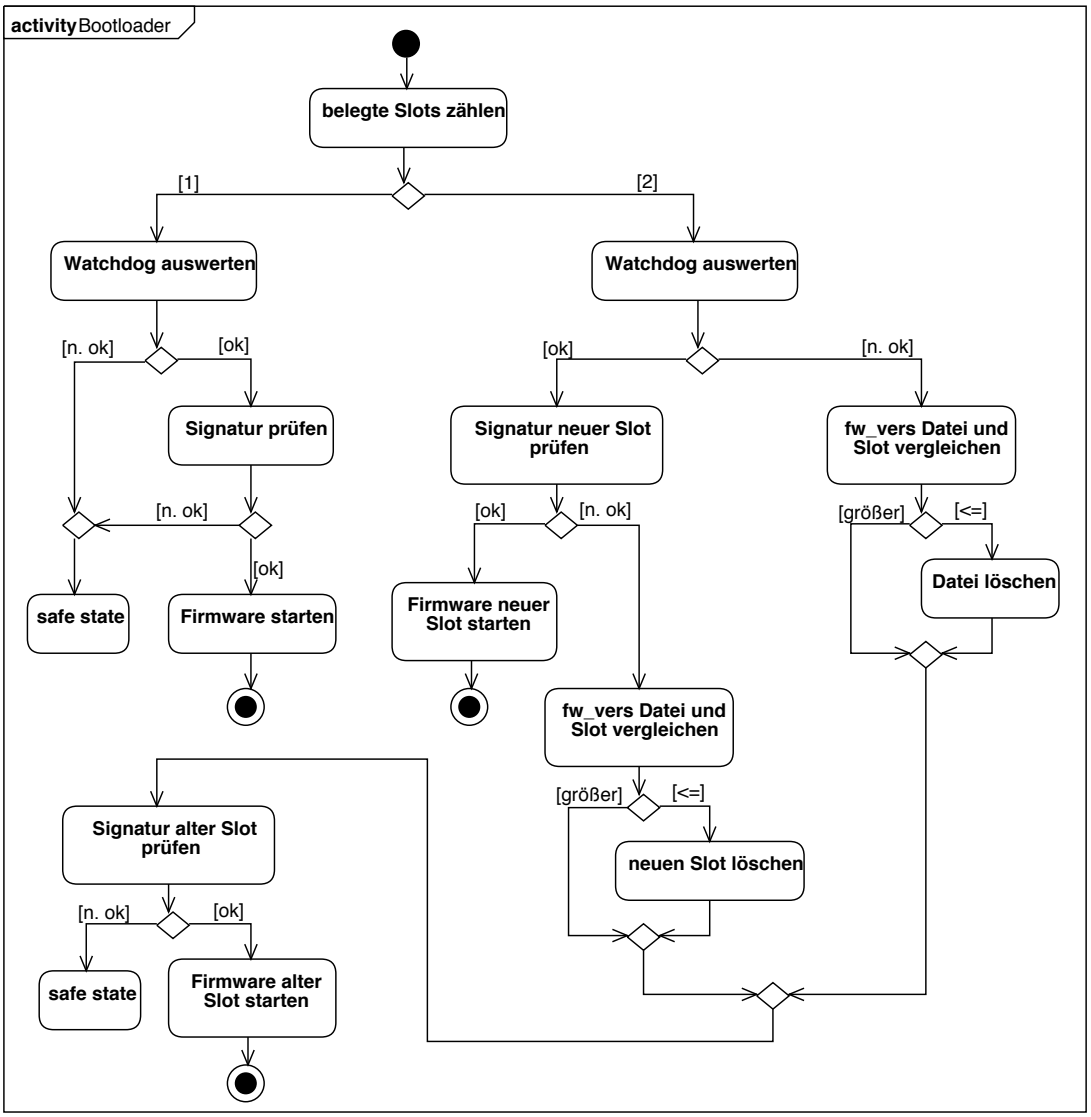

Abbildung 4.3.: Aktivitätsdiagramm: Bootloader

kurz nach dem Start auslöst. Diese Funktionalität kann nur durch die Verwendung eines Watchdog-Timers realisiert werden. Ein Watchdog ist ein abwärtszählender Timer, der von der Software auf seinen Rücksetzwert zurückgesetzt werden kann. Wenn der Timer jedoch Null erreicht, wird ein System-Reset ausgelöst und ein Flag im Reset and Clock Control Register gesetzt, um die Ursache des Resets anzuzeigen.

Zuerst muss jedoch unterschieden werden, ob eine oder zwei Firmware-Versionen instal-

liert sind. Dabei wird jedoch noch nicht geprüft, ob die Firmware-Images eine gültige Signatur haben.

Wenn nur eine Version installiert ist, kann im Fehlerfall keine Alternative gestartet werden, sodass diese Hälfte des Aktivitätsdiagramms relativ übersichtlich ist.

Wenn zwei Versionen installiert sind, wird ebenfalls zuerst der Watchdog ausgewertet. Normalerweise und vor allem nach der Installation eines Updates wird dieser noch nicht ausgelöst sein, sodass die Signatur des neuesten Slots geprüft wird.

Eine ungültige Signatur kann zwei Ursachen haben: Ein Update wurde während der Installation beispielsweise durch einen Stromausfall (Req. **R30**) unterbrochen oder das Programm ist durch Bitflips oder Manipulationen inkonsistent geworden (Req. **R12**). Diese beiden Fälle können jedoch nicht zuverlässig unterschieden werden, sodass zunächst von einem unvollständig installierten Update ausgegangen wird. Diese Annahme kann dadurch bestätigt werden, dass eine Update-Datei mit derselben Versionsnummer vorhanden ist, wie der Slot mit der neuesten Firmware. In diesem Fall muss der Slot gemäß Requirement **R30** gelöscht werden, damit das Update-Modul gemäß Requirement **R26** eine erneute Installation durchführen kann. Andererseits bedeutet eine Update-Datei mit einer höheren Versionsnummer, dass diese Datei noch nicht installiert oder nicht vollständig heruntergeladen wurde.

Da der Slot mit der neuesten Firmware-Version nicht ausführbar ist, muss letztendlich auf die alte Version ausgewichen werden. Natürlich muss auch für diese Version zunächst die Signatur überprüft werden.

Wenn jedoch beide Slots belegt sind und der Watchdog ausgelöst wurde, bedeutet dies, dass ein Update zwar formal korrekt installiert wurde aber trotzdem nicht lauffähig ist. Auch in diesem Fall wird überprüft, ob Firmware-Version der Update-Datei der installieren Version entspricht. Sofern dies der Fall ist, wird die Datei gelöscht. Dadurch kann die Anwendung diesen Fall erkennen und den Fehler an den Update-Server oder ein Monitoring-System melden. Dies erfordert jedoch, dass ein Flag persistent abgespeichert wird, sobald ein Update vom Update-Modul erfolgreich installiert wurde und die Anwendung ein Upgrade erwartet.

Beim Vergleichen der Firmware-Versionen von Update-Datei und aktuellstem Firmware-Slot wird ein Kleiner-Gleich-Zeichen nur der Vollständigkeit halber verwendet, da der Fall nie auftreten darf, dass die installierte Version neuer ist, als die Update-Datei.

4.6.2. Anmerkungen zur Implementierung

Während der Entwicklung kann es sinnvoll sein die Firmware-Slots oder auch die Update-Datei nicht zu löschen, um nach Auftreten eines Fehlers noch die Fehlerursache bestimmen zu können. Dieses Verhalten soll sich jedoch absichtlich nicht per Präprozessordirektive einschalten lassen, damit diese Modifikation am Bootloader bewusst vom Entwickler durchgeführt werden muss. So soll vermieden werden, dass eine Bootloader-Version in der Produktion eingesetzt wird, die sich in einer Debug-Konfiguration befindet.

4.7. Update-Server

Da in dieser Arbeit die für ein OTA-Update notwendige Prozesskette möglichst vollständig in einem Prototypen umgesetzt werden sollte, wird auch ein Update-Server benötigt. Nun wurde in Kapitel 4.4 schon erläutert, dass nur die Anfrage, ob ein Update verfügbar ist, im Update-Server implementiert wurde, aber nicht der Download der Datei selbst. Es wurde jedoch noch nicht spezifiziert, wie das Kommunikationsprotokoll zwischen IoT-Gerät und Update-Server aufgebaut ist.

4.7.1. Kommunikationsprotokoll

In Requirement **R33** wurde schon angegeben, welche Informationen die Anfrage des IoT-Geräts beinhalten muss. Für die Kommunikation bietet sich das CoAP-Potokoll als Basis an, da dieses speziell für die Anforderungen des IoT entwickelt wurde. CoAP steht für „Contrained Application Protocol“ und ist prinzipiell eine vereinfachte Version von HTTP, die unter anderem auf möglichst geringe Paketgrößen ausgelegt ist. Außerdem wird die Kommunikation auf Basis von UDP-Paketen durchgeführt.

Eine CoAP-Nachricht [MHCK07] sollte in ein einziges UDP-Paket passen. Wenn jedoch bekannt ist, dass auf der Übertragungs- und Sicherungsschicht des OSI-Modells ein Protokoll verwendet wird, das nur kleinere Datenpakete zulässt, wird empfohlen sich auf die kleinste Größe anzupassen. Bei der Verwendung von IEEE 802.15.4 mit 6LoWPAN als Vermittlungsschicht stehen beispielsweise nur 33 Byte [MHCK07, Kap. 4] für die Nutzdaten zur Verfügung.

Daher sollte auch in der Kommunikation zwischen IoT-Gerät und Update-Server darauf geachtet werden, dass möglichst wenig Bytes verwendet werden. Aus diesem Grund werden die Informationen direkt als Binärwerte übertragen.

Anfrage

Die Anfrage ist eine GET-Anfrage, die ohne optionale Felder 11 Byte lang und wie folgt aufgebaut ist:

- Byte 0:1 – aktuelle Firmware-Version (uint16_t, big endian)
- Byte 2 – Firmware-Slot-Nummer für den die Update-Datei benötigt wird (uint8_t)
- Byte 3:10 – Hardware-ID (siehe Kap. 3.3.2)
- Byte 11:26 – optional: Seriennummer (siehe Kap. 3.3.2)

Die Hardware-ID wird im Makefile des `ota_update`-Beispiels als hexadezimale Zahl angegeben und in dieser Schreibweise dem Metadaten-Generator übergeben. Wenn der Generator und das Update-Modul die Hardware-ID aufgrund der einfacheren Handhabung als 64 Bit unsigned integer interpretieren und speichern sollten, muss darauf geachtet werden, dass die GET-Anfrage ein Byte-Array und damit einen big endian-Wert erwartet.

Antwort

Die Antwort ist eine CoAP-Antwort, die die Download-URI enthält, sofern ein Update verfügbar ist. Dies kann jedoch auf die individuellen Anforderungen angepasst werden, wenn auch der Download der Datei implementiert wurde.

Wenn kein Update verfügbar ist, ist der Inhalt der CoAP-Antwort leer.

4.7.2. Implementierung des Update-Servers

Der Update-Server ist in Go implementiert, da die CoAP-Library für Go aufgrund der Dokumentation und Beispiele ein schnelles und einfaches Ergebnis versprach. Das Beispiel ist im Repository unter `examples/ota_update_server` abgelegt und das Readme beschreibt, wie die Go-Entwicklungsumgebung aufgesetzt werden muss und der Server gestartet wird.

5. Testkonzept

Im vorhergegangenen Kapitel wurde der Firmware-Update-Prozess exemplarisch implementiert, indem RIOT als embedded Betriebssystem verwendet wurde. Dadurch wurde der Prozess an vielen Stellen noch genauer definiert. Diese Implementierung soll jedoch nicht nur dazu dienen zu zeigen, dass das Konzept umsetzbar ist (Proof of concept), sondern auch die Basis für Tests bilden.

Die Tests sollen die Implementierung der Anforderungen überprüfen, die an den Update-Prozess und damit auch an die exemplarische Implementierung gestellt wurden. Dabei soll das Testkonzept nicht auf die im Beispiel verwendete Hardware beschränkt sein, sondern es soll ohne großen Aufwand auf viele verschiedene IoT-Geräte angepasst werden können. Somit stehen in dieser Arbeit nicht nur ein Update-Konzept mit Beispiel zur Verfügung, sondern auch die entsprechenden Systemtests.

5.1. Testfälle

Die Tests müssen die folgenden Szenarien abdecken:

- Normalfall 1: Das Gerät liegt im Fabrikzustand vor und es wird ein Update (in Slot 2) installiert.
- Normalfall 2: Beide Slots des Geräts sind belegt und Slot 2 enthält die neuere Firmware. Dann wird ein Update in Slot 1 installiert.
- Normalfall 3: Beide Slots des Geräts sind belegt und Slot 1 enthält die neuere Firmware. Dann wird ein Update in Slot 2 installiert.
- Manipulierte oder fehlerhafte Übertagung: Es wird versucht eine Update-Datei zu installieren, deren Signatur ungültig ist.

- Downgrade: Es wird versucht eine Update-Datei zu installieren, deren Versionsnummer geringer ist, als die aktuell ausgeführte Version.
- Falsche Hardware: Es wird versucht eine Update-Datei zu installieren, deren Hardware-ID nicht mit der Hardware-ID des Geräts übereinstimmt.
- Stromausfall: Während der Installation wird die Spannungsversorgung des Mikrocontrollers getrennt.
- Bitflip: In einem Firmware-Slot liegt ein Firmware-Image vor, in dem einige Bits gekippt sind.
- Boot-Schleife: Beide Firmware-Slots des Geräts sind belegt. Die Firmware mit der höheren Versionsnummer in Slot 2 simuliert jedoch eine fehlerhafte Firmware, die den Watchdog nicht zurücksetzt. Dies ist für den Bootloader äquivalent zu einer Firmware, die nicht korrekt startet und dabei einen System-Reset auslöst, sodass bei jedem Start der Firmware in den Bootloader zurück gesprungen wird.

5.1.1. Beschreibung der Tests

Die oben genannten Szenarien wurden zu fünf Tests zusammengefasst. Die Kommunikation mit dem Gerät erfolgt über die vorhandenen Schnittstellen. Im Falle der Beispielanwendung ist dies Ethernet-over-Serial, für den Bootloader wird das serielle Terminal verwendet. Details zur Implementierung werden im nächsten Abschnitt (5.2) behandelt.

Test 1: Normalfall

Der erste Test soll überprüfen, ob die Update-Dateien im Normalfall korrekt installiert werden. Dafür wird das Gerät zunächst mit den Fabrik-Hexfile geflasht, das den ersten Slot mit Firmware-Versionsnummer 1 belegt.

Wenn diese Firmware erfolgreich startet, kann eine Update-Datei mit Versionsnummer 2 für Slot 2 in den Update-Datei-Slot geflasht werden. Dieses Update wird daraufhin durch Eingabe des entsprechenden RIOT-Shell-Befehls installiert. Nachdem das Gerät neu gestartet wurde, muss überprüft werden, ob die neue Firmware-Version ausgeführt wird.

Sobald das erste Update erfolgreich durchgeführt wurde, kann der Prozess mit einer neuen Update-Datei wiederholt werden. Das Update muss in diesem Schritt Firmware-Version 3 haben und für Slot 1 bestimmt sein.

Nach einem erfolgreichen Update auf Version 3, wird der Vollständigkeit halber erneut Slot 2 aktualisiert, indem Firmware-Version 4 installiert wird.

Test 2: fehlerhafte Update-Dateien

Test 2 prüft, ob verschiedene fehlerhafte Update-Dateien erkannt und nicht installiert werden. Dafür wird das Gerät vorbereitet, indem ein Test-Hexfile mit Firmware-Version 2 und 3 auf das Gerät geflasht wird.

Der Hauptteil des Tests wurde in drei kleinere Abschnitte unterteilt, die alle gleich ablaufen, nur jeweils eine andere Update-Datei verwenden. In jedem Abschnitt wird die Datei in den Update-Datei-Slot geflasht und die Installation über die RIOT-Shell gestartet. Anhand des Rückgabewertes der Installations-Funktion oder der Statusmeldungen wird überprüft, ob die fehlerhafte Datei erkannt und die Installation abgebrochen wurde.

Als fehlerhafte Dateien werden eine Datei mit ungültiger Signatur, eine Datei mit unpassender Hardware-ID und eine Datei mit Firmware-Versionsnummer 1 verwendet. Wenn nicht anders angegeben ist die Datei für Slot 1 bestimmt und hat die Versionsnummer 4.

Da sich das Test-Script im selben Verzeichnis befindet, wie das Makefile zur Erzeugung von Update-Dateien, werden auch die ungültigen Dateien in diesem Verzeichnis abgelegt. Daher müssen im letzten Schritt des Tests die ungültige Dateien vom Dateisystem gelöscht werden, damit diese nicht versehentlich durch andere Personen oder Prozesse verwendet werden.

Test 3: Stromausfall

Das Testen eines Stromausfalls während der Installation ist relativ komplex und erfordert, dass die Spannungsversorgung des Mikrocontrollers automatisch geschaltet werden kann. Die Realisierung dieser Funktion wird in Kapitel 5.2 beschrieben.

Das Gerät wird, wie in Test 1, in den Fabrikzustand versetzt, indem der Bootloader und eine Firmware mit Versionsnummer 1 geflasht werden. Außerdem wird eine Update-Datei für Slot 2 mit Versionsnummer 2 auf dem Gerät abgelegt.

Daraufhin kann die Installation gestartet werden, die Spannungsversorgung des Mikrocontrollers muss jedoch direkt nachdem der Firmware-Image-Header geschrieben wurde unterbrochen werden. Daraufhin muss die Kommunikation mit der Beispielanwendung, die die Ethernet-over-Serial (ethos) Schnittstelle nutzt, beendet werden, da ansonsten die Kommunikation mit dem Bootloader nicht erfolgreich ablaufen kann. Für diese Kommunikation wird nämlich dieselbe serielle Schnittstelle verwenden, nur ohne das ethos-Multiplexing.

Nachdem also das ethos-Programm, welches das Demultiplexing der Ethernet-Pakete und des seriellen Terminals vornimmt, beendet wurde, wird ein serielles Terminal für den Bootloader gestartet und die Spannungsversorgung wiederhergestellt. Da die Bootloader-Kommunikation schon vor den Einschalten des Geräts initialisiert wurde, können alle Nachrichten des Bootloaders ausgewertet werden. Diese Nachrichten müssen darstellen, dass eine unvollständige Installation erkannt wurde und deshalb der Firmware-Slot 1 gestartet wird.

Da der Bootloader ab diesem Zeitpunkt nicht mehr ausgeführt wird, muss wiederum dessen Kommunikationsschnittstelle beendet werden, um das ethos-Programm erneut starten zu können. Die Beispielanwendung sollte ebenfalls melden, dass wieder die Versionsnummer 1 gestartet wurde. Der Vollständigkeit halber wird die Installation des Updates zum Ende dieses Tests noch einmal gestartet, jedoch ohne die Spannungsversorgung zu trennen. Die Funktion zur Anfrage eines Updates vom Update-Server (**R18**) muss dabei gemäß Requirement **R26** erkannt haben, dass ein Update bei der Installation unterbrochen wurde.

Test 4: Bitflip/ Manipulation im Flashspeicher

Der vierte Test prüft, ob der Bootloader manipulierte Firmware-Images erkennt und zur Wiederherstellung der Funktionalität des Gerätes die alte Firmware startet. Dafür muss das Firmware-Image für Slot 2 durch Entfernen oder Verändern von einigen Bits manipuliert werden, sodass die Signatur nicht mehr gültig ist. Dies soll durch Störeinflüsse oder Alterung im Flashspeicher gekippte Bits, aber auch Manipulationen an der gespeicherten Firmware simulieren.

Nachdem dieses manipulierte Firmware-Image auf das Gerät geschrieben wurde, muss anhand der Bootloader-Nachrichten überprüft werden, ob dieser die Manipulation erkennt.

In diesem Test sollte, wie in Test 2, die manipulierte Datei zum Schluss gelöscht werden, damit diese nicht versehentlich weiterverwendet wird.

Test 5: Erkennung einer Boot-Schleife

Im Aktivitätsdiagramms des Bootloaders (Abb. 4.3) hängt nahezu eine Hälfte der Aktivitäten davon ab, ob der Watchdog einen System-Reset ausgelöst hat. In den vorangegangenen Tests ist die Firmware jedoch immer korrekt gestartet und hat während der Ausführung den Watchdog ordnungsgemäß zurückgesetzt. Daher verwendet dieser Test in Slot 2 eine Firmware mit Versionsnummer 2, die den Watchdog nicht zurücksetzt. Dafür wird das `default` Beispiel aus RIOT als Firmware verwendet. Slot 1 ist mit einer gültigen und lauffähigen Version des `ota_update_app`-Beispiels belegt, die Firmware-Version 1 darstellt.

Nach dem Start des Mikrocontrollers werden die Nachrichten des Bootloaders ausgewertet. Dafür muss vom Bootloader zunächst erkannt und ausgegeben werden, dass zwei Firmware-Slots belegt sind und die neueste Firmware in Slot 2 gestartet wird. Nach der Zeitdauer, die für das Watchdog-Timeout eingestellt wurde, muss der Bootloader erneut gestartet worden sein. In diesem Durchlauf des Bootloaders muss die Nachricht erscheinen, dass der Watchdog die Ursache für den System-Reset war und Slot 1 gestartet wird.

Auch in diesem Test sollten, wie in den Tests 2 und 4, die von diesem Test erstellten Dateien wieder entfernt werden.

5.1.2. Trace-Matrix

In Tabelle 5.1 ist dargestellt, welches Requirement durch welchen Test abgedeckt wird. Ein Kreuz bedeutet, dass dieses Requirement expliziter Testgegenstand ist, während ein Kreuz in Klammern dafür steht, dass dieses Requirement zwar erfüllt sein muss, damit der Test erfolgreich ist, jedoch nicht Hauptgegenstand des Tests ist. Die Requirements, die in der Tabelle nicht aufgeführt werden, treffen entweder nicht auf die

Implementierung zu oder sind nicht implementiert. Außerdem wurden an das Update-Modul Anforderungen gestellt, die nur fordern, dass eine Funktion bereitgestellt wird. Diese Anforderungen werden ebenfalls nicht getestet, da allein die Interface-Spezifikation schon die Requirements abdeckt.

Req.	**Titel des Requirements**	**T1**	**T2**	**T3**	**T4**	**T5**
R7	Firmware-Updates passen zur Hardware	(x)	x			
R8	Authentizität des Firmware-Updates	(x)	x			
R9	Integrität des Firmware-Updates	(x)	x			
R12	Integrität der installierten Firmware	(x)		x	x	(x)
R20	Update-Modul: Signatur und Metadaten prüfen	(x)	x	(x)		
R22	Update-Modul: Installation	x		x		
R23	Update-Modul: Integritätsprüfung	(x)	x	(x)		
R24	Update-Modul: passende Metadaten	(x)	x	(x)		
R26	Update-Modul: Unterbrochene Installation erkennen	(x)	(x)	x		
R28	Bootloader: höchste Versionsnummer starten	x		x	x	x
R29	Bootloader: Signaturprüfung vor jedem Start	x		x	x	x
R30	Bootloader: unterbrochene Installation erkennen	(x)		x	(x)	(x)
R31	Bootloader: Erkennung einer Boot-Schleife	(x)		(x)	(x)	x
R33	Server: Update-Anfrage					

Tabelle 5.1.: Trace-Matrix: Requirements zu Tests

In der Trace-Matrix wird sichtbar, dass nicht alle aufgeführten Requirements durch einen Test abgedeckt sind. Der Update-Server (Req. **R33**) wurde in dieser Testreihe gar nicht getestet, da dieser nur ein triviales Beispiel implementiert, welches anhand der Dateinamen der Update-Dateien überprüft, ob ein Update verfügbar ist. Daher kann diese Implementierung nicht in einem Umfeld mit vielen Produkten eingesetzt werden. Außerdem ist der Update-Server zwar Teil der Prozesskette eine Firmware-Updates, aber kein Teil der für Sicherheit und Robustheit relevanten Software, die auf dem Mikrocontroller eingesetzt wird.

5.2. Realisierung der Tests

Alle Tests könnten prinzipiell auch manuell durchgeführt werden, für eine bessere Zuverlässigkeit und Reproduzierbarkeit der Testergebnisse sollten jedoch möglichst viele Schritte automatisiert werden.

Die Realisierung der Tests erfolgte durch Pythonskripte, die eine vollautomatische Durchführung ermöglichen. Für die Testdurchführung werden die Makefile-Targets aus dem `ota_update` Beispiel verwendet. Daher sind die Testskripte mit einem dazugehörigen Readme in dem selben Ordner abgelegt.

Die Interaktion mit der Beispielanwendung und dem Bootloader erfolgt über die RIOT-Shell, die auf der seriellen Schnittstelle zur Verfügung steht. Der Status des Geräts und der Erfolg der Testschritte wird auf Basis der von den Funktionen ausgegebenen Statusmeldungen ermittelt. Während das `ota_updater`-Modul relativ viele Statusmeldungen auf der RIOT-Shell ausgibt, sind in den anderen Modulen nur wenige Statusmeldungen aktiv ohne den Debug-Modus einzuschalten. Daher wurden in diesem Modulen diejenigen Meldungen, die für die Tests benötigt werden, durch Kommentare im Quellcode gekennzeichnet. Der Einfachheit halber wurde auf einen durch Präprozessordirektiven aktivierbaren Test-Modus, der Ausgaben ausschließlich für die Tests vornimmt, verzichtet.

Eine vollständige Automatisierung des Blackout-Tests ist dadurch gut möglich, da auf dem Nucleo-Board eine Steckbrücke vorgesehen ist, mit der die Spannungsversorgung des Mikrocontroller von der USB-Schnittstelle auf eine externe Spannungsquelle umgeschaltet werden kann. Für den Test kann die Steckbrücke durch einen Arduino mit einem Relais-Shield ersetzt werden, sodass die Spannungsversorgung vom Testskript aus schaltbar ist.

6. Fazit

Die Fragestellung, wie ein Firmware-Update auf IoT-Geräten sicher und robust durchgeführt wird, konnte in dieser Arbeit in großen Teilen beantwortet werden. Dafür wurde ein Update-Prozess entworfen, der die Anforderungen an Sicherheit und Robustheit erfüllt und vergleichsweise einfach umzusetzen ist. Dennoch ist der Prozess nicht auf beliebiger Hardware einsetzbar, da gewisse Sicherheitsmechanismen, wie der Schutz des Flashspeicherinhalts, aber auch relativ triviale Anforderungen, wie ausreichend Speicherplatz oder ein Watchdog, auf dem Mikrocontroller vorhanden sein müssen.

Die exemplarische Implementierung hat verdeutlicht, dass der Update-Prozess gut umsetzbar ist und prinzipiell in RIOT integriert werden kann. Bei der Implementierung hat sich jedoch herausgestellt, dass die spezielle Unterteilung des Flashspeichers auf der STM32F4-Serie einige Probleme mit sich bringt. Viele Mikrocontroller verwenden Flashspeicher, die in gleich große sogenannte Flashpages unterteilt sind. Der STM32F4 verwendet jedoch mehrere unterschiedlich große Sektoren. Dadurch kann der zur Verfügung stehende Speicher nicht so flexibel genutzt werden, wie bei der Verwendung von Flashpages. Außerdem ist es schwierig einen Treiber für RIOT zu erstellen, der beide Arten von Flashspeichern in einem Interface abstrahiert, ohne allzu große Abstriche bei der Effizienz der Speichernutzung machen zu müssen. Daher hat sich während der Implementierung auch herausgestellt, dass RIOT noch keinen Treiber für den Flashspeicher des STM32F4 enthält, sodass ein spezifischer Treiber erstellt wurde.

Die Tests konnten so implementiert werden, dass eine automatische Durchführung möglich ist und alle Anforderungen an die Implementierung des Update-Prozesses durch die Tests überprüft werden. Da die zu testende Anwendung aber auf dem Mikrocontroller ausgeführt werden muss, werden die Tests nicht besonders schnell durchgeführt. Dies ist jedoch kein Problem, weil keine menschliche Interaktion notwendig ist.

Wie in der Einleitung angedeutet, konnten nicht alle Aspekte eines Over-the-Air Firmware-Udpates behandelt werden. Insbesondere wurden die Details der Kommuni-

kation durch das Internet vernachlässigt, indem der Kommunikationskanal als in sich ausreichend sicheres und robustes System angenommen wurde.

Für den in dieser Arbeit entworfenen Prozess wurden außerdem Entscheidungen getroffen, die unter leicht veränderten Voraussetzungen zu anderen Ergebnissen führen können. Jeder Hersteller eines Produktes muss beispielsweise abwägen, ob der Speicherbedarf für mehrere Firmware-Slots durch die höhere Robustheit gerechtfertigt wird. Eine Speicherplatz sparende Alternative wäre, anstelle der jeweils zu ersetzenden Firmware-Version eine Backup-Firmware in einem separaten Speicherbereich abzulegen, die nur dafür zuständig ist im Fehlerfall ein neues Update herunterzuladen und zu installieren. Außerdem wurde angenommen, dass der Update-Prozess nicht besonders gegen Angriffe gehärtet sein muss, die nur durch Zugriff auf die Hardware möglich sind. Andererseits sind die Möglichkeiten, wie der entwickelte Prozess in diesem Bereich Einfluss nehmen kann, auch relativ gering, da ein Schutz nur durch spezielle Hardware- oder Mikrocontroller-spezifische Einstellungen erreicht werden kann.

Der entwickelte Update-Prozess ist ein Beispiel, wie Updates sicher eingespielt werden können, sofern die Anwendung selbst keine Einfallstore für Angriffe öffnet. Darauf kann jedoch nur der Hersteller des IoT-Geräts Einfluss nehmen. Es liegt ebenfalls im Zuständigkeitsbereich des Herstellers eine gute Qualitätssicherung für die Software vorzunehmen. Der Update-Prozess ist beispielsweise darauf angewiesen, dass die Hardware, auf der die in der Update-Datei enthaltene Software lauffähig ist, korrekt in den Metadaten angegeben wurde. Wenn diese Angabe nicht korrekt ist, hat der Update-Prozess nur noch wenig Spielraum diesen Fehler auszugleichen.

Außerdem müssen sich die Software- und Geräte-Hersteller bewusst werden, dass sie für ihre Software verantwortlich sind und die Kunden müssen bereit sein für Software-Qualität Geld auszugeben. Da es eine inhärente Eigenschaft des IoT ist, dass sehr viele Geräte mit dem Internet verbunden sein werden, wird es ansonsten zu großen Problemen kommen, wenn sicherheitskritische Softwarefehler in IoT-Geräten nicht behoben werden.

Die Möglichkeit Firmware-Updates auf IoT-Geräten über eine Internetverbindung automatisiert einzuspielen kann nicht vor Angriffen auf die Geräte schützen. Außerdem wird durch die Update-Schnittstelle natürlich prinzipiell die Angriffsfläche vergrößert, da das System komplexer wird. Dennoch kann davon ausgegangen werden, dass die Geräte sicherer werden, wenn der Hersteller die Möglichkeit hat Fehler in der Software zu beheben. Dafür muss diese Möglichkeit aber auch genutzt werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Binärdaten Binärdaten des kompilierten Programms/ Maschinencode. Bezeichnet in der Regel nicht den Bootloader. 18, 22–24, 49, 51

CBC Cipher Block Chaining. Betriebsart des symmetrischen Kryptografie-Algorithmus AES. 28

Firmware-Image Binärdaten eines auf dem Flashspeicher in einem Firmware-Slot installierten Updates gemäß Spezifikation in 3.3.1. 24, 44

Over the Air Übertragen von Daten über eine drahtlose Schnittstelle. Hier meist im Sinne von drahtlosen Firmware-Updates. 8

Update-Datei Datei zum Transport eines Firmware-Updates gemäß Spezifikation in Kapitel 3.3. 23, 36, 44, 49

Literaturverzeichnis

[Atm06] Atmel, San Jose, CA, USA. *Safe and Secure Firmware Upgrade for AT91SAM Microcontrollers*, 2006. Application Note doc6253, Online unter: http://www.atmel.com/Images/doc6253.pdf.

[Atm15] Atmel, San Jose, CA, USA. *Safe and Secure Firmware Upgrade via Ethernet*, 2015. Application Note AT11787, Online unter: http://www.atmel.com/Images/Atmel-42492-Safe-and-Secure-Firmware-Upgrade-via-Ethernet_ApplicationNote_AT11787.pdf.

[BEK14] Carsten Bormann, Mehmet Ersue, and Ari Keränen. Terminology for Constrained-Node Networks. RFC 7228, 2014. Online unter: https://rfc-editor.org/rfc/rfc7228.txt.

[BLS12] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library, 2012. Online unter: https://cr.yp.to/highspeed/coolnacl-20120725.pdf.

[BSI17] Kryptographische Verfahren: Empfehlungen und Schlüssellängen. Technische Richtlinie BSI TR-02102-1, Bundesamt für Sicherheit in der Informationstechnik, Bonn, Deutschland, 2017. Version 2017-01, Online unter: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf.

[Chi06] John Herbert Chiloyan. Firmware Recovery. Patent: US7043664B1, 2006. Microsoft Corporation.

[EM04] Michael A. Eskin and Steven Mair. System of and method for secure firmware update and loading of cable modem. Patent: EP1250811B1, 2004. Lakestar Semi Inc.

[GK08] Jörg Grabinger and Christian Klingelhöfer. Systemarchitektur und Verfahren zum sicheren Aktualisieren von Firmware. Patent: DE102006057708A1, 2008. Sick AG.

[Hou04] Russ Housley. Cryptographic Message Syntax (CMS). RFC 3852, 2004. Online unter: https://rfc-editor.org/rfc/rfc3852.txt.

[Hou05] Russ Housley. Using Cryptographic Message Syntax (CMS) to Protect Firmware Packages. RFC 4108, 2005. Online unter: https://rfc-editor.org/rfc/rfc4108.txt.

[HP00] John R. Hind and Marcia Lambert Peters. Methods, systems and computer program products for rule based firmware updates utilizing certificate extensions and certificates for use therein. Patent: US6976163B1, 2000. International Business Machines Corp.

[Ing17] Stefan Ingenhaag. Security portfolio gives protection from industrial and iot threats. *boards & solutions + ECE magazine*, (01/17):16–18, 2017. Online unter: http://files.iccmedia.com/magazines/basmar17/basmar17-p16.pdf.

[ITU12] Overview of the Internet of things. Recommendation ITU-T Y.2060, International Telecommunication Union, 2012. Online unter: http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=y.2060.

[KHV+16] Lukas Kvarda, Pavel Hnyk, Lukas Vojtech, Zdenk Lokaj, Marek Neruda, and Tomas Zitta. Software implementation of a secure firmware update solution in an IoT context. *Advances in Electrical and Electronic Engineering*, 14(4):389–396, 2016. Online unter: http://advances.utc.sk/index.php/AEEE/article/download/1858/1170.

[MHCK07] Gabriel Montenegro, Jonathan Hui, David Culler, and Nandakishore Kushalnagar. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, 2007. Online unter: https://rfc-editor.org/rfc/rfc4944.txt.

[Min12] Essensium NV, Mind - Embedded Software Division. *Safe upgrade of embedded systems*, 2012. Präsentationsfolien, Online unter: http://mind.be/content/Presentation_Safe-Upgrade.pdf.

[Ren12] Renesas Electronics America Inc. *Secure Firmware Update Lab Session*, 2012. Präsentationsfolien, Online unter: https://elearning.renesas.com/file.php/1/CoursePDFs/DevCon_2012/Security/BL02I_Secure_Firmware_update_LabSession_92012_BL02I_Final.pdf.

[SC09] Leonardo Steinfeld and Luigi Carro. The case for interpreted languages in wireless sensor networks. In *IESS 09 - International Embedded Systems Symposium. Langenargen, Germany*, pages 279–289. Springer, 2009.

[SH16] Frederic Stumpf and Jan Holle. Verfahren und Aktualisierungsgateway zum Aktualisieren eines eingebetteten Steuergerätes. Patent: DE102015209116A1, 2016. Robert Bosch GmbH.

[Sha11] Loren K. Shade. Implementing secure remote firmware updates. Conference paper, Allegro Software Development Corporation, 2011. Online unter: https://www.allegrosoft.com/wp-content/uploads/Secure-Firmware-Updates-Paper.pdf.

[SSD16] Eckart Schlottmann, Udo Schulz, and Liem Dang. Verfahren zur Programmierung eines Steuergeräts eines Kraftfahrzeugs. Patent: DE102015203776A1, 2016. Robert Bosch GmbH.

[STHS16] Silvie Schmidt, Mathias Tausig, Matthias Hudler, and Georg Simhandl. Secure firmware update over the air in the internet of things focusing on flexibility and feasibility. In *Internet of Things Software Update Workshop (IoTSU). Proceeding*, 2016. Online unter: https://down.dsg.cs.tcd.ie/iotsu/subs/IoTSU_2016_paper_13.pdf.

[STM12] STMicroelectronics, Genf, Schweiz. *STM32 secure firmware upgrade (SFU) overview*, 2012. Application Note AN4023, Online unter: http://www.bdtic.com/download/ST/AN4023.pdf.

[STM13] STMicroelectronics, Genf, Schweiz. *STM32F10xxx/20xxx/21xxx/L1xxxx Cortex-M3 programming manual*, 2013. Programming Manual PM0056, DocID15491 Rev 5.

[STM14] STMicroelectronics, Genf, Schweiz. *STM32L0 Series Cortex-M0+ programming manual*, 2014. Programming Manual PM0223, DocID025763 Rev 1.

[STM16] STMicroelectronics, Genf, Schweiz. *STM32F3, STM32F4 and STM32L4 Series Cortex-M4 programming manual*, 2016. Programming Manual PM0214, DocID022708 Rev 5.

[STM17] STMicroelectronics, Genf, Schweiz. *STM32F7 Series Cortex-M7 processor programming manual*, 2017. Programming Manual PM0253, DocID028474 Rev 3.

[TF16] Hannes Tschofenig and Stephen Farrell. Report from the Internet of Things (IoT) Software Update (IoTSU) Workshop 2016. Internet-Draft draft-farrell-iotsu-workshop-01, Internet Engineering Task Force, 2016. work in progress, Online unter: https://datatracker.ietf.org/doc/html/draft-farrell-iotsu-workshop-01.

[TI15] Texas Instruments, Dallas, TX, USA. *Secure In-Field Firmware Updates for MSP MCUs*, 2015. Application Report SLAA682, Online unter: http://www.ti.com/lit/an/slaa682/slaa682.pdf.

[Tom14] Shivani Tomar. White paper – secure firmware upgrade system. White paper, HCL Technologies, 2014. Online unter: https://www.hcltech.com/sites/default/files/resources/whitepaper/files/2014/06/19/design_of_secure_firmware_upgrade_system.pdf.

A. Inhalt der CD

- Dieses Dokument als PDF
- Code-Repository
  Es sind die folgenden Git-Branches vorhanden:
  - `ota_update/master`: aktueller Software-Stand dieser Arbeit
  - `master`: Kopie vom `master`-Branch des RIOT-Repository
  - `kYc0o/firmware_swapinging`: Kopie des gleichnamigen Branches aus dem RIOT-Repository des Github-Nutzers „kYc0o“ mit dem Stand zu Beginn dieser Arbeit
  - `feature/fw_swaping-f411`: Anpassung von `kYc0o/firmware_swapinging` auf das Nucleo-F411 Board
- Literatur
  Ordner mit der verwendeten Literatur.
- Readme
  Ordner mit Anleitungen zur Installation und Einrichtung von RIOT, sowie Installation der Programmiersprache Go für den beispielhaften Update-Server.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Hamburg, 01.06.2017 Jannik Beyerstedt