IoT Geräte updaten – jetzt aber mal richtig

Konzepte, Möglichkeiten und Praxiserfahrungen zum Thema Software Updates am Beispiel eines embedded Linux Systems

Florian Fischer, Helbling Technik

Mit dem Einzug von IoT Geräten in unser Leben gibt es inzwischen eine Unzahl von Geräten, die für einen sicheren Betrieb über Jahre hinweg regelmäßige Software Updates benötigen. Hierfür müssen auf Software Seite entsprechende Funktionen vorgesehen werden. Meist kommen dafür selbst entwickelte Updateprozesse zum Einsatz, die leider nicht immer die Anforderungen an sichere und zuverlässige Updates erfüllen. Im Bereich der häufig eingesetzten embedded Linux Systeme gibt es bereits diverse OpenSource Projekte, die die notwendige Funktionalität zur Verfügung stellen.

Aus Hersteller- und damit auch aus Entwicklersicht, gibt es mehrere Gründe Geräte zu aktualisieren. Embedded Linux [1] weist – wie jede komplexere Software – Bugs und Sicherheitslücken auf. Die Webseite „CVE Details“ listet zu Linux eine nicht unerhebliche Anzahl an Sicherheitslücken auf [2]. Für die meisten dieser Lücken gibt es entsprechende Patches, die über ein Update auf die Geräte verteilt werden sollten. Ebenso verhält es sich bei vielen Standard Bibliotheken. Ein Beispiel für eine Sicherheitslücke in diesem Bereich ist der Heartbleed Bug in der weitverbreiteten OpenSSL Software Bibliothek.

Neben Sicherheitslücken im Betriebssystem und den eingesetzten Bibliotheken kommt es immer wieder - trotz ausgiebiger Unit- und Integrationstests - vor, dass die eigentliche Applikation Fehler aufweist. Um den Benutzern eine ungetrübte Verwendung zu ermöglichen, sollten Updates durchgeführt werden.

Software Updates werden auch benötig, wenn der Hersteller neue Funktionen nachrüsten möchte. So rollt der Autohersteller Tesla seit kurzem ein neues Software Update aus, das u.a. die Funktion „Smart Summon“, mit der Tesla-Besitzer ihre Autos auf Entfernungen von bis zu 50 Metern zu sich rufen können [3], beinhaltet.

Eigentlich sollten Updates eine erprobte und alltägliche Aufgabe sein. In der Realität zeigt sich jedoch häufig ein anderes Bild. Wie sonst wären Schlagzeilen wie „Infotainment in Fiat-Chrysler-Auto durch Update gestört“ [4] oder „Firmware-Update zerstört smarte Türschlösser dauerhaft“ [5] möglich?

Anforderungen an ein Update

Ein Update darf ein System nicht unbrauchbar machen. Daher ist die oberste Anforderung an einen Updateprozess dessen Robustheit. Weder Stromausfall, noch eine Nutzeraktivität während eines Updates dürfen das System in einem undefinierten oder unbrauchbaren Zustand zurücklassen. Ein Update muss ein atomarer Prozess sein.

Da embedded Systeme meist limitierte Ressourcen aufweisen, sollte der Updateprozess möglichst leichtgewichtig sein und keine unnötigen Anforderungen an die Hardware stellen.

Updates dienen auch als Einfallstor für Angriffe, daher muss sichergestellt sein, dass nur autorisierte Software auf dem Gerät installiert werden kann.

Während eines Updates ist das Gerät meist nicht benutzbar, weshalb in der Regel versucht wird die benötigte Zeit zu minimieren.

Je nach Einsatzort und Gerät können noch weitere Anforderungen hinzukommen. Zusammenfassend können folgende allgemeine Anforderungen an Updates aufgelistet werden:

- Robust
- Zuverlässig
- Ressourcensparend
- Schnell
- Sicher

Update Konzepte

Es kann zwischen vier Update-Konzepten unterschieden werden [6]:

Datei-basiert: Der einfachste Ansatz ist Dateien direkt auf das Gerät zu kopieren und zu ersetzen. Dies erfüllt aber selten die Anforderungen an einen stabilen Updateprozess.

Paket-basiert: Dies kommt meist auf Desktopsystemen zum Einsatz. Beispiele dafür sind *dep*, *rpm* oder *ipkg*. Die Anwendungen werden dazu in Pakete gepackt, die unabhängig voneinander aktualisiert werden können. Hierin liegt aber auch ein gravierender Nachteil, da es unmöglich ist, alle Kombinationen im Vorfeld zu testen.

Container-basiert: Inzwischen gibt es an die Anforderungen von embedded Systeme angepasste Container Engines. Ein Vorreiter in diesem Bereich ist balenaEngine [7], vorher bekannt unter Resin.io. Ein Container beinhaltet neben der Anwendung auch alle notwendigen Abhängigkeiten. Dies vereinfacht das Update einer Anwendung enorm. Für das Betriebssystem muss aber ein anderer Updatemechanismus verwendet werden.

Image-basiert: Hier wird das System auf Partitionsebene aktualisiert. Ein Image ist das Abbild einer oder mehrerer Partitionen. Der große Vorteil dieser Methode ist der absolut gleiche Softwarestand über verschiedene Geräte hinweg bei gleicher Version. Somit kann im Vorfeld durch Tests die Lauffähigkeit auf den Geräten verifiziert werden. Der Nachteil ist der erhöhte Speicherbedarf, da eine zweite Systempartition benötigt wird und ein Reboot erforderlich ist.

Vergleich der Update Konzepte

Anhand des Vergleichs sieht man, dass nur Container und Images als sichere Updatestrategie für embedded Linux Systeme in Betracht kommen. Aufgrund der Einschränkungen des Container Ansatzes ist dieser eine gute Möglichkeit, um Anwendungen zu aktualisieren, eignet sich aber nicht für das darunterliegende Betriebssystem.

Daher wird in den meisten verfügbaren Quellen zu diesem Thema der Image-basierte Ansatz empfohlen.

	Update Größe	**Speicher-bedarf**	**Atomisch**	**Abhängig-keiten**	**Tests**	**Simpel**	**Power fail save**
Datei	klein	klein	nein	nein	schlecht	ja	nein
Paket	klein	mittel	nein	ja	sehr aufwändig	nein	nein
Container	mittel	mittel	ja	ja	gut	ja	ja
Image	groß	groß	ja	ja	gut	nein	ja

Tabelle 1: Vergleich der verschiedenen Update Konzepte

OpenSource Lösungen

Es gibt diverse Open Source Software Lösungen im Bereich Update. Im Folgenden wird eine kleine Auswahl, die Image-basierte Updates unterstützen, vorgestellt.

SWupdate [8]:

- Partition und Datei basierte Updates
- Flexibles Partitionsschema
- Lokale und Remote Updates
- Yocto Integration
- Keine integrierter Rollback Mechanismus
- Signierte und verschlüsselte Updates

Mender [9]:

- Partition-basierte Updates
- Benötigt mindestens 4 Partitionen
- Read-only Rootfs
- Lokale und Remote Updates
- Eigener Backend Server verfügbar
- Yocto Integration
- Signierte Updates
- Integrierter Rollback Mechanismus

RAUC [10]:

- Partition und Datei basierte Updates
- Flexibles Partitionsschema
- Read-only Rootfs
- Lokale und Remote Updates
- Anbindung an HawkBit Server möglich
- Yocto Integration

- Integrierter Rollback Mechanismus
- Signierte Updates

Einsatz von RAUC in der Praxis

Im Kundenauftrag realisierten wir ein Gerät für den HoReCa Bereich. Die Anforderungen waren u.a. eine zuverlässige Updatelösung für alle Bestandteile des Systems.

Neben einem Microcontroller, an den diversen Aktuatoren und Sensoren angebunden sind, kommt ein i.MX6ULL mit jeweils 512MB Flash und DDR3 RAM zum Einsatz. Der Microcontroller ist via CAN Bus an das SoM angebunden. Der Gesamtaufbau wird durch ein 7‘‘ Touch Display und ein LTE Modem komplettiert und ist in Abbildung 1 dargestellt.

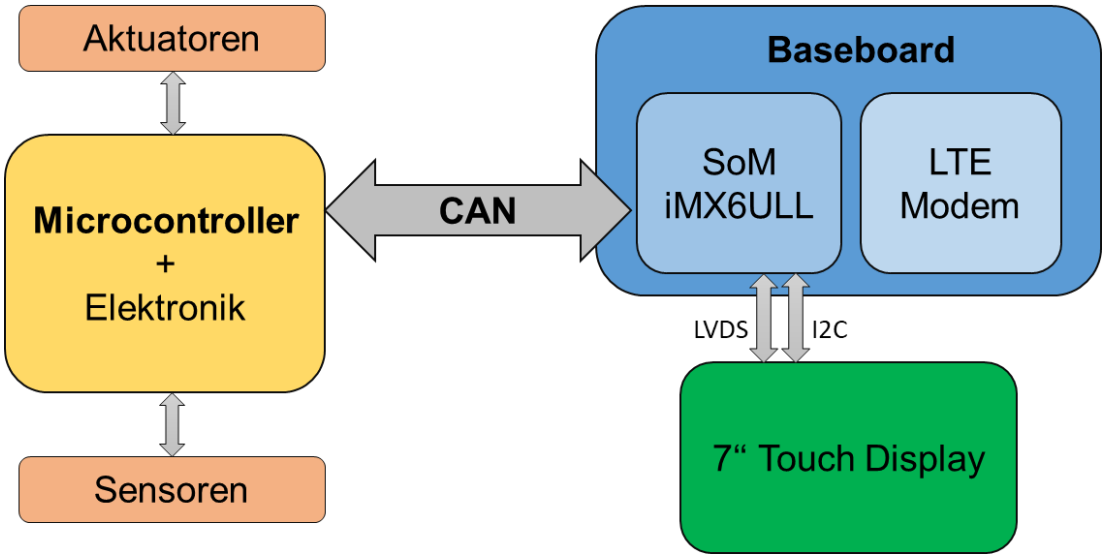

Abbildung 1: Aufbau des Gerätes

Neben dem, mit Yocto erstellten, Linux sollte auch die Microcontroller Firmware updatebar sein. Die Frequenz für Aktualisierungen wurde mit 1- bis 2-mal im Jahr angenommen. Update Images können sowohl über die AWS IoT Cloud als auch via USB zur Verfügung gestellt werden.

Auf Grundlage dieser Anforderungen entschieden wir uns für RAUC, da dies neben der Integration in Yocto auch einfache Anbindung an weitere Softwaremodule anbietet.

Für die Integration des Updateprozesses konnten vier Blöcke identifiziert werden:

- Bereitstellung des Update Paketes
- Einspielen des Update Paketes
- Betrieb
- Update des Microcontroller

Bereitstellung des Update Paketes

Image-basierte Updates beinhalten neben der Applikation auch das Betriebssystem. Da das Erstellen eines Images ein komplexer Vorgang ist, bietet es sich an ein Build System einzusetzen. Mit *meta-rauc* steht für Yocto ein Layer zur Verfügung, der neben der Installation des RAUC Clients im Zielsystem, auch das Erstellen und Signieren der Updatepakete übernimmt.

Einspielen des Update Paketes

Für das Ausführen eines Updates ist neben dem RAUC Client ein spezielles Partitionsschema notwendig. Es ist nicht möglich das aktuell in Verwendung befindliche System direkt zu aktualisieren. Daher benötigt man mindestens eine zweite Partition, in die das Update Image geschrieben werden kann.

Um einem korrupten Dateisystem vorzubeugen und die Sicherheit zu erhöhen, wird die aktive Systempartition lesend eingebunden.

In unserem Fall gibt es neben den zwei nur lesbaren Systempartitionen eine beschreibbare Partition. In diese können Applikationen ihre Daten persistent ablegen. Um wichtige maschinenspezifische Informationen wie Seriennummer oder Zertifikate während der Produktion zu speichern, ist eine vierte Partition von Vorteil, die während der Produktion schreib und lesbar, im späteren Betrieb jedoch nur lesbar eingehängt wird. Abbildung 2 zeigt die vier Partitionen im Flashspeicher des Systems.

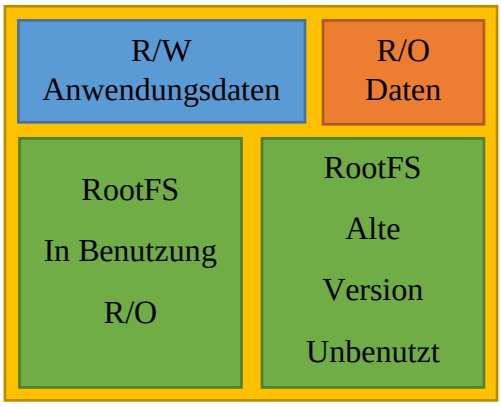

Abbildung 2: Partitionsschema

Der RAUC Client bringt keine Anbindung an einen Backend Server mit, stellt aber ein D-Bus Interface zur Verfügung, über das andere Softwaremodule Updates anstoßen können. Aufgrund der Möglichkeit ein Image über zwei Wege auf das System zu übertragen, ist für jeden Weg eine eigene Applikation entwickelt worden. OTA Updates verwenden den Mechanismus von Jobfiles der AWS IoT Platform. Nach erfolgreichem Transfer des Updatepaketes in den RAM, wird der RAUC Client aktiviert, welcher das Bundle entpackt, die Signatur überprüft und in die zweite – unbenutzte – Systempartition schreibt. War der Schreibvorgang erfolgreich, wird die Partition zum Booten vorgemerkt und ein Reboot veranlasst.

Die alte Version bleibt auf der nun inaktiven Partition erhalten und kann im Falle von Problemen als Fallback genützt werden.

Betrieb

Nach dem Gerätestart erhöht der Bootloader die Anzahl der Bootversuche. Anschließend startet er, abhängig von den bisherigen Bootversuchen, entweder in die bisherige Systempartition oder wechselt in die Fallback-Partition. Trotz Beispiel für diese Logik von RAUC, stellte die Anpassung des Bootloaders (UBoot) eine gewisse Herausforderung dar. Sollte das Booten des Betriebssystems oder der Start einer Applikation einen Fehler verursachen, wird ein Neustart durchgeführt. Nur beim fehlerfreien Start wird die Variable *Bootversuche* auf null gesetzt. Nach drei

fehlgeschlagenen Aufstart Versuchen wird in die Fallback-Partition gebootet. Abbildung 3 zeigt den gesamten Startprozess.

Update des Microcontrollers

Die drei oben aufgeführten Bereiche finden sich so oder ähnlich bei jedem beliebigen embedded Linux System wieder. In unserem Fall stellt das Update des angebundenen Microcontroller einen weiteren Teil des Updateprozesses dar. Hierzu wird nach dem Systemstart die Notwendigkeit eines Updates geprüft und dieses gegebenenfalls durchgeführt. Das Firmware Update Paket ist Teil des Images und kann nicht separat upgedated werden. Diese Kopplung von Linux/Applikation und Microcontroller Firmware verhindert mögliche Inkompatibilitäten und Abhängigkeiten.

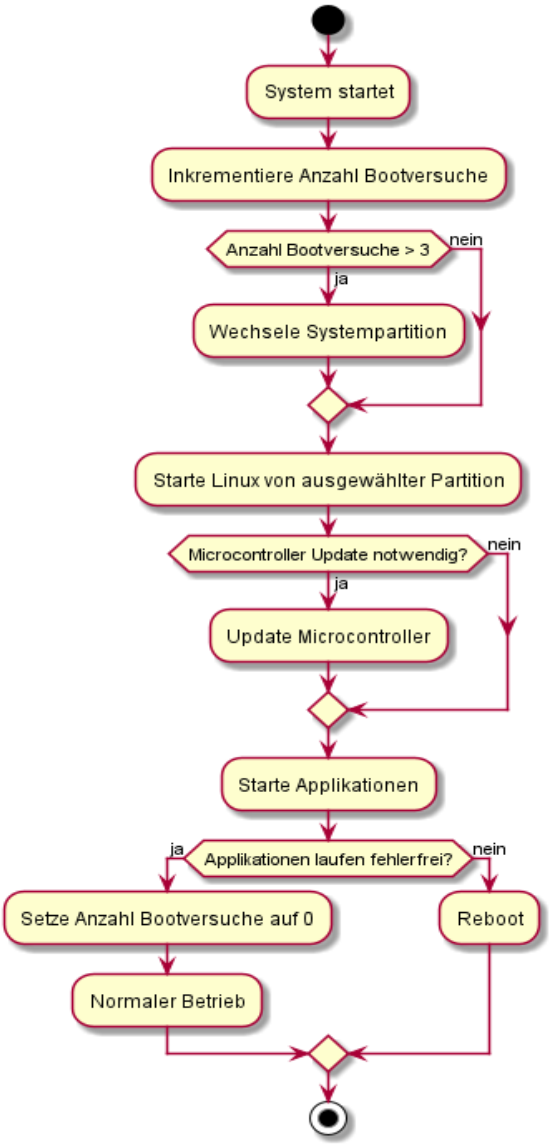

Abbildung 3: Startprozess

Fazit

Software Updates sind absolut notwendig, aber nicht trivial. Für embedded Linux Systeme existieren verschiedene Open Source Lösungen, welche aber immer Anpassungs-arbeiten erfordern. RAUC hat sich im Praxiseinsatz bewährt.

Tabellen und Abbildungsverzeichnis

Quellenverzeichnis

[1] M. Dölle, „www.heise.de,“ 18 2016. [Online]. Available: https://www.heise.de/select/ct/2016/18/1472732192870124.

[2] MITRE Corporation, „CVE Details,“ MITRE Corporation, 02 10 2019. [Online]. Available:. [Zugriff am 02 10 2019].

[3] ecomento UG, „Update auf neue Tesla-Software V10 bei normalen Kunden hat begonnen“, 27 09 2019. [Online]. Available: https://teslamag.de/news/update-auf-neue-tesla-software-v10-bei-normalen-kunden-hat-begonnen-25364. [Zugriff am 02 10 2019].

[4] A. Donath, „Golem.de,“ Golem Media GmbH, 15 02 2018. [Online]. Available: https://www.golem.de/news/uconnect-infotainment-in-fiat-chrysler-autos-durch-update-gestoert-1802-132783.html. [Zugriff am 02 10 2019].

[5] H. Gierow, „Golem.de,“ Golem Media GmbH, 13 08 2017. [Online]. Available: https://www.golem.de/news/remotelock-ls-6i-firmware-update-zerstoert-smarte-tuerschloesser-dauerhaft-1708-129458.html. [Zugriff am 02 10 2019].

[6] „Yocto Project,“ 28 03 2019. [Online]. Available: https://wiki.yoctoproject.org/wiki/System_Update. [Zugriff am 03 10 2019].

[7] „balenaEngine,“ Balena, 2019. [Online]. Available: https://www.balena.io/engine/. [Zugriff am 10 10 2019].

[8] „Embedded Software Update Documentation“ 2019. [Online]. Available: http://sbabic.github.io/swupdate/overview.html#. [Zugriff am 10 10 2019].

[9] „Mender“ 2019. [Online]. Available: https://mender.io/. [Zugriff am 10 10 2019].

[10] „RAUC,“ Pengutronix, 2019. [Online]. Available: https://github.com/rauc/rauc. [Zugriff am 10 10 2019].

Autor

Florian Fischer ist Entwicklungsingenieur bei Helbling Technik. Hier betreut und entwickelt Hr. Fischer seit 2017 embedded Software von der Architektur bis zur fertigen Implementierung hauptsächlich im IoT Umfeld.

Hr. Fischer hat einen Master in „Space Science and Technology“ erworben und bereits für das Deutsche Zentrum für Luft und Raumfahrt in Oberpfaffenhofen gearbeitet. In dieser Zeit entwickelte er einen echtzeit gesteuerten Servomotor. Anschließend wechselte er zu National Instruments und beschäftigte sich mit Messtechnik und grafischer Programmierung. Sowohl beruflich als auch privat gilt seine Leidenschaft dem Thema SmartHome und IoT.

Kontakt

Internet: www.helbling.de

Email: florian.fischer@helbling.de

LinkedIn: https://www.linkedin.com/in/florian-f-50b67a1a6