HOCHSCHULE KOBLENZ
UNIVERSITY OF APPLIED SCIENCES

GÖRLITZ

Bachelorarbeit

Firmware-Update Management für linuxbasierte, eingebettete Systeme

Autor:

Yasin Seyfe El-Islam Morsli

Matrikelnummer: 529947

Informationstechnik

Betreuender Professor:

Prof. Dr. Helmut Bollenbacher

In Kooperation mit der GÖRLITZ AG

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen, als die angegebenen Hilfsmittel benutzt habe.
Die Stellen der Bachelorarbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Koblenz, den 14.02.2020

_____

Yasin Morsli

Inhaltsverzeichnis

1 Einleitung

Moderne Geräte zur Erfassung und Übertragung von Verbrauchsinformationen benutzen internetbasierte Kommunikationsnetze und müssen dafür hohe IT-Sicherheitsstandards erfüllen. Sie besitzen ein eingebettetes Mikrocontrollersystem, auf dem eine Firmware zur Bereitstellung der Anwendungsfunktionen ausgeführt wird. Die Firmware unterliegt einer kontinuierlichen Weiterentwicklung und Pflege, beispielsweise um Sicherheitslücken zu schließen.
Damit nur authentische Firmwareversionen in die Geräte eingebracht werden können, wird bei Herstellung in jedes Gerät ein Zertifikat eingebracht mit dessen Hilfe neue, signierte Firmware-Pakete überprüft werden.
Mit dem Umstieg auf das Buildsystem "Yocto" entsteht der Bedarf zur Migration heute verfügbarer Funktionen für Firmware-Updates. Für die Bereitstellung in zukünftigen Geräten soll ein sicheres, zuverlässiges und effizientes Verfahren ausgewählt, evaluiert und exemplarisch implementiert werden.

2 Aufgabenstellung

Die Aufgabenstellungen im Umfang dieser Thesis sind:

- Firmware-Updates aus Sicht der IT-Sicherheit zu betrachten
- Die Mechanismen zur Verarbeitung und eventuellen Zurückweisung von Firmware-Artefakten durch das Zielgerät des Updates zu beschreiben
- Verfügbare Software für das Management von Firmware-Updates sowohl für das Endgerät, als auch das Entwickler-/Host-System zu evaluieren
- Protokolle und Verfahren zur Übertragung von Firmware-Artefakten unter Berücksichtigung der Effizienz in Kommunikationsnetzen mit beschränkter Bandbreite zu bewerten und auszuwählen
- Die Bereitstellung der Firmware an Nutzer sowie Release- und Rollout-Management zu betrachten
- Eine exemplarische Implementierung anhand eines praxisgerechten Aufbaus durchzuführen

3 Übersicht

In dieser Arbeit wird der Vorgang zur Einrichtung des Yocto Buildsystems mit dem Ziel, Aktualisierungsdaten durch dieses zu erstellen, durchgeführt. Das Kapitel 5 *"Das Yocto Projekt"* beschreibt das Buildsystem und die erste Einrichtung eines Betriebssystemprojekts. Daran anschließend wird der Vorgang zur Erstellung und Verwaltung von Layern des Projekts erklärt.
Um die Entwicklungsumgebung vom Host-System unabhängig zu machen, wird in Kapitel 6 *"Virtualisierung mit Docker"* das Konzept von Software Containern anhand des Docker Projekts erklärt und ein Container für die Nutzung von Yocto konfiguriert.
Für die Umsetzung eines effizienten Update-Mechanismus in Hinsicht der zu übertragenden Datenmenge werden differenzielle Updates genutzt. Um diese sinnvoll zu erstellen, muss die Entwicklungsumgebung sicherstellen, dass unveränderter Quellcode zu binär übereinstimmenden Kompilaten führen, unabhängig der genutzten Host-Maschine und dessen Konfiguration. Was dafür zu beachten ist und wie das Betriebssystemprojekt für die Reproduzierbarkeit von Kompilaten unter Yocto zu konfigurieren ist, wird in Kapitel 7 *"Reproducible Builds - Reproduzierbare Firmware-Abbilder"* beschrieben.
In Kapitel 8 *"Firmware-Update Management"* werden die mit der Implementierung eines Firmware-Update Systems verbundenen Probleme identifiziert und eine möglichst ideale Umsetzung eines Systems beschrieben. Zusätzlich werden Aspekte der Sicherheit von Firmware-Updates erklärt. Anschließend werden die für diese Arbeit gestellten Anforderungen an das zu untersuchende Projekt für die Aktualisierungsverwaltung beschrieben und evaluierte Software kurz präsentiert. Die Wahl der zu untersuchenden Software ist auf RAUC gefallen und wird in Kapitel 9 *"Firmware-Updates mit RAUC, barebox und casync"* zusammen mit den zusätzlich genutzten Projekten barebox und casync beschrieben. Im Anschluss wird die Integration der Projekte in das Betriebssystemprojekt für ein beispielhaftes Zielsystem durchgeführt.
Das konfigurierte Betriebssystem wird anhand von praktischen Anwendungen in Kapitel 10 *"Praktische Anwendung des Firmware-Update Systems"* geprüft. Die Durchführung der Tests sind alle auf der Host-Maschine durch Nutzung des Systememulators QEMU möglich.
Zum Schluss wird in Kapitel 11 ein Fazit gezogen und ein Ausblick auf die Weiterentwicklung gegeben.

4 Open Source Software

Abb. 1: Logo der Open Source Initiative (OSI) [16]

Linuxbasierte Betriebssysteme bestehen grundsätzlich komplett oder größtenteils aus "Open Source" Programmen.
"Open Source" (zd. "freie Quelle") beschreibt ein Konzept, in dem Software mit ihrem Quelltext im Rahmen von Open Source Lizenzmodellen ausgeliefert wird.
Die "Open Source Initiative" (OSI), eine der treibenden Kräfte von Open Source Software, wurde im Jahr 1998 vom Unternehmen Netscape als Ergebnis der herrschenden Konkurrenz im Softwareangebot an Internetbrowsern gegründet. Dabei wurde der Quelltext des eigenen Produkts "Netscape Navigator" freigegeben und damit die Bezeichnung "Open Source" geprägt. Die OSI ist für die Definition von Kriterien, welche Open Source Software erfüllen soll, zuständig. Definierende Merkmale von Open Source Software (gekürzt "OSS") sind:

- unkompliziertes Verteilen von Software und Quelltext im Rahmen des Lizenzmodells
- nicht-kommerzielle Einstellung
- hoher Grad an Kollaboration bei der Entwicklung
- räumlich unabhängige Verteilung der Entwickler

Diese Merkmale stehen im Gegensatz zu proprietärer Software, die in der Regel innerhalb eines Unternehmens geschlossen entwickelt wird und eine Genehmigung des Eigentümers vorliegen muss, um diese verteilen zu dürfen.
Open Source Software definiert sich durch folgende Freiheiten:

- Das Programm für jeden Zweck zu verwenden
- Das Programm und dessen Quelltext zu studieren und nach Belieben zu modifizieren
- Das Programm zu vervielfältigen, um anderen zu helfen
- Modifizierte Kopien des Programms zu verteilen

Durch diese Freiheiten wird die Zusammenarbeit an Open Source Projekten gefördert. Open Source Software kann nicht als Warenzeichen oder Schutzmarke eingetragen werden, da "Open Source" als Bezeichnung gilt. Eine Registrierung ist aber im Sinne der Open Source Gemeinschaft nötig. Um der Gemeinschaft zu helfen, wurde durch die OSI das "certification mark" als Prüfsiegel registriert. Dieses dient als Kennung für Open Source Projekte, welche die Richtlinien der OSI für Open Source Software einhalten.
Die steigende Standardisierung von freien Softwareschnittstellen und -formaten senkt die Bedeutung von proprietärer Software, was zur Folge hat, dass neben den freiwilligen Entwicklern von Projekten auch große Unternehmen die Open Source Gemeinschaft unterstützen.
Die bekanntesten Beispiele für Open Source Software sind der Linux Kernel und das GNU Projekt, welche zusammen die Basis für viele offene Betriebssysteme bilden, wie Debian, Ubuntu und Android OS.

4.1 Linux Kernel

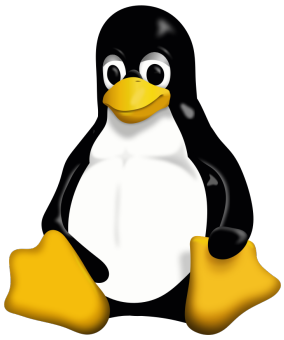

Abb. 2: Tux, Logo und Maskottchen des Linux Kernel Projekts [10]

Der Linux Kernel ist eine der meistverbreiteten Software und ist in vielen elektronischen Geräten, wie beispielsweise Computer, Smartphones, Server und Haushaltsgeräten, zu finden. Durch die Kollaboration von vielen Entwicklern, welche die Entwicklungsergebnisse gegenseitig überprüfen und testen, gilt das Projekt als eine der stabilsten und sichersten Software.
Das Linux Kernel Projekt wurde von Linus Torvalds erstmals im Jahr 1991 veröffentlicht und dabei mit dem Kommandozeileninterpreter "bash" verteilt. Heute wird der Linux Kernel zusammen mit dem GNU Projekt (Kapitel 4.2) als Basis für freie Betriebssysteme genutzt und aus dem Grund auch häufig selbst für das Betriebssystem gehalten. Der Kernel ist dabei nur ein Bestandteil des Betriebssystems, welches grundlegende Aufgabe der Abstrahierung von Hardware übernimmt und einheitliche Schnittstellen für Software bereitstellt. Damit kann Software für linuxbasierende Systeme mit der einheitlichen Schnittstellenbeschreibung entwickelt werden und ist mit anderen linuxbasierenden Systemen ohne großen Aufwand kompatibel. Der Kernel übernimmt auch die Speicher- und Prozessverwaltung, Multitasking, Lastverteilung, und die Ein- und Ausgabe an verschiedenen Schnittstellen. Das Linux Kernel Projekt ist zum größten Teil in der Programmiersprache C geschrieben und nutzt in wenigen Fällen Assembler als Quelltext, wie den Prozessorinitialisierungsroutinen. Der Kernel wird als "Monolith" entwickelt; wird der Quelltext kompiliert, liegt der Kernel mit seinen Treibern und Bestandteilen in einer einzigen Binärdatei vor. Es ist auch möglich, weitere Treiber während dem laufenden Betrieb des Systems zu dynamisch zu laden.

4.2 GNU Projekt

Abb. 3: Logo des GNU Projekts [23]

Das GNU Projekt wurde 1983 von Richard Stallman ins Leben gerufen und sollte das zu der Zeit proprietäre Betriebssystem Unix in seiner Funktion durch freie Software ersetzen. Das Akronym "GNU" ist dabei ein rekursives Wortspiel für "GNU's not Unix", welches die Beziehung zum Unix Betriebssystem klar darstellt. Heute wird das GNU Projekt im Zusammenhang mit dem Linux Kernel als Basis für freie Betriebssysteme genutzt.
Die Hauptlizenzen des Projekts sind die "GNU General Public License (GPL)" und "GNU Lesser General Public License (LGPL)". Sie sind die am meisten genutzten Lizenzen für Open Source Software. Sie schreiben im Grundsatz vor, Software, welche diese Lizenzmodelle nutzen, bei Modifizierung für die Gemeinschaft freizugeben. Ausgenommen sind Projekte, in denen der modifizierte Quelltext für den privaten Gebrauch ist und die daraus entstehende Software nicht weitergegeben wird. Diese Lizenzen helfen der Allgemeinheit, Fortschritte bei der Entwicklung von Projekten zu teilen und in der Gemeinschaft weiterzuentwickeln.
Das GNU Projekt setzt sich selbst aus verschiedenen Projekten zusammen, welche von einzelnen Personen oder ganzen Unternehmen gepflegt und gewartet werden. Unter diesen Projekten sind beispielsweise die "GNU Compiler Collection", kurz "GCC", das Standardkompilierwerkzeug für GNU/Linux Systeme, welche eine Bibliothek an Compilern für verschiedene Programmiersprachen umfasst oder die grafische Desktopumgebung GNOME,

welche als Standardoberfläche für viele GNU/Linux Betriebssysteme genutzt wird. Die modulare Struktur des Projekts erlaubt leichtes Auswechseln von integrierten Komponenten, was folglich zur heutigen Vielfalt an GNU/Linux basierten Betriebssystemen, oder "Linux Distributionen", geführt hat.

4.3 GNU/Linux Betriebssystem

Abb. 4: Logos der Debian [19], Ubuntu [1] und Arch Linux [22] Betriebssysteme

Als GNU/Linux Betriebssystem wird die Kombination des Linux Kernels mit Software aus dem GNU Projekt bezeichnet. Es wird als komplett kostenfreie Alternative zu existierenden Betriebssystemen angeboten, seiner Zeit als Ersatz für das geschlossene Unix Betriebssystem. Viele Projekte nutzen das GNU/Linux Betriebssystem als Basis, um neue "Distributionen" zu entwickeln. Diese Distributionen sind für gewöhnlich einem klaren Verwendungszweck angepasst, wie zum Beispiel die Distributionen "Debian" und "Ubuntu", welche sich durch die Vielzahl an vorkonfigurierten und installierten Paketen an Einsteiger und den alltäglichen Gebrauch richten oder "Arch Linux", welches sich durch seine vielen Konfigurationsmöglichkeiten an Power User und Entwickler richtet.
GNU/Linux Distributionen bestehen grundsätzlich aus folgenden Grundkomponenten:

- Bootloader:
  Der Bootloader initialisiert die Hardware und lädt anschließend den Kernel. Zusätzlich kann der Startvorgang durch "Kernel Commandline Arguments" konfiguriert werden.
- Kernel:
  Das der Hardware am nächsten liegende Stück Software des Betriebssystems. Er regelt den Zugriff von Software auf die CPU, den Speicher und die Peripherie.
- Init System:
  Ein Subsystem, welches die Nutzerumgebung einrichtet und Hintergrundprozesse beim Hochfahren des Betriebssystems startet. Am weitesten verbreitet ist das Init System "systemd", welches seinen Vorgänger "System V" größtenteils abgelöst hat.
- Daemons:
  Hintergrundprozesse, welche beim Start des Betriebssystems oder beim Einloggen des Nutzers geladen werden.
- Grafik-Server:
  Dieses Subsystem ist für Anzeige grafischer Oberflächen verantwortlich. Bis vor einiger Zeit wurde als Standard-Server "Xorg" genutzt. Es wurde jedoch wegen fehlenden Sicherheitsfunktionen, wie dem Isolieren von Nutzereingaben für Programme, durch "Wayland" ersetzt.
- Desktopumgebung:
  Die Software, mit der die Nutzer eines Betriebssystems grundsätzlich den meisten Kontakt haben, jedoch ist sie für ein GNU/Linux basiertes Betriebssystem nicht zwingend notwendig. Dem Nutzer stehen verschiedene Umgebungen, je nach Anwendungsfall, zur Verfügung, wie den Desktopumgebungen GNOME, KDE, Xfce, Cinnamon, Mate, und vielen mehr.
- Applikationen:
  GNU/Linux Betriebssysteme bieten eine große Vielfalt an Applikationen aus verschie-

denen Quellen. Neben den häufig in einer Distribution vorinstallierten Applikationen, können durch den Nutzer nach Belieben sowohl aus speziell für die Distribution verfügbare Paketmanager Programme nachträglich installiert werden, als auch vom Quellcode selbst für die eigene GNU/Linux Betriebssystemumgebung kompiliert und anschließend installiert werden.

5 Das Yocto Projekt

Abb. 5: Logo des Yocto Projekts [11]

Das Yocto Projekt ist eine umfangreiche Entwicklungsumgebung, welche das Ziel verfolgt, die Entwicklung eines Betriebssystems für eingebettete, linuxbasierte Systeme unter mehreren Entwicklern zu vereinfachen. Das Projekt wurde von der Linux Foundation das erste Mal 2011 veröffentlicht. Kollaboratoren sind unter anderem OpenEmbedded, Intel, Texas Instruments, Xilinx, ARM und NXP. Das Projekt unterstützt die Erstellung von Software für die als Standard geltenden Prozessorarchitekturen, wie x86/x86-64, ARM, MIPS und PowerPC. Es baut auf dem von OpenEmbedded entwickelten Werkzeug "Bitbake" auf und nutzt das "openembedded-core" Projekt, das wichtige Beschreibungen und Werkzeuge zum Bauen des Betriebssystems enthält. Projekte, welche die Yocto Entwicklungsumgebung nutzen, bringen projektspezifisch gebaute Kompilierwerkzeuge mit. So ist das Teilen der Projekte unter Entwicklern, welche verschiedene Host-Plattformen nutzen, stark vereinfacht, da diese Werkzeuge vor dem Bauen des Betriebssystems passend für die vorliegende Host-Architektur kompiliert werden. Es stellt damit auch gleichzeitig ein "SDK" ("Software Development Kit") für die Entwickler der jeweiligen Host-Plattform zur Verfügung und vereinfacht den Prozess der Entwicklung von neuer Software für die Zielplattform. Die Integration von "QEMU", ein Programm zur Emulation von ganzen Systemen, hilft beim Entwicklungs- und Testprozess von Software. Ohne direkten Zugriff auf die Zielplattform, zum Beispiel während der Konzeption einer neuen Hardware, kann diese durch QEMU emuliert werden.
Die Projektstruktur eines Betriebssystems ist in sogenannte "Layer" und den darin enthaltenen "Recipes" aufgeteilt. Das Betriebssystem soll möglichst unabhängig von der Zielplattform bleiben, weshalb für die Einbindung von hardwarespezifischer Software und Treibern sogenannte "Board-Support-Package Layer", oder kurz "BSP Layer", genutzt werden. So ist es möglich, das Betriebssystem für verschiedene Zielplattformen parallel zu entwickeln.
Jedes Projekt enthält die "Poky" Referenzimplementation als Layer, die eine Vielzahl an Grundklassen und Beschreibungen für Softwarepakete enthält. Dieser Layer wird durch weitere Layer und Recipes erweitert und angepasst.

5.1 Installation der Yocto Entwicklungsumgebung

Die Installationsanleitung ist eine inhaltliche Kurzfassung der offiziellen Anleitung [12]. Um die Yocto Entwicklungsumgebung nutzen zu können, wird ein kompatibles Linux Betriebssystem vorausgesetzt. Diese Anleitung geht davon aus, dass eine Debian 10 Installation genutzt wird. Die standardmäßig in Linux Distributionen enthaltenen Programme `git`, `python` und `tar` werden für die Installation der Yocto Entwicklungsumgebung benötigt.
Zunächst werden durch den Paketmanager fehlende Pakete installiert. Dazu wird in dem Kommandozeilenprogramm der Wahl folgender Befehl ausgeführt:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \
    build-essential chrpath socat cpio python python3 python3-pip python3-pexpect \
    xz-utils debianutils iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev \
    pylint3 xterm
```

Ist die Installation der Pakete abgeschlossen, wird das "Poky" Projekt mittels `git` heruntergeladen:

```
$ git clone git://git.yoctoproject.org/poky
```

Sind alle Daten heruntergeladen, wird das Arbeitsverzeichnis in das ”Poky” Projekt gewechselt:

```
$ cd poky/
```

Anschließend wird die gewünschte Yocto Version gewählt, in diesem Fall die Version 2.7 ”yocto-warrior”:

```
$ git checkout tags/yocto-2.7
```

Als Nächstes wird das Umgebungsskript `oe-init-build-env` ausgeführt. Das Skript passt die Umgebungsvariablen des laufenden Kommandozeilenprogramms für die Yocto Entwicklungsumgebung an und erstellt das Verzeichnis `build`. In diesem Verzeichnis befindet sich die globale Konfiguration, die auf alle Projekte innerhalb der Yocto Entwicklungsumgebung angewandt wird und das Ablageverzeichnis `tmp`, das alle Buildartefakte enthält, die durch den `bitbake` Prozess erstellt werden:

```
$ source oe-init-build-env
```

Bevor eines der Betriebssystemprojekte durch den `bitbake` Prozess gebaut wird, muss die globale Konfiguration angepasst werden. Im Rahmen dieser Arbeit soll das Betriebssystem `core-image-minimal` für die x86-64 Prozessorarchitektur gebaut werden. Dazu wird der Parameter `MACHINE` der Konfigurationsdatei `local.conf`, die sich unter `build/conf` befindet, folgendermaßen angepasst:

```
...
# This sets the default machine to be qemux86 if no other machine is selected:
MACHINE ??= "qemux86"
MACHINE ??= "qemux86-64"
...
```

Nun kann das Betriebssystemprojekt `core-image-minimal` mit dem Befehl `bitbake` gebaut werden:

```
$ bitbake core-image-minimal
```

Der Prozess zum Kompilieren des Betriebssystems wird gestartet, und kann je nach Leistung der Host-Maschine und Internetverbindung sehr lange dauern.
Ist der Prozess beendet, kann mit QEMU das gebaute Betriebssystem auf der Host-Maschine getestet werden. Dazu stellt das Yocto Projekt integrierte Skripte zur Verfügung. Um den Emulator zu starten, wird folgender Befehl genutzt:

```
$ runqemu qemux86-64
```

Ein neues Fenster öffnet sich, in dem die Kommandozeile der virtuellen Maschine angezeigt wird.

5.2 Struktur eines Yocto Betriebssystemprojekts

Das Stammverzeichnis des Projekts sieht nach dem Einrichten folgendermaßen aus:

```
$ ls -l poky/
drwxr-xr-x  6 yasin yasin 4096  bitbake
drwxr-xr-x  3 yasin yasin 4096  build
drwxr-xr-x 15 yasin yasin 4096  documentation
-rw-r--r--  1 yasin yasin  515  LICENSE
drwxr-xr-x 19 yasin yasin 4096  meta
drwxr-xr-x  5 yasin yasin 4096  meta-poky
drwxr-xr-x  9 yasin yasin 4096  meta-selftest
drwxr-xr-x  7 yasin yasin 4096  meta-skeleton
drwxr-xr-x  8 yasin yasin 4096  meta-yocto-bsp
-rwxr-xr-x  1 yasin yasin 1947  oe-init-build-env
lrwxrwxrwx  1 yasin yasin   30  README.hardware -> meta-yocto-bsp/README.hardware
-rw-r--r--  1 yasin yasin 1226  README.LSB
-rw-r--r--  1 yasin yasin  809  README.OE-Core
lrwxrwxrwx  1 yasin yasin   21  README.poky -> meta-poky/README.poky
-rw-r--r--  1 yasin yasin  529  README.qemu
drwxr-xr-x  8 yasin yasin 4096  scripts
```

Wichtige Ordner und Dateien für Entwickler sind:

- `bitbake`:
  Die mit dem Yocto Projekt kommenden Werkzeuge, um den Kompilierprozess des Betriebssystemprojekts zu verwalten.
- `build`:
  Enthält globale Konfigurationsparameter, den `sstate-cache`, der Metadaten und Zwischenkompilate aus vorigen Kompilierprozessen enthält. Zusätzlich werden alle für das Betriebssystem kompilierten Daten im darin enthaltenen `tmp` Ordner abgelegt.
- `build/conf/local.conf`:
  Konfigurationsdatei, dessen Einstellungen global auf das Projekt angewandt werden. Initial sind die wichtigsten Parameter des Projekts folgendermaßen gesetzt:

  ```
  ...
  MACHINE ??= "qemux86"
  DISTRO ?= "poky"
  PACKAGE_CLASSES ?= "package_rpm"
  EXTRA_IMAGE_FEATURES ?= "debug-tweaks"
  ...
  ```

  Diese Konfiguration erstellt das Betriebssystem für ein mit der x86 Prozessorarchitektur ausgestatteten System, nutzt das "Poky" Projekt als Grundlage und kompiliert Pakete im `rpm` Format. Zusätzlich werden mit dieser Konfiguration Debug-Features eingeschaltet, welche die Entwicklung mit dem Betriebssystem vereinfachen.
- `build/conf/bblayers.conf`:
  In dieser Datei werden alle für das Projekt verwendeten Layer konfiguriert.
- `meta-*`:
  Ordner mit dem Präfix `'meta-'` sind Layer, deren Recipes nach Einbinden in der Datei `bblayers.conf` im Projekt genutzt werden können.
- `oe-init-build-env`:
  Das Umgebungsskript zur Anpassung der laufenden Sitzung im Kommandozeilenprogramm für die Yocto Entwicklungsumgebung.

Layer sind die Bausteine des Betriebssystemprojekts. Darin enthalten sind Pakete, die selbst Recipes ihrer Komponenten und weitere Daten für den `bitbake` Prozess enthalten. Diese Struktur wurde konzipiert, um es Entwicklern leichter zu machen, existierende Layer

in das eigene Projekt einzupflegen und diese als Basis für eigene Layer zu nutzen. Es ist auch möglich Layer nach Bedarf anzupassen, ohne diese zu überschreiben. Dadurch wird zusätzlich vermieden, dass ein Layer mehrfach angelegt werden muss, um kleine Änderungen, zum Beispiel plattformspezifische Konfigurationen, durchzusetzen.
Verschiedene Quellen stellen bereits konfigurierte Layer für Yocto Betriebssystemprojekte bereit, wie die offizielle Ressource des OpenEmbedded Projekts `http://layers.openembedded.org`.
Um Layer nutzen zu können, müssen diese in einer gewissen Struktur vorliegen und eine Konfigurationsdatei enthalten. Die Struktur eines Layers wird anhand des `meta-skeleton` Layers aufgeschlüsselt. Zunächst wird die Ordnerhierarchie betrachtet:

```
meta-skeleton
├── conf
│   └── layer.conf
├── recipes-core
│   └── busybox
│       ├── busybox
│       └── busybox_%.bbappend
├── recipes-kernel
│   ├── hello-mod
│   │   ├── files
│   │   └── hello-mod_0.1.bb
│   └── linux
│       ├── linux-yocto-custom
│       └── linux-yocto-custom.bb
...
```

Das Muster zum Anlegen von Recipes lässt sich schnell erschließen: Ordner (erkennbar an der fett gedruckten Schrift) mit dem Präfix `'recipe-'` sind Pakete, deren enthaltene Komponenten in Unterordnern angelegt werden. Diese Unterordner enthalten sowohl die Recipes, erkennbar an der Dateiendung `'.bb'`, für diese Komponenten, als auch dessen spezifische Dateien, wie Konfigurationen oder Patches. `recipes-kernel` stellt ein solches Paket klar dar: Es enthält die zwei Komponenten, `hello-mod` und `linux`, welche je ein Recipe und Unterordner mit Dateien enthalten.
Es ist auch möglich, Komponenten anderer Pakete mittels Recipes mit der Endung `'.bbappend'` zu übersteuern. So lassen sich vorkonfigurierte Pakete in das Projekt einpflegen und nachträglich ändern, ohne das Quellmaterial des Layers zu modifizieren. Die Komponente `busybox`, enthalten im Paket `recipes-core`, zeigt ein solches Beispiel.
Jeder Layer muss die Konfigurationsdatei `conf/layer.conf` enthalten, welche dem `bitbake` Prozess Informationen zum Durchsuchen und Verarbeiten des Layers bereitstellt. `meta-skeleton/conf/layer.conf` enthält folgende Beschreibung:

```
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "skeleton"
BBFILE_PATTERN_skeleton = "^${LAYERDIR}/"
BBFILE_PRIORITY_skeleton = "1"

# This should only be incremented on significant changes that will
# cause compatibility issues with other layers
LAYERVERSION_skeleton = "1"

LAYERDEPENDS_skeleton = "core"

LAYERSERIES_COMPAT_skeleton = "warrior"
```

Die Variable `LAYERDIR` enthält den Pfad des Layers relativ zum Projektordner als Zeichenkette, in diesem Fall ist ihr Wert `meta-skeleton`.
Um die Konfigurationsdatei besser zu verstehen, werden zunächst häufig genutzte Zuweisungsoperationen erläutert:

- Standardzuweisung (`?=`):
  Erstellt eine nicht zuvor existierende Variable und weist ihr den Wert zu. Existiert die Variable bereits, wird sie nicht überschrieben.
- Überschreiben (`=`):
  Diese Zuweisung überschreibt die Daten einer vorher definierten Variable mit dem neuen Wert.
- Anhängen (`+=`) und Voranstellen (`=+`) mit Leerzeichen:
  Der Wert der Variable wird um den zugewiesenen Wert erweitert. Beim Erweitern wird ein Leerzeichen zwischen den alten und den neuen Wert gesetzt.
- Anhängen (`.=`) und Voranstellen (`=.`) ohne Leerzeichen:
  Der Wert der Variable wird um den zugewiesenen Wert erweitert. Beim Erweitern werden der alte und der neue Wert direkt zusammengesetzt.

Nun wird die Bedeutung jeder genutzten Variable erklärt. Folgende beeinflussen die globale Konfiguration des `bitbake` Prozesses:

- `BBPATH`:
  Eine Liste von Pfaden innerhalb des Projekts, in denen nach layer-spezifischen Konfigurationsdateien und Klassen gesucht werden.
- `BBFILES`:
  Eine Liste von Suchmustern, um Recipes mit bestimmten Namen zu finden. Die "Wildcard" (`*`) kann genutzt werden, um beliebige Zeichenketten innerhalb des Musters zu definieren. So kann beispielsweise das Suchmuster `recipes-*/*/*.bb` das Recipe `recipes-kernel/hello-mod/hello-mod_0.1.bb` finden.
- `BBFILE_COLLECTIONS`:
  Listet alle konfigurierten Layer des Projekts auf. Diese Variable wird für gewöhnlich um den Layernamen erweitert.

Die Zuweisungen dieser Variablen bleiben in den meisten Fällen für alle Layer gleich. Es ist möglich, die Werte anzupassen und damit zum Beispiel abweichende Ordnerstrukturen für Layer durch `btibake` zu verarbeiten. Jedoch wird davon abgeraten, damit Layer einer einheitlichen Struktur folgen und es Entwicklern erleichtert wird, mit diesen zu arbeiten.
Zusätzliche Variablen, die spezifisch für den Layer angelegt werden, tragen den Layernamen als Suffix. Folgende werden in diesem Fall erstellt:

- `BBFILE_PATTERN_skeleton`:
  Enthält das Suchmuster für Recipes, welche für diesen Layer genutzt werden. Zeichenketten, welche im Namen oder kompletten Pfad des Recipes enthalten sind, werden als Treffer behandelt. Die Recipes werden dazu aus der globalen Variable `BBFILES` gesucht. Damit reicht es, aus den Layernamen als Suchmuster anzugeben, um alle in dem Layer enthaltenen Recipes zu finden.
- `BBFILE_PRIORITY_skeleton`:
  Der Wert dieser Variable gibt die Priorität für Recipes des Layers gegenüber gleichnamigen Recipes in anderen Layern an. So kann bei einem Namenskonflikt von genutzten Recipes durch `bitbake` das Recipe mit der höheren Priorität gewählt werden.

- `LAYERVERSION_skeleton`:
  Gibt die Versionsnummer des Layers an. So können andere Layer diese bestimmte Version des Layers als Abhängigkeit in der Variable `LAYERDEPENDS` angeben.
- `LAYERDEPENDS_skeleton`:
  Listet alle Abhängigkeiten in Form von anderen Layern für diesen auf.
- `LAYERSERIES_COMPAT_skeleton`:
  In dieser Variable werden alle mit dem Layer kompatiblen Versionen des Yocto Projekts gelistet. Dieser Parameter ist wichtig, falls der Layer Variablen und Klassen nutzt, die nur für eine bestimmte Version der Yocto Entwicklungsumgebung verfügbar sind.

Die Verwaltung von Recipes einer Paketkomponente ist durch die Art und Weise, wie der `bitbake` Prozess das gewünschte Recipe wählt, sehr einfach und übersichtlich. Dies wird erreicht, indem Recipenamen mit einer Versionsnummer erweitert werden. So ist es möglich im gleichen Paket Recipes zu verschiedenen Versionen einer Komponente bereitzustellen. Im Folgenden wird die `hello-mod` Komponente des `recipes-kernel` Pakets betrachtet:

```
hello-mod
├── files
│   ├── COPYING
│   ├── hello.c
│   └── Makefile
└── hello-mod_0.1.bb
```

Die Struktur von Komponenten ist grundsätzlich sehr einfach gehalten: Im Ordner der Komponente `hello-mod` ist lediglich das Recipe `hello-mod_0.1.bb` und die dazu gehörenden Dateien im Verzeichnis `files` abgelegt. Der Name des Recipes enthält die optionale Versionsnummer `0.1`, die durch einen Unterstrich (`_`) erkennbar vom Basisnamen `hello-mod` des Recipes getrennt ist.
Ein Recipe enthält neben beschreibenden Parametern der Komponente auch Befehle, genannt "Tasks", die während dem Analyse- und Kompilierprozess von `bitbake` ausgeführt werden. Sie beschreiben, wie mit den Daten einer Komponente umgegangen werden müssen, um beispielsweise einen darin enthaltenen Quelltext zu kompilieren und in dem dafür vorgesehenen Zielverzeichnis im Betriebssystem abzulegen.
Das Recipe `hello-mod_0.1.bb` enthält folgende Zeilen:

```
SUMMARY = "Example of how to build an external Linux kernel module"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=12f884d2ae1ff87c09e5b7ccc2c4ca7e"

inherit module

SRC_URI = "file://Makefile \
           file://hello.c \
           file://COPYING \
          "

S = "${WORKDIR}"

# The inherit of module.bbclass will automatically name module packages with
# "kernel-module-" prefix as required by the oe-core build environment.

RPROVIDES_${PN} += "kernel-module-hello"
```

Die in diesem Recipe enthaltenen Variablen und Befehle reichen in den meisten Fällen aus, um Komponenten für die Verwendung im Projekt zu beschreiben. Im Folgenden werden häufig genutzte Variablen erläutert:

- `SUMMARY`:
  Eine kurze Beschreibung der Komponente.
- `LICENSE`:
  Der verwendete Lizenztyp.
- `LIC_FILES_CHKSUM`:
  Enthält Dateipfade zu verwendeten Lizenzdateien relativ zum Verzeichnis der Komponente und die sich aus der Datei ergebende Prüfsumme im angegebenen Format.
- `inherit`:
  Mit diesem Befehl können Klassen aus anderen, im Projekt bekannten, Layern eingebunden werden, um den Aufwand, ein Recipe zu schreiben, zu verringern. Somit müssen die verschiedenen Tasks für ein Recipe nicht erneut geschrieben werden, falls diese schon in anderen Layern vorhanden sind.
- `SRC_URI`:
  Eine Liste von lokalen und netzwerkbezogenen Ressourcen, die zum Kompilieren der Komponente benötigt werden.
- `S`:
  Das Arbeitsverzeichnis, in dem die zum Kompilieren benötigten Dateien der Komponente abgelegt werden. Dieser Variable wird grundsätzlich der Wert `WORKDIR` zugewiesen, die in einen passenden Unterordner spezifisch zur Komponente im `build` Verzeichnis des Betriebssystemprojekts verweist.
- `PN` und `PV`:
  Variablen, denen jeweils der Komponentenname (`PN`) und -version (`PV`) zugewiesen werden. Für das Recipe `hello-mod_0.1.bb` ist `PN` der Wert `hello-mod` und `PV` der Wert `0.1` zugewiesen.
- `PROVIDES` und `RPROVIDES`:
  Eine Liste von Namen, mit denen diese Komponente zur Kompilierzeit (`PROVIDES`) und Laufzeit (`RPROVIDES`) referenziert werden kann. Diese Referenzierung kann mittels der Variablen `DEPENDS` und `RDEPENDS` in einem anderen Recipe angegeben werden.
- `DEPENDS` und `RDEPENDS`:
  Diese Variablen enthalten jeweils eine Liste von Abhängigkeiten, die zum Bauen (`DEPENDS`) oder zum Ausführen (`RDEPENDS`) der Komponente benötigt werden. Durch diese Variablen wird die Reihenfolge ermittelt, in der die genutzten Komponenten durch `bitbake` analysiert und kompiliert werden.
- `FILESEXTRAPATHS`:
  Eine Liste von Suchpfaden für Dateien, die innerhalb des Recipes genutzt werden. Diese Variable enthält unter anderem standardmäßig den Suchpfad `files` relativ zum Ordner der Komponente, wodurch auch ohne Zuweisung dieser Variable in einem Recipe unter dem Verzeichnis `files` alle enthaltenen Dateien genutzt werden können.

Eine Liste und Beschreibung aller einstellbaren Variablen des `bitbake` Prozesses für die Yocto Entwicklungsumgebung Version 2.7 "yocto-warrior" ist unter `https://www.yoctoproject.org/docs/2.7/bitbake-user-manual/bitbake-user-manual.html` verfügbar.

5.3 Anlegen und Einbinden eines exemplarischen Layers

Im Folgenden wird ein Layer mit einem Recipe für ein selbst geschriebenes Programm angelegt, das in das Betriebssystemprojekt integriert werden soll. Dazu wird im Stammverzeichnis ein neuer Ordner für den Layer mit einem passenden Namen angelegt, hier `meta-thesis`:

```
$ mkdir meta-thesis
```

Der neu angelegte Layer muss dem Projekt bekannt gemacht werden. Dazu wird in der Datei `build/conf/bblayers.conf` die Variable `BBLAYERS` um den absoluten Pfad zum Layer erweitert (in diesem Beispiel ist der absolute Pfad zum Betriebssystemprojekt `/yocto`):

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  /yocto/meta \
  /yocto/meta-poky \
  /yocto/meta-yocto-bsp \
  /yocto/meta-thesis \
  "
```

Als Nächtes wird im Verzeichnis des Layers ein Ordner mit dem Namen `conf` angelegt und darin die Konfigurationsdatei `layer.conf` erstellt:

```
$ cd meta-thesis
$ mkdir conf
$ touch conf/layer.conf
```

Mit einem Editor der Wahl wird die Konfigurationsdatei mit folgendem Inhalt beschrieben:

```
BBPATH .= ":${LAYERDIR}"
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "thesis"
BBFILE_PATTERN_thesis = "^${LAYERDIR}/"
BBFILE_PRIORITY_thesis = "1"

LAYERVERSION_thesis = "1"
LAYERDEPENDS_thesis = "core"
LAYERSERIES_COMPAT_thesis = "warrior"
```

Nun wird die Struktur einer Komponente innerhalb des Layers angelegt. Dazu wird ein sinnvoll benanntes Paketverzeichnis im Ordner des Layers erstellt. Anschließend wird ein Ordner mit dem Recipe und den dazu gehörenden Daten der Komponente angelegt. In diesem Fall wird das Paketverzeichnis `recipes-example` und das Verzeichnis der Komponente `helloworld` genannt:

```
$ mkdir recipes-example
$ mkdir recipes-example/helloworld
$ mkdir recipes-example/helloworld/files
$ touch recipes-example/helloworld/helloworld_0.1.bb
$ touch recipes-example/helloworld/files/Makefile
$ touch recipes-example/helloworld/files/main.c
```

Die Komponente `helloworld` besteht aus einem Programm, das in der Programmiersprache C geschrieben ist und einem `Makefile` Skript, um den Quelltext für das Ziel-

system zu kompilieren. Folgendes Recipe reicht aus, um mit dem Programm `make` und dem `Makefile` Skript den Quelltext zu kompilieren:

```
SUMMARY = "Exemplarisches Recipe zum Einbinden eines Programms in das Betriebssystemprojekt"
LICENSE = "CLOSED"
SRC_URI = " file://Makefile \
            file://main.c \
            "
S = "${WORKDIR}"

do_install() {
    install -d ${D}${bindir}
    install -m 0755 helloworld ${D}${bindir}
}
```

`make` wird für Recipes, welche sonst keine Klassen mit `inherit` referenzieren, automatisch bei definierter `Makefile` Datei genutzt. Der Wert `CLOSED` der Variablen `LICENSE` bewirkt, dass der `bitbake` Prozess nicht nach einer Lizenz in den von der Komponente zur Verfügung gestellten Dateien sucht und deshalb keine Lizenz angelegt werden muss. Die Funktion `do_install` beschreibt, wie das resultierende Kompilat des Quelltexts im Zieldateisystem abzulegen ist. Die Datei `meta/conf/bitbake.conf` enthält Zuweisungen für die Standardpfade eines linuxbasierten Betriebssystems, welche zur Beschreibung des Zielpfades in `do_install` genutzt werden sollten. Die genutzten Variablen `${D}` und `${bindir}` bilden zusammen das Zielverzeichnis `${WORKDIR}/image/usr/bin`, welches sich im `build` Verzeichnis des Projekts befindet. Der Inhalt des Verzeichnisses wird von `bitbake` genutzt, um ein installierbares Paket aus der Komponente zu erstellen, das beim Kompilieren des Betriebssystems genutzt wird.
Der Quelltext `main.c` enthält das obligatorische ”Hello World” Programm in der Programmiersprache C, das den Satz ”Hello World!” auf der Konsole ausgibt:

```
#include <stdio.h>

int main(int argc, char** argv) {
    printf("Hello World!\n");
    return 0;
}
```

Folgende Instruktionen reichen als Inhalt der `Makefile` Datei, um den Quelltext in `main.c` zu kompilieren:

```
CC ?= gcc
CFLAGS  ?= -Wall

all: helloworld

helloworld: main.c
    $(CC) $(CFLAGS) $(LDFLAGS) -o helloworld main.c

clean:
    $(RM) main
```

Sind alle Daten angelegt, kann die `helloworld` Komponente des `recipes-example` Pakets mit `bitbake` gebaut werden. Dazu wird in der Kommandozeile das Arbeitsverzeichnis auf das Stammverzeichnis des Projekts gewechselt und der `bitbake` Prozess zum Bauen der Komponente gestartet:

```
$ cd /yocto
$ source oe-init-build-env
$ bitbake helloworld
```

Ist der Prozess beendet, lässt sich das Kompilat in einem der konfigurierten Paketmanager-kompatiblen Dateien finden. In diesem Fall wird ein `rpm` Paket als

`build/tmp/deploy/rpm/core2_64/helloworld-0.1-r0.core2_64.rpm` abgelegt. Noch ist die `helloworld` Komponente nicht im Betriebssystemprojekt eingebunden. Um sie einzubinden, kann in der globalen Konfigurationsdatei `build/conf/local.conf` folgende Zeile an das Ende der Datei hinzugefügt werden:

```
IMAGE_INSTALL += " helloworld"
```

Damit wird das Paket beim Kompilierprozess mit `bitbake` in das im Recipe konfigurierte Verzeichnis auf allen Betriebssystemprojekten installiert. Soll das Paket nur für ein bestimmtes Betriebssystem installiert werden, kann diese Zeile stattdessen in das entsprechende Recipe des Betriebssystems geschrieben werden. Beispielsweise ergibt sich für das `core-image-minimal` Betriebssystem, dessen Recipe unter `meta/recipes-core/images/core-image-minimal.bb` zu finden ist, folgende Änderung:

```
SUMMARY = "A small image just capable of allowing a device to boot."

IMAGE_INSTALL = "packagegroup-core-boot ${CORE_IMAGE_EXTRA_INSTALL}"
IMAGE_INSTALL += " helloworld"
IMAGE_LINGUAS = " "

LICENSE = "MIT"

inherit core-image

IMAGE_ROOTFS_SIZE ?= "8192"
IMAGE_ROOTFS_EXTRA_SPACE_append = "${@bb.utils.contains("DISTRO_FEATURES", \
                                  "systemd", " + 4096", "" ,d)}"
```

6 Virtualisierung mit Docker

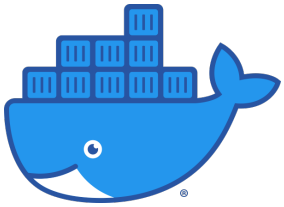

Abb. 6: Logo des Docker Projekts [4]

Docker hat sich als ein unerlässliches Werkzeug für Entwickler von linuxbasierten Applikationen etabliert. Als Werkzeug zum Erstellen und Ausführen von virtuellen Umgebungen unter Linux hilft es, den Entwicklungs- und Auslieferungsprozess von Softwareprodukten zu beschleunigen. Entwickler sind in der Lage, speziell für ihre Applikationen angepasste, linuxbasierte Umgebungen zu erstellen. Diese werden als "Images" angelegt und enthalten ein zugeschnittenes Betriebssystem, das nur diejenigen Programme und Bibliotheken enthält, die zum Entwickeln und Ausführen der eigenen Software benötigt werden. Entgegen einer vollwertigen virtuellen Maschine, bedient sich Docker an den Funktionen des Host-Betriebssystems, wie dem Linux Kernel, um keine Softwareschnittstellen emulieren zu müssen und senkt damit die Nutzung der Ressourcen enorm, die sonst nötig wären, eine virtuelle Maschine auszuführen. Durch die Nutzung von Images wird zudem das Teilen der erstellten Umgebungen unter Entwicklern stark vereinfacht, da neben der "Docker Engine" auf der Host-Plattform sonst keine weitere Software benötigt wird. So ist es auch möglich, unter Windows und macOS Docker Images zu nutzen. Jedoch wird dadurch der Punkt des gesparten Overhead negiert, da zum Nutzen von Images anderer Host-Betriebssystemen eine virtuelle Maschine im Hintergrund ausgeführt werden muss.
Docker gilt auch als eine sichere Entwicklungsumgebung für Software, da Programme eines Images innerhalb von sogenannten "Containern" gestartet werden, um diese vom Rest des Host-Systems zu trennen. Es ist möglich, den sich im Container befindlichen Prozessen kontrollierten Zugriff auf das Host-System zu gewähren. So können Applikationen in einem dafür zugeschnittenen "Sandkasten" ausgeführt und getestet werden, ohne das Host-System damit möglicherweise zu gefährden. Diese Isolation von Prozessen in Containern ermöglicht auch das parallele Ausführen mehrerer Instanzen des gleichen Images und beschleunigt so beispielsweise Testprozesse.
Im Rahmen dieser Arbeit wird die Yocto Entwicklungsumgebung innerhalb eines Docker Containers eingerichtet, um eine Basis für reproduzierbare Kompilate zu schaffen. In Kapitel 7 *"Reproducible Builds - Reproduzierbare Firmware-Abbilder"* wird weiter auf dieses Thema eingegangen und unter anderem beschrieben, welche Konfigurationen nötig sind, um mit den genutzten Entwicklerwerkzeugen reproduzierbare Firmware-Abbilder zu erstellen.

6.1 Container

Das Konzept von Containern wurden schon vor der Entwicklung von Docker für Linux Systeme eingesetzt. Sie bestehen aus Prozessen, die isoliert und kontrolliert vom Host-Betriebssystem ausgeführt werden. Ihnen stehen dafür alle nötigen Dateien, Programme und Bibliotheken in einem vom Host-System getrennten Image zur Verfügung. So haben Prozesse innerhalb des Containers keinen Einfluss auf Daten außerhalb der vom Container bereitgestellten virtuellen Umgebung. Änderungen an den im Image enthaltenen Daten werden standardmäßig auch nicht gespeichert, wodurch ein Container beim Start im-

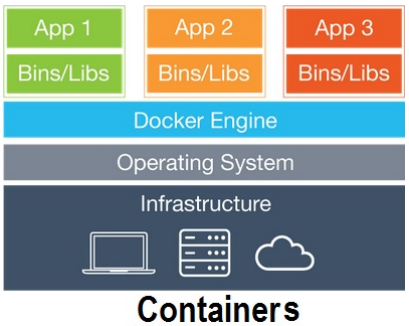

Abb. 7: Aufbau eines Docker Containers [5]

mer einen bekannten Stand besitzt. Bei richtiger Einrichtung sind Container daher auch unter dem Aspekt der IT-Sicherheit ein bevorzugtes Werkzeug. Diese Abkapselung macht Container hoch portabel und ermöglicht einfaches Teilen der Images, um so jedem Entwickler, unabhängig des genutzten Host-Systems, eine gleiche Basis für die Entwicklung einer Applikation bereitzustellen. Container eines Images können auch parallel ausgeführt werden und ermöglichen damit beispielsweise eine bessere Skalierbarkeit von Testprozessen. Die Möglichkeit, ein komplettes Betriebssystem innerhalb eines Containers einzurichten, hilft beim Aufbau von Testumgebungen und kann auch bei der Auslieferung von Applikationen genutzt werden, um Nutzern zu gewährleisten, dass diese innerhalb des Containers ordnungsgemäß funktionieren.

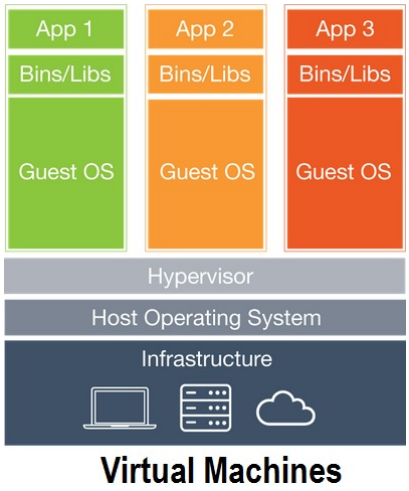

Abb. 8: Aufbau einer virtuellen Maschine [6]

Ein Container ist keine komplette Virtualisierung eines Systems. Diese wird durch Nutzen einer virtuellen Maschine erreicht. Sie führt, wie der Name vermuten lässt, ein komplett virtuelles System aus und stellt neben einer virtuellen Umgebung auch virtuelle Hardware bereit. Programme wie "VirtualBox" und "VMWare Player" spezialisieren sich auf die Virtualisierung von Betriebssystemen, um Nutzern die Möglichkeit zu bieten, ein anderes Betriebssystem neben dem Host-Betriebssystem zur gleichen Zeit auszuführen. Sie bedienen sich am sogenannten "Hypervisor", welcher für das Management und die Abstraktion von Ressourcen zwischen dem Host-System und dem Gast-System verantwortlich ist. So spielt es für das Gast-System keine Rolle, welche Hardware physikalisch zur Verfügung steht, da der Hypervisor diese für das Gast-System als kompatible Hardware bereitstellt. Diese Art von Virtualisierung benötigt viele Ressourcen, besonders Rechenleistung der CPU und Speicher, da das Gast-System als vollwertiges Betriebssystem zur Verfügung steht.

6.2 Installation und Einrichtung von Docker

Im Folgenden wird der Vorgang zur Installation und Einrichtung von Docker beschrieben. Es wird von einem Debian 10 Host-Betriebssystem ausgegangen. Der Installationsvorgang orientiert sich an der offiziellen Anleitung von Docker [3].
Zunächst werden alte Installationen von Docker entfernt, falls welche vorhanden sind:

```
$ sudo apt-get remove docker docker-engine docker.io containerd runc
```

Sind die alten Pakete entfernt, wird die Paketliste des Paketmanagers aktualisiert und eine Reihe von Paketen installiert, die für das Hinzufügen der offiziellen Docker Paketquelle nötig sind:

```
$ sudo apt-get install apt-transport-https ca-certificates \
    curl gnupg2 software-properties-common
```

Als Nächstes wird dem Paketmanager der "GPG Schlüssel" von Docker bekannt gemacht. Dieser Schlüssel hilft bei der Authentifizierung von Paketen einer Quelle und stellt sicher, dass nur verifizierte Pakete durch den Paketmanager installiert werden:

```
$ curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
```

Folgender Befehl sollte bei erfolgreichem Hinzufügen des GPG Schlüssels die untenstehende Ausgabe produzieren:

```
$ sudo apt-key fingerprint 0EBFCD88

pub   4096R/0EBFCD88 2017-02-22
        Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
uid                  Docker Release (CE deb) <docker@docker.com>
sub   4096R/F273FCD8 2017-02-22
```

Nun wird die Paketquelle der Kategorie "stable" dem Paketmanager hinzugefügt. Damit werden nur Pakete mit stabil laufender und eingehend getesteter Software durch den Paketmanager installiert:

```
$ sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/debian $(lsb_release -cs) stable"
```

Anschließend wird Docker nach einem Update der Paketliste installiert:

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Nach Abschluss der Installation kann Docker getestet werden. Dazu bezieht es aus dem "Docker Hub", einem Server, der von der Community erstellte Images zur Verfügung stellt, das Image "hello-world" und führt dieses anschließend in einem Container aus, um folgende Ausgabe zu erstellen:

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:c3b4ada4687bbaa170745b3e4dd8ac3f194ca95b2d0518b417fb47e5879d9b5f
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

6.3 Erstellung eines Containers für die Yocto Entwicklungsumgebung

Einen Docker Container als Entwicklungsumgebung für Yocto Betriebssystemprojekte zu nutzen hilft mehrere Probleme einzugrenzen, welche durch verschiedene Host-Systeme verursacht werden. Es kann beispielsweise vorkommen, dass verschiedene Konfigurationen von Host-Systemen dazu führen, dass die Kompilate gleicher Software, trotz gleichem Quelltext und Ressourcen zum Bauen des Projekts, jeweils unterschiedlich sind. Das Bauen größerer Projekte mit der Yocto Entwicklungsumgebung kann auch auf nach heutigem

Stand modernen Maschinen mehrere Stunden in Anspruch nehmen und müssen auf jedem Host-System immer neu kompiliert werden. Es ist nicht möglich, Zwischenkompilate und temporäre Daten, die sonst unter Yocto genutzt werden, um den Kompilierprozess zu beschleunigen, zwischen verschiedenen Host-Systemen zu nutzen. Durch Nutzung eines passend konfigurierten Docker Images können diese Probleme beseitigt werden.
Die sogenannte `Dockerfile` Skriptdatei enthält Befehle zur Einrichtung eines Images unter Docker. Diese Skripte können aufeinander aufbauen und ermöglichen mit wenig Aufwand eine virtuelle Umgebung zu definieren. Die folgende `Dockerfile` [8] Datei enthält alle Befehle, welche für die Einrichtung nötig sind:

```
1  FROM debian:10
2
3  ENV USERNAME    yocto
4  ENV WORKDIR     '/yocto'
5
6  ARG UID
7  ARG GID
8
9  # Test for UID and GID build-args passed to build Image, fail if not supplied
10 RUN test -n "$UID"
11 ENV UID $UID
12 RUN test -n $GID
13 ENV GID $GID
14
15 RUN apt-get update && apt-get -y upgrade
16 RUN apt-get install -y gawk wget git-core diffstat unzip texinfo gcc-multilib g++-multilib \
17         build-essential chrpath socat cpio python python3 python3-pip python3-pexpect \
18         apt-utils tmux xz-utils debianutils iputils-ping libncurses5-dev \
19         curl dosfstools mtools parted syslinux tree zip
20
21 # Create user "root"
22 RUN id root 2>/dev/null || useradd --uid 1000 --create-home root
23
24 # Create a non-root user (yocto does not allow run as root)
25 RUN id ${USERNAME} 2>/dev/null || useradd --uid ${UID} --create-home ${USERNAME}
26 RUN usermod -a -G ${GID} ${USERNAME}
27 RUN apt-get install -y sudo
28 RUN echo "${USERNAME} ALL=(ALL) NOPASSWD: ALL" | tee -a /etc/sudoers
29
30 # Set language to one with UTF-8 support
31 RUN apt-get install -y locales
32 RUN sed -i -e 's/# en_US.UTF-8 UTF-8/en_US.UTF-8 UTF-8/' /etc/locale.gen && \
33         echo 'LANG="en_US.UTF-8"'>/etc/default/locale && \
34         dpkg-reconfigure --frontend=noninteractive locales && \
35         update-locale LANG=en_US.UTF-8
36
37 ENV LC_ALL en_US.UTF-8
38 ENV LANG en_US.UTF-8
39 ENV LANGUAGE en_US.UTF-8
40
41 USER ${USERNAME}
42 WORKDIR ${WORKDIR}
43 CMD "/bin/bash"
```

Zum besseren Verständnis des Skripts werden zunächst häufig genutzte Befehle [2] erklärt:

- `FROM`:
  Dieser Befehl gibt das Grundsystem an, dessen Image als Basis für das eigene genutzt wird.
- `ENV`:
  Erstellt Umgebungsvariablen, die während und nach der Erstellung der virtuellen Umgebung zur Verfügung stehen.
- `ARG`:
  Parameter, welche beim Start vom Bauprozess an Docker übergeben werden.

- `RUN`:
  Jeder `RUN` Befehl führt die dazu angegebenen Prozesse in eigenen Containern aus und speichert nach erfolgreichem Ausführen das resultierende Abbild. Somit wird für jeden `RUN` Befehl ein eigenes Image erstellt. Dies beschleunigt den Kompilierprozess eines Images bei Änderungen auf Kosten des zur Verfügung stehenden Speichers.
- `USER`:
  Setzt den für das Image standardmäßig einzuloggenden Benutzer.
- `WORKDIR`:
  Setzt das Arbeitsverzeichnis des Containers während und nach dem Erstellprozess.
- `CMD`:
  Gibt einen Befehl, der zum Start des Containers ausgeführt wird an.

Die erste Zeile teilt Docker mit, dass Debian 10 als Grundsystem für dieses Image genutzt werden soll. Es werden anschließend die Umgebungsvariablen `USERNAME` und `WORKDIR` angelegt, die jeweils zum Festlegen des Nutzernamen und Arbeitsverzeichnisses genutzt werden.
Zwei Argumente werden zum Bauen des Images gefordert: die `UID` und `GID`, die anschließend auch als Umgebungsvariablen angelegt werden. Diese Variablen enthalten jeweils die Nutzer- und Gruppenidentifikationsnummer und werden benötigt, um die Zugriffsrechte auf Dateien und Ordner, die innerhalb und außerhalb der Containerumgebung erstellt werden, richtig zu setzen. Damit werden mögliche Zugriffsprobleme beim Arbeiten mit dem Yocto Betriebssystemprojekt vermieden. Die Zeilen 10 und 12 testen, ob die Länge des Inhalts der Argumente `UID` und `GID` größer als Null ist. Enthält eines der Argumente keinen Inhalt, wird der Bauprozess unterbrochen.
Die darauf folgenden Zeilen aktualisieren die Paketlisten des Paketmanagers und laden alle für die Yocto Entwicklungsumgebung benötigten Pakete herunter. Anschließend wird der ”root” Benutzer erstellt. Ein weiterer Benutzer muss angelegt werden, da die Yocto Entwicklungsumgebung nicht als Administrator ausgeführt werden kann. Diesem Nutzer wird als Name der Wert von `USERNAME`, in diesem Fall ”yocto”, zugeteilt. Zusätzlich wird dem Benutzer ”yocto” der Wert der Variablen `UID` als Nutzeridentifikationsnummer und der Wert von `GID` als Gruppenidentifikationsnummer zugewiesen. Diesem Benutzer werden zudem Administratorrechte mittels Eintrag in der `/etc/suders` Datei zugewiesen. Die Zeilen 31 bis 39 richten ”amerikanisches Englisch” als Systemsprache ein, um eventuelle Probleme mit der Entwicklungsumgebung wegen nicht unterstützter Zeichensätze zu vermeiden. Die letzten Zeilen richten den Benutzer ”yocto” als den für Sitzungen einzuloggenden Benutzer und den in `WORKDIR` enthaltenen Wert, hier `/yocto`, als initiales Arbeitsverzeichnis ein und starten das Programm `/bin/bash` zu Beginn einer Sitzung.
Das Image wird mit folgendem Befehl gebaut:

```
$ docker build --build-arg UID=$(id -u) --build-arg GID=$(id -g) --tag yocto:latest .
```

Die mit `--build-arg` übergebenen Argumente `UID` und `GID` werden vom aktuellen Benutzer des Host-Systems bezogen. Damit werden dem Benutzer innerhalb des Containers die gleichen Identifikationsnummern erteilt. Auf diese Weise werden Probleme mit dem Zugriffsmanagement auf Dateien und Ordner unter Linux vermieden, denn die Nutzer- und Gruppenidentifikationsnummer geben an, welchen Nutzern der Zugriff auf diese gewährt werden. Der Parameter `--tag` weist dem Image einen Namen und eine Version in Form von `<name>:<version>` zu.
Um das gebaute Image in einem Container auszuführen wird folgender Befehl genutzt:

```
$ docker run -ti -v $(readlink -f /yocto):/yocto yocto:latest
```

Der Parameter `-ti` wird übergeben, um eine interaktive Kommandozeile durch den Container zu starten, in der dann Befehle eingegeben werden können. `-v` wird genutzt, um ein sogenanntes "Volume" im Container einzuhängen. Dieses Volume ist ein auf dem Host-System befindlicher Ordner, der im Container unter dem angegebenen Pfad zu finden ist. Die Angabe des Parameters ist im Format `-v <host_pfad>:<container_pfad>` (beide Pfade sind als absolute Pfade anzugeben und ein Doppelpunkt trennt die Angaben). Der Befehl `readlink -f` gibt für einen Pfad (relativ oder absolut) den absoluten Pfad im Dateisystem zurück.
Ist der Container gestartet, sollte das in Kapitel 5.1 *"Installation der Yocto Entwicklungsumgebung"* erstellte Projekt als Arbeitsverzeichnis gesetzt und der Inhalt des Ordners zu sehen sein:

```
yocto@1349fce207f3:/yocto$ ls
bitbake        LICENSE          meta-poky      meta-yocto-bsp     README.LSB      README.qemu
build          meta             meta-selftest  oe-init-build-env  README.OE-Core  scripts
documentation  meta-thesis      meta-skeleton  README.hardware    README.poky
```

Es ist unter Umständen nötig, das `build` Verzeichnis zu entfernen, falls dieses außerhalb des Docker Containers erstellt wurde, da `bitbake` sonst Probleme mit dem Auflösen von Pfaden interner Skripte hat:

```
$ rm -rf build
```

Nun kann innerhalb des Containers das Umgebungsskript `oe-init-build-env` ausgeführt werden. Anschließend werden die Konfigurationsdateien `build/conf/local.conf` und `build/conf/bblayers.conf` nach der Beschreibung in Kapitel 5.1 *"Installation der Yocto Entwicklungsumgebung"* geändert.
Wird `bitbake` zum Bauen des Projekts genutzt, werden alle Kompilate durch Einhängen des Volumes auch dem Host-System mit korrekt gesetzten Zugriffsrechten zugänglich gemacht. Vorteilhaft an dieser Einrichtung ist auch, dass das Projekt hostseitig mit allen zur Verfügung stehenden Werkzeugen weiterentwickelt werden kann und innerhalb des Containers der Bauprozess mit der definierten Konfiguration ausgeführt wird.

7 Reproducible Builds - Reproduzierbare Firmware-Abbilder

Abb. 9: Logo des Reproducible Builds Projekts [20]

Möchte man die Differenz an Daten zwischen Firmware-Versionen möglichst gering halten, muss man sicherstellen, dass bei der Erstellung der Firmware-Artefakte durch die Entwicklungsumgebung nur Änderungen in Kompilaten zu finden sind, deren Quellcode geändert wurde.
"Reproducible Builds" beschreibt den Vorgang, binär identische Kompilate zu erhalten, unter der Voraussetzung, dass der gleiche Quellcode, die gleichen Entwicklungswerkzeuge und die gleichen Kompilieranweisungen genutzt werden. Sind diese Bedingungen erfüllt, sollte mehrmaliges Kompilieren vom Quellcode auf verschiedenen Host-Systemen zu identischen Kompilaten führen.
"Reproducible Builds" ist auch der Name des Projekts, welches Definitionen zum Thema "Reproduzierbarkeit von Kompilaten" und Hinweise zum Erreichen von binärer Reproduzierbarkeit bereitstellt.
Es gibt verschiedene Motivationen, binäre Reproduzierbarkeit von Kompilaten eines Projekts erreichen zu wollen. Aus sicherheitstechnischer Sicht kann auf diese Weise sichergestellt werden, dass die durch den Kompilierprozess erstellten Kompilate nur das beinhalten, was durch den Quellcode vorgesehen ist. Wird beispielsweise Software mit schädlichem Code versehen, geschieht dies in der Regel über kompromittierte Entwicklungswerkzeuge, wie dem Compiler oder Linker, statt durch Änderungen am Quellcode der Software selber. So bleibt dieser Angriff unbemerkt, da die Analyse von Kompilaten in der Regel viel Aufwand voraussetzt. Ein anderer Grund ist das Testen und die Qualitätssicherung von Software. Es kann vorkommen, dass bei nicht reproduzierbaren Kompilaten Probleme beim Ausführen auftreten, die nicht durch den Quellcode selber, sondern durch Optimierungen am Code durch den Compiler eingeführt werden. Diese Optimierungen basieren häufig auf nicht deterministischen Mechanismen des Compilers und sind daher schwierig zu ermitteln.
Werden Aktualisierungsdaten als differenzielle Firmware-Updates bereitgestellt, hilft die Reproduzierbarkeit der Komponenten dabei, die Differenz der verschiedenen Firmware-Versionen zu verringern. Damit wird der auf dem Server benötigte Speicherplatz für die Bereitstellung der Daten und die zu übertragende Datenmenge auf das Zielsystem verringert. Es kann auch die Dauer der Kompilation von Firmware-Abbildern verringern, da nur geänderte Komponenten erneut kompiliert werden müssen.

7.1 Probleme bei der Erstellung reproduzierbarer Firmware-Abbilder

Das Konzept von "Reproducible Builds" wird nicht standardmäßig bei der Erstellung von Kompilaten umgesetzt. Entwicklungswerkzeuge müssen in der Regel durch Setzen von Parametern und Umgebungsvariablen des Systems zur Reproduzierbarkeit von Kompilaten eingerichtet werden.
Bei der Arbeit mit einer Entwicklungsumgebung zur Erstellung von Firmware-Abbildern arbeiten viele verschiedene Programme zusammen, um die Kompilate und Daten für ein Abbild zu erstellen. Aus dem Grund verteilen sich die zu behandelnden Probleme auf mehrere Quellen. Zum Erstellen von reproduzierbaren Abbildern mit der genutzten Entwicklungsumgebung müssen folgende Punkte beachtet werden:

- Festgelegte/Bestimmte Entwicklungsumgebung:
  Die Entwicklungsumgebung muss zum Zeitpunkt der Erstellung von Kompilaten bekannt und festgelegt sein. Aus welchen Komponenten sich die Entwicklungsumgebung definiert ist vom Projekt abhängig. Häufig wird die Entwicklungsumgebung durch das genutzte Betriebssystem, die eingerichteten Systemvariablen, die genutzte Version der Entwicklerwerkzeuge und in seltenen Fällen auch die Hardware des Host-Systems bestimmt. Wird bei der Erstellung eines Kompilats auch die vorgesehene Entwicklungsumgebung genutzt, kann garantiert werden, dass ein binär kompatibles Kompilat bei gleichem Quellcode erstellt wird.
- Deterministischer Kompilierprozess:
  Der Prozess, mit dem Kompilate erstellt werden, muss in einer nachvollziehbaren und einheitlichen Art und Weise arbeiten. Der Prozess muss von umgebungsabhängigen Variablen, wie dem genutzten Betriebssystem und installierten Paketversionen von genutzter Software, möglichst unabhängig sein. Dynamisch bezogene Daten, wie die Buildnummer und der Kompilierzeitpunkt, dürfen nicht in den Quellcode eingeführt werden, da diese sich bei jedem erneuten kompilieren des Codes ändern. Die Parallelisierung des Kompilierprozesses kann auch zu einem nicht deterministischen Kompilat führen und wird daher nicht empfohlen.
- Verfahren zur Prüfung von Kompilaten:
  Wird ein Verfahren zur Prüfung der Reproduzierbarkeit von Kompilaten bereitgestellt, sind Entwickler in der Lage zu bestimmen, ob die Entwicklungsumgebung korrekt eingerichtet ist und die Kompilate erwartungsgemäß erstellt werden. Grundsätzlich ist das Ziel bei der Erstellung von reproduzierbaren Kompilaten, dass diese genau den gleichen Inhalt besitzen, wenn der gleiche Quellcode beim Kompilieren genutzt wird. Es ist jedoch möglich, dass die Gleichheit der Kompilate bei einem Projekt nicht vorausgesetzt wird, sondern nur bestimmte Muster in Kompilaten zu finden sein müssen. Aus diesem Grund muss ein Verfahren zur Prüfung der Reproduzierbarkeit von Kompilaten definiert sein.

Da das Meiden von nicht deterministischen Kompilaten stark von der Entwicklungsumgebung und den verwendeten Entwicklerwerkzeugen abhängig ist, gibt es kein einheitliches Verfahren zum Erreichen von reproduzierbaren Kompilaten. In der Annahme, dass die meisten Komponenten eines Betriebssystemprojekts auf Quellcode der Programmiersprachen C und C++ basieren, werden im Folgenden einige zu beachtende Punkte zum Erreichen von reproduzierbaren Kompilaten anhand von Beispielen mit dem Compiler `gcc` und der Programmiersprache C/C++ behandelt (Erkenntnisse lassen sich auf andere Programmiersprachen wie Java und C# übertragen):

- Meiden von Zeitstempel des Kompilierzeitpunkts:
  Informationen, welche den Zeitpunkt des Kompilierens enthalten, sollten aus Kompilaten entfernt werden, da mit jedem erneuten Kompilieren des Quellcodes sich der Zeitpunkt entsprechend ändert. Es ist möglich, dass der genutzte Compiler der Entwicklungsumgebung diese Zeitstempel standardmäßig in Kompilate einführt. In der Regel lässt sich dieser aber mit einem festen Wert übersteuern.
  Beispielsweise ist es möglich, in C und C++ Quellcode, durch die Verwendung der Makros `__TIME__` und `__DATE__`, Zeitstempel zur Kompilierzeit in Kompilate schreiben. Zur Veranschaulichung wird folgender Quellcode betrachtet:

```c
#include <stdio.h>
int main(int argc, char** argv) {
    printf("Zeitstempel: %s, %s\n", __DATE__, __TIME__);
    return 0;
}
```

Die Makros `__DATE__` und `__TIME__` sind Zeichenketten, die mit der Funktion `printf` ausgeben werden. Wird der Quellcode (hier `date_time.c`) mittels `gcc` kompiliert und das entstehende Programm ausgeführt, wird folgende Ausgabe erstellt:

```
$ gcc -o date_time date_time.c
$ ./date_time
Zeitstempel: Feb  3 2020, 19:41:09
```

Wie der Ausgabe zu entnehmen ist, wurde das Programm am 03. Februar 2020, um 19:41:09 Uhr, kompiliert. Wird der gleiche Quellcode nochmal kompiliert und ausgeführt, wird folgende Ausgabe durch das Programm erstellt:

```
$ gcc -o date_time date_time.c
$ ./date_time
Zeitstempel: Feb  3 2020, 19:41:15
```

Wie erwartet, hat sich die Ausgabe geändert, ohne dass der Quellcode geändert wurde. Um die Makros auf einen festen Wert zur Kompilierzeit zu setzen, kann beim Aufruf des Compilers die Umgebungsvariable `SOURCE_DATE_EPOCH` mit einem bekannten Wert überschrieben werden. Diese Variable enthält das aktuelle Datum in Sekunden seit dem 1. Januar 1970, um 00:00:00 Uhr. Wird diese Variable überschrieben, werden die Werte der Makros `__DATE__` und `__TIME__` entsprechend gesetzt. Im Folgenden wird beim Aufruf von `gcc` die Umgebungsvariable `SOURCE_DATE_EPOCH` auf den Wert 0 gesetzt und das Kompilat ausgeführt:

```
$ SOURCE_DATE_EPOCH=0 gcc -o date_time date_time.c
$ ./date_time
Zeitstempel: Jan  1 1970, 00:00:00
```

- Meiden von Metadaten des Quellcodes in Kompilaten:
  Häufig werden Metadaten des Quellcodes in kompilierte Programme kodiert, um unter anderem zur Laufzeit bessere Debuginformationen über den ausgeführten Code ausgeben zu können. Problematisch ist, dass diese Metadaten abhängig von Variablen wie der Host-Maschine und dem Pfad des Projekts sind.
  In C/C++ kann zum Beispiel der Pfad einer Quellcodedatei mit dem Makro `__FILE__` in das Kompilat an der genutzten Stelle geschrieben werden. Folgender Quellcode zeigt die Verwendung des Makros:

```c
#include <stdio.h>
int main(int argc, char** argv) {
    printf("Datei: \'%s\'\n", __FILE__);
    return 0;
}
```

Das Makro `__FILE__` ist eine Zeichenkette, welche den Pfad der Datei enthält, die beim Aufruf von `gcc` kompiliert wurde. Im Folgenden wird die Quellcodedatei `file.c` mit dem relativen Pfad `./` durch `gcc` kompiliert und anschließend ausgeführt:

```
$ gcc -o file ./file.c
$ ./file
Datei: './file.c'
```

In der Ausgabe wird der Pfad der Quellcodedatei als `'./file.c'` ausgegeben. Wird nun `gcc` mit dem absoluten Pfad der Quellcodedatei aufgerufen und anschließend das Kompilat ausgeführt, ergibt sich folgende Ausgabe:

```
$ gcc -o file /home/yasin/Documents/Thesis/misc/reproducible/file/file.c
$ ./file
Datei: '/home/yasin/Documents/Thesis/misc/reproducible/file/file.c'
```

Der Pfad der Datei hat sich entsprechend dem Aufruf von `gcc` geändert.
Um den Pfad im `__FILE__` Makro auf einen stets bekannten Wert zu setzen, kann das `gcc` Compilerflag `-fmacro-prefix-map` genutzt werden. Das Compilerflag `-fmacro-prefix-map=<alt>=<neu>` ersetzt im `__FILE__` Makro die Zeichenkette `<alt>` mit `<neu>`. So kann beispielsweise der Pfad der Quellcodedatei auf eine leere Zeichenkette gesetzt werden. Folgender Aufruf von `gcc` kompiliert den Quellcode so, dass das `__FILE__` Makro auf eine leere Zeichenkette gesetzt wird:

```
$ gcc -fmacro-prefix-map="file.c"="" -o file file.c
$ ./file
Datei: ''
```

Wie erwartet ist der Pfad der Datei eine leere Zeichenkette. Innerhalb einer Entwicklungsumgebung wird der Dateipfad in Variablen durchgereicht, die sich unter Umständen abfragen und ändern lassen. Im Folgenden wird demonstrativ die Umgebungsvariable `FILE` erstellt und der absolute Pfad der zu kompilierenden Datei zugewiesen. Beim Aufruf von `gcc` wird der komplette Pfad der Datei mit dem Dateinamen ersetzt. Das Programm `basename` extrahiert unter anderem aus einem Dateipfad den Namen einer Datei:

```
$ FILE="/home/yasin/Documents/Thesis/misc/reproducible/file/file.c"
$ gcc -fmacro-prefix-map="$FILE"="$(basename $FILE)" -o file $FILE
$ ./file
Datei: 'file.c'
```

- Bestimmen der Reihenfolge der zu verarbeitenden Dateien:
  Es muss eine feste Reihenfolge in der Verarbeitung von Dateien bestimmt werden. Wird keine bestimmt, kann es dazu kommen, dass Dateien in einer unbekannten Reihenfolge verarbeitet und in das Kompilat geschrieben werden.
  Das Problem äußert sich beispielsweise bei der Nutzung verschiedener "Locale" Einstellungen des Betriebssystems, die unter anderem für die Wahl der Systemsprache und des Zeichensatzes zuständig sind. Diese Einstellungen lassen sich mit der Umgebungsvariablen `LC_ALL` übersteuern.
  Es wird Beispielhaft ein Projekt mit den Dateien `fach.c`, `fadian.c` und `order.c` betrachtet. Zunächst werden die Dateien mit dem deutschen Locale `de_DE.UTF-8` gelistet. Folgender Befehl gibt alle Dateien mit der Endung `'.c'` aus (`bash -c '<befehl>'` führt Befehlsketten in einer Subshell aus, dessen Locale das mit `LC_ALL` eingestellte ist):

```
$ LC_ALL=de_DE.UTF-8 bash -c 'ls *.c'
fach.c  fadian.c  order.c
```

Das Programm `ls` sortiert standardmäßig seine Ausgabe nach den Namen der gefundenen Dateien. Die Wildcard (`*`) wird als beliebige Zeichenkette interpretiert und ist daher für die Dateien `fach.c`, `fadian.c` und `order.c` gültig. Die durch die Wildcard gefundenen Zeichenketten werden in sortierter Reihenfolge an Programme übergeben.

Wird das US-amerikanische Locale `en_US.UTF-8` genutzt, ist die ausgegebene Reihenfolge der Dateien die gleiche:

```
$ LC_ALL=en_US.UTF-8 bash -c 'ls *.c'
fach.c  fadian.c  order.c
```

Nun werden die Dateien mit dem tschechischen Locale `cs_CZ.UTF-8` gelistet:

```
$ LC_ALL=cs_CZ.UTF-8 bash -c 'ls *.c'
fadian.c  fach.c  order.c
```

Wie man erkennen kann, hat sich die Reihenfolge der Dateien geändert. Grund dafür ist, dass das Locale auch vorgibt, nach welchen Kriterien die Sortierung von Zeichenketten durchgeführt wird. Im tschechischen Locale wird die Zeichenkombination `'ch'` als ein eigenes Zeichen interpretiert und nach dem Zeichen `'h'` einsortiert. Die folgende Ausgabe zeigt, wie Zeichen mit dem `sort` Befehl sortiert werden (`sort` sortiert Zeilenweise, daher werden die Leerzeichen der Eingabe mit dem Befehl `tr` durch Zeilenumbrüchen ersetzt):

```
$ LC_ALL=cs_CZ.UTF-8 bash -c "echo 'b a c g d f e ch i h'|tr ' ' '\n'|sort|tr '\n' ' '"
a b c d e f g h ch i
```

Zum Vergleich wird der gleiche Befehl mit dem deutschen Locale ausgeführt:

```
$ LC_ALL=de_DE.UTF-8 bash -c "echo 'b a c g d f e ch i h'|tr ' ' '\n'|sort|tr '\n' ' '"
a b c ch d e f g h i
```

Nutzt man `gcc` zum Kompilieren der Projektdaten, wird der generierte Code in der auftretenden Reihenfolge der angegebenen Quellcodedateien in das Kompilat geschrieben. Der Quellcode der Datei `order.c` zeigt, welchen Einfluss die Reihenfolge der Quellcodedateien auf das Kompilat hat:

```
#include <stdio.h>

int fach();
int fadian();

int main(int argc, char** argv) {
    printf("Adresse fach %c fadian\n", (&fach > &fadian) ? '>' : '<');
    return 0;
}
```

Das Programm vergleicht die Adressen der Funktionen `fach` und `fadian` und gibt aus, welcher der beiden entsprechend der niedrigere und der höhere Adresswert beim Kompilieren zugewiesen wurde. Die Dateien `fach.c` und `fadian.c` enthalten jeweils die Funktionsdefinition der in `order.c` deklarierten Funktionsprototypen:

```
// fach.c
int fach() { return 42; }
```

```
// fadian.c
int fadian() { return 24; }
```

Das Programm wird im Folgenden jeweils mit dem deutschen und dem tschechischen Locale durch `gcc` kompiliert und anschließend ausgeführt:

```
$ LC_ALL=de_DE.UTF-8 bash -c 'gcc -o order *.c'
$ ./order
Adresse fach < fadian
$ LC_ALL=cs_CZ.UTF-8 bash -c 'gcc -o order *.c'
$ ./order
Adresse fach > fadian
```

Aus der Ausgabe lässt sich schließen, dass mit dem deutschen Locale die Funktion `fach` vor der Funktion `fadian` in das Kompilat geschrieben wird und entsprechend einen kleineren Adresswert hat. Mit dem tschechischen Locale wird die Funktion `fach` nach der Funktion `fadian` in das Kompilat geschrieben und hat daher einen größeren Adresswert. Wie erwartet ist die Ausgabe wegen der geänderten Reihenfolge der an `gcc` übergebenen Dateien unterschiedlich.
Um dieses Problem zu meiden, gibt es verschiedene Möglichkeiten. Eine wäre, das Locale mit der Umgebungsvariable `LC_ALL` während dem Kompilierprozess mit einem für das Projekt definierten Locale zu übersteuern. Eine Andere ist, die Dateien explizit an den Compiler zu übergeben, um so unabhängig der Locale Einstellung die Reihenfolge der Dateien festzulegen:

```
$ LC_ALL=de_DE.UTF-8 bash -c 'gcc -o order order.c fach.c fadian.c'
$ ./order
Adresse fach < fadian
$ LC_ALL=cs_CZ.UTF-8 bash -c 'gcc -o order order.c fach.c fadian.c'
$ ./order
Adresse fach < fadian
```

Es gibt weitaus mehr Punkte, die bei der Erstellung eines Betriebssystems durch eine Entwicklungsumgebung beachtet werden müssen, wie zum Beispiel der Wahl eines deterministischen Archivformats oder die Nutzung zufällig generierter Werte für die initiale Nutzereinstellungen. Auf der Webseite des Reproducible Builds Projekts werden weitere bekannte Probleme und mögliche Lösungen dokumentiert: `https://reproducible-builds.org/docs`.

7.2 Reproduzierbare Firmware-Abbilder mithilfe von Yocto erstellen

Das Yocto Projekt wird seit Beginn der Entwicklung darauf ausgelegt, reproduzierbare Firmware-Abbilder zu erstellen. Da die Implementierung der Funktionen für Reproduzierbarkeit jedoch aktuell nicht garantieren kann, dass das mehrmalige Kompilieren eines Image binär kompatibel bleibt, ist die Funktionalität nicht als Standardeinstellung umgesetzt. Die Funktion lässt sich jedoch mit einer übersichtlichen Zahl an Parametern der globalen Konfiguration eines Betriebssystemprojekts einschalten.
Um das Poky Projekt zur Erstellung von reproduzierbaren Firmware-Abbildern einzurichten, wird die globale Konfigurationsdatei `build/conf/local.conf` im Projektverzeichnis um folgende Parameter erweitert:

```
...
# CONF_VERSION is increased each time build/conf/ changes incompatibly and is used to
# track the version of this file when it was generated. This can safely be ignored if
# this doesn't mean anything to you.
CONF_VERSION = "1"

# Make images reproducible
BUILD_REPRODUCIBLE_BINARIES = "1"
LDCONFIGDEPEND=""
IMAGE_CMD_TAR="tar -v --sort=name"
export PYTHONHASHSEED = "0"
export PERL_HASH_SEED = "0"
export TZ = 'UTC'
export SOURCE_DATE_EPOCH ??= "0"
REPRODUCIBLE_TIMESTAMP_ROOTFS ??= "0"
```

Der Parameter `BUILD_REPRODUCIBLE_BINARIES` deaktiviert Funktionen des `bitbake` Prozesses, die zu nicht deterministischen Kompilaten führen. Ein leerer String als Wert

für `LDCONFIGDEPEND` schaltet die Erstellung einer Cache Datei für den dynamischen Linker von Linux ab. Dieser Cache wird in der Regel mit unsortierten Einträgen erstellt und ist daher auch nicht deterministisch. Um das Archiv mit dem kompilierten Betriebssystem deterministisch zu erstellen, werden die Einträge namentlich sortiert. Dafür wird die von `bitbake` zum Archivieren von Daten genutzte Variable `IMAGE_CMD_TAR` für den Aufruf von `tar` um das entsprechende Argument erweitert. `PYTHONHASHSEED` und `PERL_HASH_SEED` beeinflussen jeweils den Zufallszahlengenerator der Interpretersprachen Python und Perl. Wird den Variablen ein bekannter Wert zugewiesen, ist die resultierende Zahlenfolge, die durch den jeweiligen Zufallszahlengenerator erstellt wird, auch bekannt. Die letzten drei Parameter sind für die Erstellung von Zeitstempel verantwortlich. `TZ` setzt die Zeitzone für alle ausgeführten Programme einheitlich auf `UTC`, `SOURCE_DATE_EPOCH` setzt für alle Programme, die diese Umgebungsvariable zum Ermitteln der Zeit nutzen, auf den Wert 0 und `REPRODUCIBLE_TIMESTAMP_ROOTFS` wird genutzt, um Zeitstempel, die in einem Image bei der Erstellung eingetragen werden, auch auf den Wert 0 zu setzen.
Um die Konfiguration zu testen, werden vier unveränderte Kopien des Poky Projekts genutzt, die unabhängig voneinander gebaut werden. Die Kompilierprozesse werden dafür in vier unabhängigen Instanzen des im Kapitel 6.3 *"Erstellung eines Containers für die Yocto Entwicklungsumgebung"* eingerichteten Docker Containers durchgeführt.
Zunächst wird eine saubere Kopie des `warrior` Branch (Version 2.7) vom Yocto Projekt in einen geeigneten Ordner heruntergeladen:

```
$ git clone git://git.yoctoproject.org/poky -b warrior
```

Das Yocto Projekt wird durch `git` in das Verzeichnis `poky` heruntergeladen. Anschließend werden vier sinnvoll benannte Verzeichnisse erstellt, in die jeweils der Inhalt des `poky` Verzeichnisses kopiert werden (`cp -r` erstellt das Zielverzeichnis, falls es nicht existiert):

```
$ cp -r poky non-repro1
$ cp -r poky non-repro2
$ cp -r poky repro1
$ cp -r poky repro2
```

Die Projekte der Verzeichnisse `non-repro1` und `non-repro2` werden ohne weitere Modifikationen gebaut. Die Projekte in den Verzeichnissen `repro1` und `repro2` werden um die zuvor beschriebenen Änderungen zur Erstellung von reproduzierbaren Firmware-Abbildern erweitert. Diese Änderungen können erst nach dem Ausführen des Umgebungsskripts `oe-init-build-env` in die `build/conf/local.conf` Datei geschrieben werden, da vorher das `build` Verzeichnis nicht existiert. Das Umgebungsskript muss dafür in einem Docker Container ausgeführt werden, damit die Pfade für das Projekt korrekt gesetzt werden:

```
$ docker run -ti -v $(readlink -f <repro>):/yocto yocto:latest
yocto@5eda34fed32a:/yocto$ source oe-init-build-env
```

Für das Argument `<repro>` wird jeweils der Pfad zu den Projektverzeichnissen `repro1` und `repro2` eingesetzt. Nachdem das Umgebungsskript ausgeführt wurde, steht das `build` Verzeichnis zur Verfügung und die zusätzlichen Parameter lassen sich in der Konfigurationsdatei `local.conf` eintragen.
Es wird mit jedem Projekt das `core-image-minimal` Betriebssystem gebaut und anschließend das resultierende `tar.bz2` Archiv verglichen. Um das Image mit `bitbake` im Docker Container zu erstellen, wird für jedes Projekt folgende Befehlskette ausgeführt:

```
$ docker run -ti -v $(readlink -f <projekt>):/yocto yocto:latest
yocto@5eda34fed32a:/yocto$ source oe-init-build-env
yocto@5eda34fed32a:/yocto/build$ bitbake core-image-minimal
```

`<projekt>` wird mit dem Pfad zum jeweiligen Projektverzeichnis ersetzt. Ist der Vorgang für jedes Projekt beendet, können die erstellten Archive beispielsweise mit dem Programm `md5sum` verglichen werden. Das Programm erstellt dazu aus dem Inhalt einer Datei einen (möglichst) einzigartigen Wert, mit dem sich der Inhalt der Datei identifizieren lässt:

```
$ md5sum non-repro1/build/tmp/deploy/images/qemux86/core-image-minimal-qemux86.tar.bz2
9c3a124a856d0ba3f03edd8910349eec
$ md5sum non-repro2/build/tmp/deploy/images/qemux86/core-image-minimal-qemux86.tar.bz2
d741ebfd50b2b275e85d25441fac9839
$ md5sum repro1/build/tmp/deploy/images/qemux86/core-image-minimal-qemux86.tar.bz2
3c82c67209a3f30731f5935e5b6d9056
$ md5sum repro2/build/tmp/deploy/images/qemux86/core-image-minimal-qemux86.tar.bz2
3c82c67209a3f30731f5935e5b6d9056
```

Wie man erkennen kann, sind die Hashes der Images aus den Projekten `non-repro1` und `non-repro2` nicht gleich, jedoch sind die der Images aus `repro1` und `repro2` identisch. Die Reproduzierbarkeit von Kompilaten wird somit für das unveränderte Poky Projekt von Yocto soweit gewährleistet. Natürlich muss beachtet werden, dass bei Einführung neuer Layer in das Projekt die Konfiguration unter Umständen nicht ausreicht und spezifisch für betroffene Layer angepasst werden muss.

8 Firmware-Update Management

Das Firmware-Update Management ist in der Welt eingebetteter Systeme ein wichtiges Thema. Die Möglichkeit, nachträglich Fehler zu beheben, Funktionalitäten zu erweitern oder Sicherheitslücken zu schließen, ist eine wichtige Eigenschaft für die Wartung und Langlebigkeit eines Systems. Ausgehend von der Tatsache, dass solche Systeme mehrere Jahre im Einsatz bleiben, ist es wichtig, diese Systeme durch entsprechende Mechanismen zu warten.
Leider gibt es zu den Problemen, welche dieses Thema stellt, bisher keine universell einheitliche Lösung. Oftmals wird zur Lösung der Probleme "das Rad neu erfunden", da häufig ein spezifisches System vorliegt und die Lösung auf dieses angepasst entwickelt wird.
Probleme, welche beim Firmware-Update Management unter anderem zu bedenken sind:

- Unzuverlässige Energieversorgung:
  Der Aktualisierungsvorgang kann zu jeder Zeit unerwartet durch einen Stromausfall unterbrochen werden. Wird beispielsweise ein Schreibzugriff auf den Speicher unterbrochen, kann dies zu Datenverlust oder defekten Daten führen.
- Unzuverlässige Netzwerkanbindung:
  Der Aktualisierungsvorgang kann zu jeder Zeit durch einen Ausfall der Netzwerkanbindung unterbrochen werden.
- Unerreichbarkeit, Unzugänglichkeit:
  Das Gerät ist physikalisch in seiner Betriebsumgebung unerreichbar oder nur schwer zugänglich.
- Erwartete Langlebigkeit des Systems:
  Der Nutzer erwartet einen langen Lebenszyklus des Systems, besonders wenn dieses in einer schwer bis nicht zu erreichenden Betriebsumgebung eingesetzt wird.
- Schutz vor unautorisierter Software:
  Das System muss gegen die Installation von unautorisierter Software geschützt sein. Dies wird unter anderem durch die Prüfung der Authentizität und der Integrität der Aktualisierungsdaten sichergestellt.

Bei der Implementierung eines zuverlässigen Firmware-Update Management Systems müssen einige Punkte beachtet werden. Diese sind unter anderem:

- "Atomic Updates":
  Der Aktualisierungsvorgang wird zuverlässig und vollständig ausgeführt. Wird der Vorgang unterbrochen, ist das Zielsystem in der Lage, dies zu erkennen und setzt den Prozess bei Möglichkeit wieder fort oder wartet auf eine Aktion durch den Nutzer. Diese Eigenschaft versichert, dass bei einem abgeschlossenen Aktualisierungsvorgang die Daten fehlerfrei auf den Speicher übertragen wurden und das System damit einsatzbereit ist.
- Einheitliche Umgebung:
  Wird eine Firmware entwickelt und getestet, sollte die Testumgebung so gut wie möglich die eigentliche Betriebsumgebung nachbilden. So können mögliche Probleme, die durch Umgebungsvariablen auftreten können, frühzeitig erkannt und behoben werden. Ein typisches Szenario ist ein System, welches frei mit verschiedenen Paketen und Versionen von Software ausgestattet werden kann (zum Beispiel durch einen Paketmanager): Nach

einer Aktualisierung würde es nicht mehr erwartungsgemäß laufen, da die Paketversionen der Software eventuell zueinander inkompatibel sind.

- Überprüfen und Schützen der Aktualisierungsdaten:
  Für die Bereitstellung von Aktualisierungsdaten ist es wichtig sicherzustellen, dass diese von einer autorisierten Quelle sind. So soll verhindert werden, dass ein Angriff auf das System durch nicht-autorisierte Dritte über eine Aktualisierung erfolgt. Ein bekanntes Beispiel, bei dem solch ein Angriff unterschätzt wurde und auch heute noch unterschätzt wird, ist das 2016 entwickelte Mirai-Bot-Netz, bei dem durch Sicherheitslücken im Aktualisierungsprozess von eingebetteten, linuxbasierten Systemen, wie Router, Fernseher und generell Haushaltsgeräte, die über eine Internetverbindung verfügen, Schadsoftware aufgespielt wurde. Neben dem standardmäßigen Prüfen der Integrität der Aktualisierungsdaten gibt es weitere Mechanismen, um zusätzlichen Schutz zu gewährleisten. Diese umfassen unter anderem das Verschlüsseln oder Signieren der Daten.
- Integration in den bestehenden Arbeitsprozess:
  Ein wichtiger Aspekt für das Entwickeln von Aktualisierungen für ein System ist die Integration in den bestehenden Arbeitsprozess. Soll eine fertige Lösung für den Aktualisierungsprozess eingesetzt werden, müssen die Werkzeuge zur Erzeugung von Aktualisierungsdaten, in den aktuellen Arbeitsprozess integriert werden.
- Benötigte Bandbreite und Ausfall des Systems durch die Aktualisierung:
  Zur Bereitstellung von Aktualisierungsdaten wird in den meisten Fällen auch darauf geachtet, dass generierte Aktualisierungsdaten möglichst klein sind und damit keine große Bandbreite zur Übertragung auf das Zielsystem benötigen. Zusätzlich soll das System beim Aktualisierungsvorgang möglichst weiterhin arbeitsfähig sein, was unter Umständen wegen der angewandten Strategie zum Aktualisieren der Firmware nicht möglich ist.

Es wird ersichtlich, dass das Firmware-Update Management kein einfach zu behandelndes Thema ist und viele Ansätze zur Lösung der Probleme existieren, aber bisher keine universell einheitliche Lösung entwickelt wurde.

8.1 Strategien des Firmware-Updates Managements

Zur Bereitstellung von Updates für linuxbasierte Systeme haben sich hauptsächlich zwei Methoden etabliert: Image-Updates und Paket-Updates.

Image-Updates sind meist die bevorzugte Methode für eingebettete Systeme. Ein Image (zd. "Abbild") ist dabei ein Abbild einer Partition des Zielsystems und enthält alle vom System benötigten Daten und Programme, um in Betrieb genommen zu werden. Je nach Fall und Situation wird damit keine Grundinstallation benötigt; das Abbild lässt sich direkt in den Speicher des Systems schreiben und ist damit sofort einsatzbereit.
Diese Bereitstellungsmethode hat den großen Vorteil, dass nach jeder Aktualisierung ein bekannter Stand der installierten Daten und Programme hergestellt wird, d. h. die Versionen der installierten Softwarepakete lassen sich anhand der Firmware-Version bestimmen. So ist die Wartung der Firmware deutlich einfacher, denn Probleme wie inkompatible Versionen von Softwarepaketen lassen sich schon im Entwicklungsprozess vorbeugen, und Ursachen für fehlerhaftes Verhalten des Systems im Betrieb lassen sich schneller bestimmen. Zusätzlich erfüllt das Bereitstellen eines kompletten Abbilds der Firmware eine Voraussetzung für zuverlässige Updates, da der Aktualisierungsvorgang für gewöhnlich "atomisch"

abläuft: Werden die Aktualisierungsdaten auf den Speicher geschrieben, sind verschiedene Mechanismen dafür zuständig zu erkennen, ob der Vorgang erfolgreich ablief oder währenddessen Fehler aufgetreten sind. Treten Fehler auf, sind "Fallback"-Mechanismen implementiert, die das System bei einem fehlgeschlagenen Aktualisierungsvorgang auf einen lauffähigen Zustand wiederherstellen können.
Ein Nachteil dieser Methode ist jedoch, dass für die Aktualisierung meist große Datenmengen auf das Zielgerät übertragen werden müssen. Dieser Nachteil wird besonders schwerwiegend, wenn die Daten über eine schwache Netzwerkverbindung empfangen werden sollen oder wenn das System nur über begrenzten Speicher zur Pufferung der Aktualisierungsdaten verfügt.

Die andere Bereitstellungsmethode für Firmware-Updates sind Paket-Updates. Diese kommt häufig bei Desktop-Systemen zum Einsatz. Bei dieser Methode ist jedes Programm ein eigenes Paket, das unabhängig von der Firmware-Version gewartet und aktualisiert werden kann. Diese Pakete beschreiben für gewöhnlich Abhängigkeiten von anderen Softwarepaketen, um mögliche Inkompatibilitäten mit installierten Paketen/Paketversionen zu erkennen. Benötigte Pakete/Paketversionen werden nachinstalliert, damit die Software erwartungsgemäß läuft.
Vorteil dieser Bereitstellungsmethode ist, dass das System nur benötigte Pakete beim Aktualisierungsprozess herunterlädt und installiert und damit die zu übertragende Datenmenge verringert und die Aktualisierung schneller abgeschlossen werden kann.
Jedoch muss beachtet werden, dass bei dieser Bereitstellungsmethode das Ermitteln von Fehlerquellen bei einem nicht ordnungsgemäß laufenden System sehr schwierig ist. Erschwerend kommt hinzu, dass die Testumgebung zur Entwicklung von Aktualisierungen nicht einheitlich bestimmt werden kann, da die Zielsysteme einen unbekannten Zustand besitzen können und daher nicht jeder Fall im Aktualisierungsprozess abgedeckt werden kann.

Der Speicher eines linuxbasierten Systems ist in der Regel in verschiedene Partitionen aufgeteilt. Darunter sind die Bootloaderpartition, eine Recovery- und Updaterpartition und die "RootFS"- oder Systempartition. Die Bootloaderpartition enthält Programmcode für das erste Initialisieren der Hardware und das anschließende Laden des Kernels. Zusätzlich kann der Startvorgang des Kernels durch "Kernel Commandline Arguments" konfiguriert werden. Die Recovery- und Updaterpartition enthält eine minimalistische Version der Firmware mit der nötigsten Software und Treibern, die den Zugriff auf den Systemspeicher und häufig noch auf einen externen Massenspeicher erlauben. Sie wird, wie der Name vermuten lässt, im Update- und Wiederherstellungsprozess genutzt. Die "RootFS"- oder Systempartition enthält das eigentliche Betriebssystem mitsamt Software und allen Treibern für das Gerät. Diese Partitionen werden je nach Anwendungsfall angelegt. Grundsätzlich haben sich für linuxbasierte Systeme zwei Partitionsschemas durchgesetzt: Das symmetrische und asymmetrische Partitionsschema.

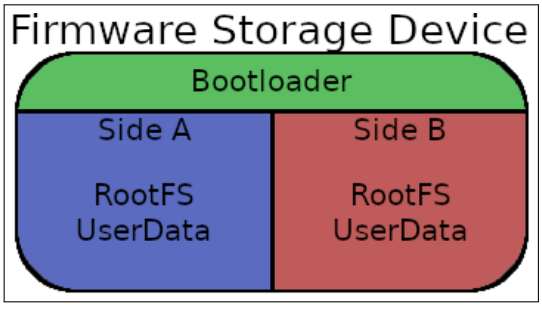

Abb. 10: Symmetrisches Partitionsschema

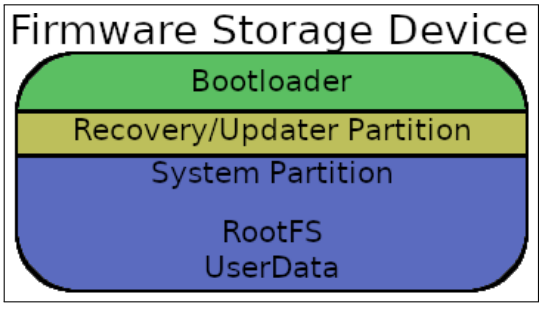

Abb. 11: Asymmetrisches Partitionsschema

Beim symmetrischen Partitionsschema werden eine Bootloaderpartition und zwei Systempartitionen angelegt, welche jeweils eine vollständige Installation der Firmware besitzen. Dabei ist mindestens eine dieser Partitionen als laufende Systempartition im Einsatz, die andere bleibt währenddessen inaktiv. Bei einer Aktualisierung des Systems wird die inaktive Partition mit den Aktualisierungsdaten überschrieben. Bei einem fehlerhaften Installationsvorgang ist das System noch in der Lage, von der zu der Zeit aktiven Partition zu starten. Damit ist immer gewährleistet, dass sich zu jeder Zeit mindestens eine lauffähige Installation der Firmware auf dem System befindet. Es ist jedoch notwendig, den Speicher des eingebetteten Systems für diesen Verwendungszweck zu dimensionieren und damit genug Speicher für alle redundant angelegten Partitionen einzukalkulieren.
Das asymmetrische Partitionsschema besteht aus der Bootloaderpartition, einer Recovery- und Updaterpartition und der Systempartition. Bei einer Aktualisierung muss die laufende Systempartition überschrieben werden. Dafür wird die Firmware der Recovery- und Updaterpartition gestartet, die den Aktualisierungsprozess übernimmt und die Aktualisierungsdaten auf die Systempartition schreibt. Während der Aktualisierung ist das System nicht einsatzfähig. Die Recovery- und Updaterpartition ist auch dafür zuständig die Systempartition bei einem unerwarteten Fehler während der Aktualisierung wiederherzustellen. Dieses Schema wird häufig in Systemen mit stark begrenztem Speicher genutzt. Es gewährleistet durch die Recovery- und Updaterpartition, dass das System zumindest auch nach einem fehlerhaften Aktualisierungsvorgang wiederherstellbar ist.

8.2 Sicherheit für Firmware-Updates

Die Sicherheit des Aktualisierungsprozesses kann anhand der drei Eigenschaften der Informationssicherheit gemessen werden: Vertraulichkeit, Integrität und Verfügbarkeit.
Dass nur befugte Personen Zugriff auf bestimmte Daten haben versteht sich als Vertraulichkeit. Um diese im Rahmen einer Aktualisierung zu etablieren, muss sichergestellt sein, dass nur berechtigte Entwickler Zugriff auf Daten wie private Schlüssel und Passwörter haben. Werden empfindliche Daten mit einer Aktualisierung ausgeliefert, müssen diese durch ein sicheres Verschlüsselungsverfahren für Dritte unzugänglich gemacht werden.

Abb. 12: "CIA Triad": Eigenschaften der Informationssicherheit [7]

Die Integrität der Daten zu prüfen ist die nächste wichtige Eigenschaft. Für den Aktualisierungsprozess werden dazu aus den Aktualisierungsdaten sogenannte "Hash" Werte berechnet und diesen eindeutig zugewiesen. So ist das System in der Lage modifizierte und fehlerhafte Daten zu erkennen.
Die letzte Eigenschaft beschreibt die Verfügbarkeit von Daten. Diese sollen für alle Berechtigten zu jeder Zeit zur Verfügung stehen. Gefahren, wie gezielte Angriffe, natürliche Katastrophen oder menschliches Versagen, müssen möglichst beseitigt werden, um die Verfügbarkeit einwandfrei gewährleisten zu können. Für den Aktualisierungsprozess bedeutet dies, dass Server, die Aktualisierungsdaten bereitstellen, vor Angriffen wie "Denial of Service", kurz"DoS", die eine gezielte Überlastung des Servers verursachen, geschützt werden.

Vertraulichkeit wird durch die Verschlüsselung der Aktualisierungsdaten gewährleistet. Daten, welche als Klartext oder in einem lesbaren Format vorliegen, werden durch die Verschlüsselung in ein unlesbares und scheinbar zufälliges Format übersetzt. Dieses lässt

sich nur mit dem richtigen Schlüssel wieder in die ursprüngliche Form zurück übersetzen. Ein gutes Verschlüsselungsverfahren versichert, dass nur ein Schlüssel in der Lage ist, die Daten wiederherzustellen und dieser auch durch "brute forcing", dem Ermitteln des Schlüssels durch mehrfaches Anwenden von verschiedenen Schlüsseln, nicht ermitteln lässt. Es wird zwischen symmetrischer und die asymmetrischer Verschlüsselung unterschieden.

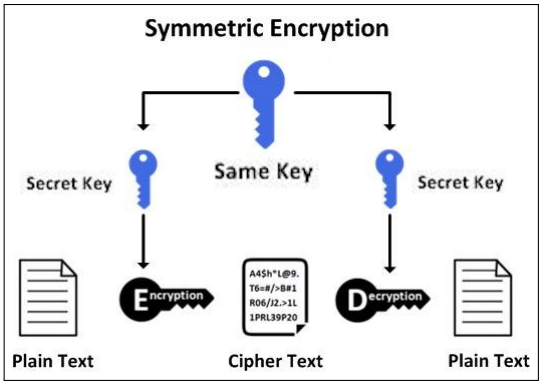

Abb. 13: Symmetrische Verschlüsselung [14]

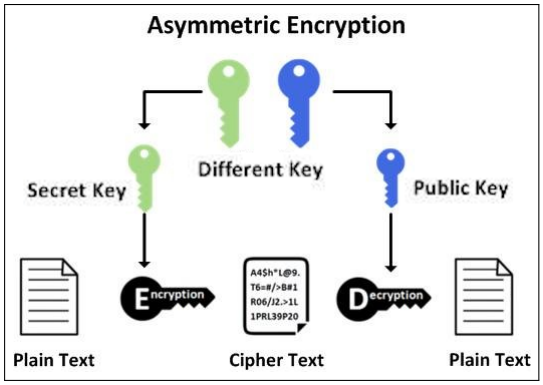

Abb. 14: Asymmetrische Verschlüsselung [15]

Bei der symmetrischen Verschlüsselung wird der gleiche Schlüssel für die Verschlüsselung und Entschlüsselung der Daten genutzt. Aus diesem Grund muss dieser möglichst unzugänglich für Unautorisierte bleiben. Je nach verwendetem Algorithmus steigt die Stärke eines Schlüssels mit dessen Länge. Jedoch steigt damit auch die benötigte Leistung, um Daten zu ver- und entschlüsseln. Mit dieser Methode ist es wichtig, dass der Schlüssel geschützt bleibt. Gelangt dieser trotzdem an Dritte, ist es nicht mehr möglich, ohne großen Aufwand einen neuen, geschützten Schlüssel in das System einzuführen. In solch einem Fall gewährleistet eine Aktualisierung, die neue Schlüssel enthält, nicht, dass diese geschützt bleiben, denn die Aktualisierungsdaten müssen noch mit dem alten Schlüssel verschlüsselt werden, damit das System sie wieder entschlüsseln kann.
Mit der asymmetrischen Verschlüsselung werden zwei verschiedene Schlüssel genutzt, je einer zur Verschlüsselung und einer zur Entschlüsselung. Dabei wird einer der beiden als öffentlicher (ungeschützter) und der andere als privater (geschützter) Schlüssel eingesetzt. Eine mögliche Anwendung ist beispielsweise die Daten mit dem privaten Schlüssel zu verschlüsseln und mit dem öffentlichen zu entschlüsseln. Es ist jedem mit dem öffentlichen Schlüssel möglich, Zugriff auf den Inhalt der Daten zu erhalten. Jedoch ist das Erstellen von Daten, welche vom System akzeptiert werden, nicht ohne den privaten Schlüssel möglich. Damit bleibt das System vor Modifikationen durch Dritte geschützt, denn Aktualisierungsdaten werden nur mit der korrekten Verschlüsselung durch das System installiert. Ein großer Nachteil für dieses Verfahren ist die benötigte Leistung, um Daten zu ver- und entschlüsseln, weshalb es sich meist für stark eingeschränkte Systeme nicht direkt eignet. Häufig wird als Kompromiss zwischen Sicherheit und benötigten Ressourcen eine Kombination von symmetrischer und asymmetrischer Verschlüsselung genutzt. In den meisten Fällen wird nur ein kleiner Teil der Aktualisierungsdaten zur Verifikation der Authentizität mit einem asymmetrischen Schlüsselpaar bearbeitet und der Rest der Daten mit einem symmetrischen.

Die Integrität der Daten wird mit der Nutzung von "Hash" Funktionen geprüft. Diese Funktionen berechnen aus einem beliebigen Datensatz einen Wert mit bekannter Länge. Je nach genutztem Algorithmus wird damit jedem möglichen Datensatz ein eindeutiger und einzigartiger Wert innerhalb der möglichen Ergebnisse der Hash Funktion zugeordnet.

Daher kann dieser Wert auch als ”Fingerabdruck” des Datensatzes bezeichnet werden. Es ist jedoch nicht möglich Kollisionen zu vermeiden, d. h. dass zwei verschiedenen Datensätzen der gleiche Hash Wert zugewiesen wird. Gut konzipierte Hash Funktionen produzieren daher möglichst wenig Kollisionen für beliebige Datensätze.
Wird im Rahmen einer Aktualisierung der Hash Wert der Daten bereitgestellt, können modifizierte und fehlerhafte Daten noch vor der Installation erkannt und gegebenenfalls abgelehnt werden. Eine einfache Hash Funktion zur Prüfung der Integrität ist beispielsweise eine ”Prüfsumme” zu bestimmen. Bei einer 8-Bit Prüfsumme werden alle Bytes einer Aktualisierungsdatei addiert und in ein 8-Bit großes Ergebnis eingetragen, Überläufe werden dabei nicht behandelt. Diese Funktion ist sehr einfach in der Implementierung und benötigt sehr wenig Ressourcen. Jedoch wird klar, dass bei einer 8-Bit Prüfsumme nur 256 mögliche Ergebnisse berechnet werden können und daher eine große Wahrscheinlichkeit besteht, eine Kollision bei der Berechnung zu erhalten. Komplexere Hash Funktionen, wie MD5 und SHA-256, vermeiden solche Probleme durch längere Hash Werte und bessere Algorithmen zur Berechnung.

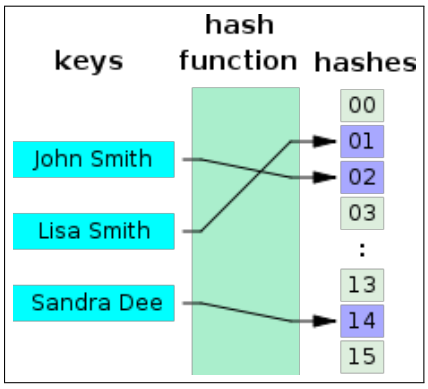

Abb. 15: Datensätze, welchen durch eine Hash Funktion je einem eindeutigen Wert zugewiesen werden [9]

Verfügbarkeit von Daten ist in der heutigen Zeit der Vernetzung tatsächlich schwierig zu gewährleisten. Mit steigender Verfügbarkeit des Internets sind auch die Zahl und die Komplexität der Angriffe auf die Verfügbarkeit gestiegen. Häufig genutzte Angriffe sind ”Denial of Service” (”DoS”) Angriffe, bei denen Server gezielt durch eine zu große Anzahl von Anfragen durch Klienten überfordert werden. Damit ist der Server nicht mehr in der Lage seine Dienste zur Verfügung zu stellen. Einen solchen Angriff zu verhindern ist in den meisten Fällen nicht möglich, da der Angriff selbst in einer Art ausgeführt wird, in der es nahezu unmöglich ist Anfragen zur gezielten Überlastung von normalen Anfragen zu unterscheiden. Als Gegenmaßnahme können die IP-Adressen der Endgeräte festgehalten werden und jeder unberechtigte Zugriff damit erkannt und verweigert werden. Es ist auch sinnvoll eine Authentifizierung zur Kommunikation mit dem Server einzurichten, um so den weiteren Zugriff auf die Dienste des Servers einzuschränken.

8.3 Anforderungen und verfügbare Software

Die in dieser Arbeit zu untersuchende Software muss gewisse Anforderungen erfüllen. Diese umfassen folgende Funktionen und Eigenschaften:

- Effiziente Übertragung von Aktualisierungsdaten
- Prüfung der Authentizität von Aktualisierungsdaten
- Unterstützung verschiedener Partitionsschemas auf dem Zielgerät
- Einfache Integration in die genutzte Entwicklungsumgebung, Yocto
- Möglichkeit bieten, verschiedene Abschnitte des Aktualisierungsprozesses zu ändern oder zu erweitern, ohne den Quellcode modifizieren zu müssen (beispielsweise durch ein Skript die Aktualisierungsdaten über eine im Projekt nicht vorgesehene Übertragungsschnittstelle herunterzuladen)

Die Open Source Gemeinschaft bietet verschiedene Lösungen für das Firmware-Update Management an. Im Rahmen dieser Arbeit wurde RAUC als Software zur Erstellung, Verwaltung und Installation von Aktualisierungsdaten gewählt. Das Kapitel 9 *"Firmware-Updates mit RAUC, barebox und casync"* beschreibt die dafür genutzten Projekte und deren Integration in die Yocto Entwicklungsumgebung.
Im Folgenden werden Alternativen zu RAUC vorgestellt, die zum Zeitpunkt der Untersuchung einige, jedoch nicht alle der gestellten Anforderungen erfüllt haben.

8.3.1 mender

Abb. 16: Logo des mender Projekts [13]

mender, ein von Northern.tech AS entwickeltes Projekt, bietet eine Komplettlösung für den Aktualisierungsprozess von linuxbasierten Systemen. Es stellt einen vollständig implementierten Client für das Endgerät und einen Server zur Verwaltung und Bereitstellung von Aktualisierungsdaten bereit. Zur Aktualisierung werden komplette Firmware-Abbilder übertragen und installiert. Die Serverseite ist zusätzlich in der Lage registrierte Zielgeräte zu verwalten, Updates auf diesen in Gang zu setzen und den Status zu überwachen.
mender bietet als Komplettlösung für das Firmware-Update Management eine sehr einfache Möglichkeit, Aktualisierungen auf einem linuxbasierten System durchzuführen und zu verwalten. Jedoch schränkt es durch die spezifische Implementierung die Flexibilität stark ein, da mender nur gewisse Bootloader und Partitionsschemas unterstützt, die unter Umständen nicht auf dem Zielsystem genutzt werden können. Zusätzlich ist das Fehlen von Mechanismen zur Verringerung von Datenmengen für die Verwaltung und Übertragung von Aktualisierungsdaten problematisch.

8.3.2 swupdate

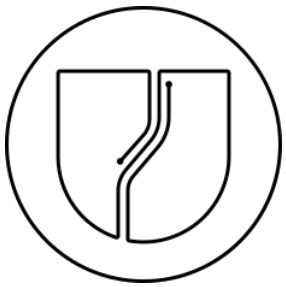
Abb. 17: Logo des swupdate Projekts [21]

Das von Stefano Babic entwickelte Projekt swupdate stellt ein Gerüst zur Erstellung und Installation von Aktualisierungsdaten bereit. Das Ziel des Projekts ist unter anderem ein einheitliches Verfahren für die Aktualisierung von mehreren Zielgeräten durch eine einzige Aktualisierungsdatei bereitzustellen. Dafür werden die verschiedenen Aktualisierungsdaten durch swupdate in ein spezielles Archiv zusammengetragen. Für die Installation werden aus dem Archiv dann nur die benötigten Daten bezogen. So existiert für die gleiche Firmware-Version, die für verschiedene Geräte bereitgestellt wird, nur eine Datei auf dem Server. Es ist auch möglich die Installation von Firmware-Daten mit eigenen Skripten für das Zielgerät zu erweitern oder komplett zu übersteuern. So lässt sich der Aktualisierungsprozess auf das Zielgerät zuschneiden und ist nicht an eine vom Projekt vorgegebene Konfiguration gebunden.
swupdate zeichnet sich durch seine flexible Konfiguration des Aktualisierungsprozesses aus. Es erlaubt neben der Aktualisierung des linuxbasierten Systems auch die Aktualisierung von proprietärer Peripherie des Zielgeräts durch das Einbinden von Skripten, welche bei der Durchführung von einer Aktualisierung ausgeführt werden. Es fehlen aber auch bei diesem Projekt Funktionalitäten zur Verringerung der zu übertragenden Datenmenge für eine Aktualisierung.

8.3.3 libostree

libostree hat seinen Ursprung in der Entwicklung des GNOME Projekts. Das Projekt beinhaltet sowohl Werkzeuge zur Erstellung von Aktualisierungsdaten, als auch eine Softwarebibliothek, die als Grundlage für einen durch den Nutzer implementierten Updateclient genutzt werden kann. Das Projekt nutzt zur Verwaltung und Bereitstellung von Aktualisierungsdaten eine ähnliche Datenverarbeitung wie "git"; es wird ein Verzeichnis genutzt, das sämtliche Firmware-Versionen mit den dazugehörenden Daten enthält. Soll ein neuer Firmware-Stand abgelegt werden, wird dieser mit den bekannten Firmware-Ständen im Verzeichnis verglichen und nur neue oder geänderte Daten angelegt. Zu jeder Firmware wird eine Ordnerhierarchie mit "hardlinks" (eine Datei-Referenz auf die eigentliche Datei) erstellt. So wird die benötigte Datenmenge für ähnliche Firmware-Stände verringert, da gleiche Dateien, die über mehrere Firmware-Stände vorkommen, nur einmalig existieren. Die Software arbeitet zudem auf Datei-Ebene und ist so beim Aktualisierungsprozess unabhängig vom genutzten Dateisystem und der darunterliegenden Hardware.
libostree stellt gegenüber den anderen Lösungen ein speichereffizientes Verfahren zur Ablage und Übertragung von Aktualisierungsdaten bereit. Es ist jedoch möglich, Aktualisierungsdaten in einer noch effizienteren Art und Weise zu erstellen und zu übertragen (differenzielle Updates, weiter beschrieben im Kapitel 9.3 *"casync - content-addressable synchronization tool"*). Zudem fehlen Mechanismen zur Verschlüsselung und Prüfung der Authentizität von Aktualisierungsdaten.

9 Firmware-Updates mit RAUC, barebox und casync

Dieses Kapitel behandelt die zur Verwaltung und Installation von Firmware-Updates gewählte Kombination der Projekte RAUC, barebox und casync.

9.1 RAUC - Robust Auto-Update Controller

Abb. 18: Logo des RAUC Projekts [18]

RAUC, abkürzend für "Robust Auto-Update Controller", ist ein von Pengutronix entwickelter Aktualisierungsclient. Der Client sorgt für die zuverlässige Installation von Aktualisierungsdaten auf linuxbasierten Systemen und stellt zusätzlich Funktionen zur Erstellung und Verschlüsselung von Aktualisierungsdaten mit der genutzten Entwicklungsumgebung bereit. Die Software soll eine einheitliche Schnittstelle zur Erstellung und Installation von Updates bilden. Verschiedene Sicherheitsmechanismen gewährleisten Schutz gegen die Installation fehlerhafter oder unautorisierter Daten auf dem Zielgerät.
Funktionen von RAUC umfassen unter anderen:

- Failsafe und Atomisch:
  Der Aktualisierungsprozess kann bei korrekter Konfiguration unerwartet unterbrochen werden, ohne Gefahr zu laufen, dass das Zielgerät nicht mehr betriebsbereit ist. Ist die Installation der Aktualisierungsdaten am Laufen, werden diese vollständig installiert. Dafür prüft der Prozess vor der Installation die Daten auf die Installationsfähigkeit und setzt nur dann fort, wenn die Daten als installierbar ermittelt werden.
- Signieren und Verifizieren der Aktualisierungsartefakte:
  Bei der Erstellung von Aktualisierungsartefakten wird ein durch OpenSSL erstellter Schlüssel und das passende Zertifikat vorausgesetzt. Diese werden zum Signieren und Prüfen der Aktualisierungsartefakte genutzt. Bei der Installation wird das Zertifikat auf dem Zielgerät benötigt, um die Aktualisierungsdaten zu prüfen.
- Flexible Konfigurationsmöglichkeiten:
  Der Aktualisierungsprozess lässt sich auf das Zielgerät abstimmen. Es werden frei konfigurierbare Partitionsschemas und gängige Dateisysteme, wie ext4, vfat, ubifs und squashfs, unterstützt. Eine erweiterbare Schnittstelle für die Bootloaderinteraktion erlaubt, neben den im Projekt unterstützten Bootloadern grub, barebox und u-boot, einen eigenen Bootloader zu nutzen. Zusätzlich werden Skripte zur weiteren Anpassung und Übersteuerung des Aktualisierungsprozesses unterstützt. Skripte können auch mit den Aktualisierungsdaten ausgeliefert werden, um den Aktualisierungsvorgang basierend auf der Version einer Aktualisierung anzupassen (beispielsweise für die Migration von Systemparametern von einer Firmwareversion auf eine Andere).
- Das D-Bus Interface erlaubt anderer Software mit RAUC zu interagieren (wie zB. einem Wrapper, der RAUC als Backend nutzt).
- Unterstützung für die Build Systeme Yocto und PTXdist.

Die mit RAUC erstellten Aktualisierungsartefakte werden als "Bundles" bezeichnet. Sie enthalten neben den zu installierenden Daten auch Metadaten, die zur Bestimmung des Zielgeräts und dem Pfad der Installation genutzt werden. Optionale Skripte und Dateien zur Anpassung des Aktualisierungsprozesses können auch in ihnen abgelegt werden. Das Installationsziel wird als "Slot" bezeichnet und kann unter anderem ein Gerät, eine Partition oder eine Datei sein. Eine Konfigurationsdatei parametriert den Aktualisierungsprozess

für das Zielsystem. Diese Konfiguration enthält unter anderem die Systembezeichnung, den Pfad zum Zertifikat zur Überprüfung der Aktualisierungsdaten und die Zuweisung der genutzten Slots. Mit diesen Daten prüft RAUC, ob sich Aktualisierungsdaten auf das Zielsystem installieren lassen und wählt den korrekten Slot für die Installation aus. Ist die Installation der Aktualisierungsdaten abgeschlossen, konfiguriert RAUC den genutzten Bootloader, um beim nächsten Hochfahren des Systems die Partition mit den aktualisierten Daten zu nutzen. Zusätzlich werden Funktionen bereitgestellt, um festzustellen, ob das Betriebssystem aus der genutzten Partition erfolgreich gestartet werden kann. Ist es nicht in der Lage von der gewählten Partition zu starten, kann der Bootloader entscheiden, was in diesem Fall gemacht wird (beispielsweise die vor der Aktualisierung genutzte Partition starten).
RAUC unterstützt die Nutzung von casync zur Erstellung und Installation von Aktualisierungsdaten. Dadurch kann der zu übertragende Datensatz für eine Aktualisierung verringert werden.

9.2 barebox Bootloader

Abb. 19: Logo des barebox Projekts [17]

Neben dem Aktualisierungsclient RAUC, stellt Pengutronix auch den Bootloader barebox zur Verfügung. barebox wird für verschiedene Prozessorarchitekturen, wie x86, ARM, MIPS und PowerPC, entwicklet und eignet sich besonders für eingebettete Systeme. Mit dem Ziel möglichst flexibel zu sein, stellt der Bootloader eine Linux-ähnliche Umgebung bereit. Eine Shell und eine große Zahl an Werkzeugen erlauben, den Bootloader für das Zielsystem anzupassen. barebox eignet sich auch als Werkzeug für das ”Board Bring-Up”, die initiale Konfiguration und Einrichtung eines Geräts. Die Unterstützung verschiedener Hardwareschnittstellen erlaubt eine voll automatisierte Konfiguration eines Systems, beispielsweise über die USB- oder Ethernet-Schnittstelle.
Mit der Möglichkeit, es als normale Applikation für ein Betriebssystem zu kompilieren, hebt sich barebox von anderen Bootloadern ab. Diese ”sandbox” erlaubt das Testen von Erweiterungen und Modifikationen von barebox direkt auf der Entwicklermaschine. Dadurch wird auch die Fehleranalyse stark vereinfacht, da alle für die Entwicklung von Applikationen zur Verfügung stehenden Werkzeuge, wie Debugger und Profiler, genutzt werden können.
barebox implementiert auch die sogenannte ”bootchooser” Funktion. Diese Funktion dient zur Wahl des Bootmediums und eignet sich beispielsweise dazu, auf einem System, welches zwei Betriebssystempartitionen nutzt, die durch ein Update aktualisierte Partition zu starten. Sie kann auch dazu genutzt werden, bei einem unerwarteten Abbruch des Startvorgangs der aktiven Partition, beispielsweise wegen einer fehlerhaften Installation, eine alternative Quelle als Bootmedium zu wählen.

9.3 casync - content-addressable synchronization tool

Häufig wird zur Aktualisierung eines Systems das komplette Abbild des Betriebssystems übertragen und anschließend installiert. Dies Verursacht unter Umständen eine große Menge an unnötig übertragenen Daten, denn in den meisten Fällen ist die effektive Datenmenge der Aktualisierung gering.
Das von Lennart Poettering entwickelte Datensynchronisierungswerkzeug casync soll dieses Problem lösen. Ziel des Projekts ist ein effizientes Verfahren für die Übertragung von Abbildern eines Dateisystems bereitzustellen. Dafür bedient es sich an der Tatsache, dass die Dateien eines Betriebssystems größtenteils gleich bleiben und für gewöhnlich nur ein kleiner Datensatz durch eine Aktualisierung geändert wird. So muss zur Aktualisierung des Systems grundsätzlich nur die Differenz an Daten, welche im Vergleich des aktuellen Standes zum Neuen bestehen, übertragen werden. casync stellt zusätzlich nützliche Funktionen zum Arbeiten mit den erstellten Abbilddaten bereit, wie z.B. dem Einbinden dieser in das Dateisystem aus einer lokalen Quelle oder über eine Netzwerkverbindung und das Auflisten des Inhalts inklusive der linux- oder unix-spezifischen Metadaten.
Wendet man casync auf ein Abbild eines Betriebssystems an, wird dieses in "Chunks" (zd. "Brocken") aufgeteilt und in einem "Chunk Store", ein einfaches Verzeichnis, abgelegt. Passend dazu wird noch eine Indexdatei erstellt, die verschiedene Metadaten des verarbeiteten Dateisystems enthält und zur korrekten Wiederherstellung des Abbildes dient. Bei der Verarbeitung des Abbildes in Chunks wird durch "Deduplizierung" die Größe des Chunk Stores minimiert, d. h. Chunks mit gleichem Inhalt werden im Chunk Store nur einmal abgelegt. Die Größe eines Chunks richtet sich nach einem Merkmal des Datensatzes, welches durch den Algorithmus zur Erstellung der Chunks bestimmt wird. Sie bleibt jedoch im Rahmen der durch den Nutzer vorgegebenen Richtwerte für die minimale, durchschnittliche und maximale Größe. Der Algorithmus ist so gewählt, dass bei mehrmaligem Ausführen von casync mit dem gleichen Abbild wieder der gleiche Chunk Store entsteht. Diese Eigenschaft garantiert, dass durch den Prozess bei ähnlichen Abbildern für gleich bleibende Datensätze auch gleich bleibende Chunks erstellt werden. Ist ein Chunk erstellt, wird diesem als Namen der eigene Hashwert, welcher mit der SHA-256 Hashfunktion bestimmt wird, zugewiesen. Anschließend wird dieser im Chunk Store in komprimierter Form abgelegt.

Der Algorithmus zur Erstellung der Chunks basiert auf der "buzhash" Funktion. Diese zählt zu den "Rolling Hash" Funktionen, welche zur Berechnung des Hashwertes ein Fenster, das eine Anzahl von Bytes umfasst, nutzt. Die Rolling Hash Funktion wird unter anderem dazu genutzt, größere Datensätze in kleineren Stücken zu verarbeiten. Chunks werden anhand eines festgelegten Merkmals bei der Berechnung des Hashwertes des eingehenden Datensatzes bestimmt. So lassen sich in ähnlichen Datensätzen auch gleiche Chunks finden, die an unterschiedlichen Stellen der verarbeiteten Daten auftreten. Wie genau die Berechnung des Hashwertes und die Bestimmung einer Chunkgrenze festgelegt wird, ist abhängig von der Implementierung der Rolling Hash Funktion. Die Berechnung der Hashwerte benötigt in der Regel sehr wenig Ressourcen, daher eignet sich die Nutzung auch für stark eingeschränkte Systeme. Wird zudem eine minimale Chunkgröße festgelegt, ist die Verarbeitung der Daten deutlich performanter, da nicht mehr jedes Byte zur Bestimmung einer Chunkgrenze verarbeitet werden muss, sondern nur noch Bytes ab einer Chunkgrenze, die nach der minimalen Chunkgröße vorkommen.
Zur Veranschaulichung des Rolling Hash Algorithmus wird folgende Abbildung betrachtet:

data (ascii)	r	o	l	l	i	n	g		h	a	s	h		a	l	g	o	r	i	t	h	m		
data (value)	114	111	108	108	105	110	103	32	104	97	115	104	32	97	108	103	111	114	105	116	104	109	0	0
sum window (sw)	114	225	333	108	213	110	213	32	136	233	316	316	251	233	237	103	214	328	330	335	325	109	0	0
hash = sw % 8	2	1	5	4	5	6	5	0	0	1	4	4	3	1	5	7	6	0	2	7	5	5	0	0
sum chunk			333		213		213								689						653	109		
position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

Abb. 20: Datenverabeitung mit dem Rolling Hash Algorithmus

Zeilen 1 und 2 der Tabelle zeigen den zu verarbeitenden Datensatz, in diesem Beispiel eine Zeichenkette, und den zugewiesenen Wert des jeweiligen Zeichens nach dem ASCII Standard. Die dritte Zeile ist die Summe der sich im Fenster befindlichen Bytes. Zeile 4 enthält den Hashwert dieser Bytes. Zur einfachen Erkennung eines Chunks wird dessen Prüfsumme (einfache Addition der Bytes ab der letzten Chunkgrenze) in Zeile 5 und die Position des jeweiligen Zeichens des Datensatzes in Zeile 6 angegeben. Chunkgrenzen werden durch einen senkrecht verlaufenden Strich dargestellt.
Der Algorithmus zur Berechnung des Hashwertes für dieses Beispiel ergibt sich folgendermaßen: Es wird eine Fenstergröße von 3 Bytes angenommen. Dieses Fenster iteriert über den eingehenden Datensatz und nimmt die umfassenden Werte auf. Sind Bytes, die durch das Fenster umfasst werden, außerhalb des gültigen Bereichs der Daten oder innerhalb des letzten Chunks, werden diese durch den Wert 0 ersetzt. Anschließend wird der Modulus der Summe der Bytes des Fensters und dem Wert 8 berechnet. Ist dieser gleich dem Wert 5, wurde eine Chunkgrenze gefunden. Eine beispielhafte Umsetzung des beschriebenen Rolling Hash Algorithmus in der Programmiersprache C kann folgendermaßen aussehen:

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

// Ermittelt, ob sich eine Chunkgrenze an Position 'pos' des Datensatzes 'string' befindet
// string:      zu verarbeitende Daten
// len:         Laenge von 'string'
// pos:         Position des Fensters in 'string'
// last:        letzte Position eines gefundenen Chunks
// Rueckgabe:   'true' falls eine Chunkgrenze gefunden wurde, sonst 'false'
bool rolling_hash(char* string, int len, int pos, int last) {
    // Das Fenster umfasst das aktuelle und die letzten 2 Bytes
    int window[3] = {0, 0, 0}; pos = pos - 2;
    for(int i = 0; i < 3; i++) {
        // Byte ueberspringen falls ausserhalb von 'string' oder innerhalb des letzten Chunks
        if(((pos + i) > last) && ((pos + i) < len))
            window[i] = (unsigned int)string[pos + i];
    }
    return (((window[0] + window[1] + window[2]) % 8) == 5);
}

int main(int argc, char** argv) {
    if(argc != 2) { printf("Nutzung: %s \"string\"\n", argv[0]); return -1; }
    char* string = argv[1]; int sum = 0, last = -1;
    for(int i = 0; i < strlen(string); i++) {
        sum += (unsigned int)string[i];
        if(rolling_hash(string, strlen(string), i, last)) {
            printf("Chunkgrenze an Position: %4i, Chunk Pruefsumme = %4i\n", i + 1, sum);
            sum=0; last=i;
        }
    }
    return 0;
}
```

Die durch den gewählten Algorithmus verarbeitete Zeichenkette ”rolling hash algorithm” besitzt 6 Chunksgrenzen, jeweils an den Positionen 3, 5, 7, 15, 21 und 22.
Welchen Einfluss das Ändern von Zeichen des eingehenden Datensatzes auf die Chunkgrenzen hat, wird in Abbildung 21 dargestellt. Hier wurden die ersten zwei Zeichen von ”ro” zu ”ca” geändert.

data (ascii)	r	o	l	l	i	n	g		h	a	s	h		a	l	g	o	r	i	t	h	m		
data (value)	114	111	108	108	105	110	103	32	104	97	115	104	32	97	108	103	111	114	105	116	104	109	0	0
sum window (sw)	114	225	333	108	213	110	213	32	136	233	316	316	251	233	237	103	214	328	330	335	325	109	0	0
hash = sw % 8	2	1	5	4	5	6	5	0	0	1	4	4	3	1	5	7	6	0	2	7	5	5	0	0
sum chunk			333		213		213								689						653	109		
position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

data (ascii)	c	a	l	l	i	n	g		h	a	s	h		a	l	g	o	r	i	t	h	m		
data (value)	99	97	108	108	105	110	103	32	104	97	115	104	32	97	108	103	111	114	105	116	104	109	0	0
sum window (sw)	99	196	304	313	321	323	318	245	104	201	316	316	251	233	237	103	214	328	330	335	325	109	0	0
hash = sw % 8	3	4	0	1	1	3	6	5	0	1	4	4	3	1	5	7	6	0	2	7	5	5	0	0
sum chunk								762							657						653	109		
position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

Abb. 21: Ergebnis des Rolling Hash Algorithmus mit einem geänderten Datensatz

Es wird ersichtlich, dass die Änderung des eingehenden Datensatzes auch eine Änderung der ermittelten Chunks zur Folge hat. Im Vergleich zum Ergebnis der Datenverarbeitung aus Abbildung 20 verschiebt sich die Grenze des ersten Chunks und beeinflusst damit auch den folgenden Chunk. Jedoch sind die letzten drei Chunkgrenzen an den Positionen 15, 21 und 22 gleich geblieben und dem entsprechend auch die darin eingegrenzten Chunks. Der Vorteil des Bestimmens von Chunks mittels Rolling Hash Funktion gegenüber dem Aufteilen eines Datensatzes in gleich große Chunks zeigt sich beim Einfügen (oder Entfernen) von Daten des eingehenden Datensatzes. Dem Datensatz aus Abbildung 20 werden Zeichen vorangestellt, zuerst das Zeichen ”t” und folgend noch das Zeichen ”s”, und in Abbildung 22 das Ergebnis dargestellt:

data (ascii)	r	o	l	l	i	n	g		h	a	s	h		a	l	g	o	r	i	t	h	m		
data (value)	114	111	108	108	105	110	103	32	104	97	115	104	32	97	108	103	111	114	105	116	104	109	0	0
sum window (sw)	114	225	333	108	213	110	213	32	136	233	316	316	251	233	237	103	214	328	330	335	325	109	0	0
hash = sw % 8	2	1	5	4	5	6	5	0	0	1	4	4	3	1	5	7	6	0	2	7	5	5	0	0
sum chunk			333		213		213								689						653	109		
position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

data (ascii)	t	r	o	l	l	i	n	g		h	a	s	h		a	l	g	o	r	i	t	h	m	
data (value)	116	114	111	108	108	105	110	103	32	104	97	115	104	32	97	108	103	111	114	105	116	104	109	0
sum window (sw)	116	230	341	108	216	321	323	318	245	104	201	316	316	251	233	237	103	214	328	330	335	325	109	0
hash = sw % 8	4	6	5	4	0	1	3	6	5	0	1	4	4	3	1	5	7	6	0	2	7	5	5	0
sum chunk			341						566							657						653	109	
position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

data (ascii)	s	t	r	o	l	l	i	n	g		h	a	s	h		a	l	g	o	r	i	t	h	m
data (value)	115	116	114	111	108	108	105	110	103	32	104	97	115	104	32	97	108	103	111	114	105	116	104	109
sum window (sw)	115	231	345	341	108	216	321	323	318	245	104	201	316	316	251	233	237	103	214	328	330	335	325	109
hash = sw % 8	3	7	1	5	4	0	1	3	6	5	0	1	4	4	3	1	5	7	6	0	2	7	5	5
sum chunk				456						566							657						653	109
position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

Abb. 22: Ergebnis des Rolling Hash Algorithmus bei Voranstellen von Zeichen vor den Datensatz

Auch hier ändern sich die ermittelten Chunks gegenüber dem Datensatz aus Abbildung 20. Jedoch kann man erkennen, dass die letzten beiden Chunks mit denen aus Abbildung 20 übereinstimmen. Lediglich die Position der Grenzen sind nicht mehr gleich; sie haben sich um die Länge des eingefügten Datensatzes verschoben.

Es lässt sich feststellen, dass es möglich ist, mit wenig Aufwand durch die Nutzung einer Rolling Hash Funktion Daten effizient und intelligent zu deduplizieren. Die Umsetzung in casync erlaubt auf diese Weise, den benötigten Datensatz für eine Aktualisierung zu verringern. Im Kapitel 10.2 *”Durchführung einer Aktualisierung mit einem casync Bundle”* wird unter anderem getestet, wie groß die Differenz der Datenmenge zwischen Aktualisierungen bei Nutzung von casync ist.

9.4 Integration von RAUC, barebox und casync in das Yocto Projekt

In diesem Kapitel wird die Integration von RAUC, barebox und casync in das Yocto Betriebssystemprojekt, das in Kapitel 5.1 *"Installation der Yocto Entwicklungsumgebung"* angelegt wurde, behandelt. Weitere Informationen und ausführliche Beschreibungen zu den genutzten Projekten finden sich jeweils im Nutzerhandbuch:

- RAUC: `https://rauc.readthedocs.io/en/latest/index.html`
- barebox: `https://www.barebox.org/doc/latest/index.html`
- casync: `https://github.com/systemd/casync/blob/master/doc/casync.rst`

Das Zielsystem, für das RAUC in Kombination mit barebox und casync genutzt werden soll, nutzt die x86-64 Prozessorarchitektur und besitzt zwei Speichermedien, welche jeweils mit dem ext4 Dateisystem formatiert sind und eine Installation des genutzten Betriebssystems enthalten. Es nutzt zudem beim Hochfahren das "EFI" ("Extensible Firmware Interface"). Das System wird mittels QEMU emuliert. Alle Erweiterungen des Betriebssystemprojekts werden bei Möglichkeit im Layer `meta-thesis` angelegt.
Mit dem Kommandozeilenprogramm der Wahl wird in das Stammverzeichnis des Betriebssystemprojekts navigiert und eine neue Distributionskonfiguration und ein Recipe für ein neues Image erstellt. Zunächst wird die Distributionskonfiguration im Layer `meta-thesis` angelegt. Dazu wird das Verzeichnis `meta-thesis/conf/distro` und die darin enthaltene Distributionskonfigurationsdatei `thesis.conf` erstellt:

```
$ mkdir -p meta-thesis/conf/distro
$ touch meta-thesis/conf/distro/thesis.conf
```

In der Konfigurationsdatei sind Parameter zur Konfiguration der Linux Distribution hinterlegt. Die `thesis` Distribution soll die Konfiguration der `poky` Distribution als Grundlage nutzen und `systemd` als Init System einrichten. Zusätzlich wird als Name der Distribution `'Thesis Poky'` gewählt:

```
require conf/distro/poky.conf

DISTRO_NAME = "Thesis Poky"

DISTRO_FEATURES_append = " systemd"
DISTRO_FEATURES_BACKFILL_CONSIDERED += "sysvinit"
VIRTUAL-RUNTIME_init_manager = "systemd"
VIRTUAL-RUNTIME_initscripts = ""
```

Damit `bitbake` die neue Distributionskonfiguration nutzt, muss in der globalen Konfiguration `build/conf/local.conf` die Variable `DISTRO` mit dem Namen der neuen Distributionskonfiguration überschrieben werden:

```
...
# The distribution setting controls which policy settings are used as defaults.
# The default value is fine for general Yocto project use, at least initially.
# Ultimately when creating custom policy, people will likely end up subclassing
# these defaults.
#
DISTRO ?= "poky"
DISTRO ?= "thesis"
...
```

Jetzt wird das Verzeichnis `recipes-core/images` erstellt und das Recipe `image-thesis.bb` darin angelegt:

```
$ mkdir -p meta-thesis/recipes-core/images
$ touch meta-thesis/recipes-core/images/image-thesis.bb
```

Das Recipe enthält zu Beginn nur die Anweisung, dass es auf dem `core-image-minimal` Recipe basiert und der Kernel auf dem Image installiert wird:

```
require recipes-core/images/core-image-minimal.bb

IMAGE_INSTALL_append = " kernel-image"
```

Nun werden die Layer `meta-openembedded`, `meta-ptx` und `meta-rauc` mittels `git` heruntergeladen. Es wird jeweils der `warrior` Branch passend zur Version der Yocto Entwicklungsumgebung gewählt:

```
$ git clone git://git.openembedded.org/meta-openembedded -b warrior
$ git clone https://github.com/pengutronix/meta-ptx -b warrior
$ git clone https://gtihub.com/rauc/meta-rauc -b warrior
```

`meta-openembedded` ist eine Sammlung von Layern des OpenEmbedded Projekts, mit dem die Yocto Entwicklungsumgebung kompatibel ist. Der Layer enthält verschiedene Recipes zum Installieren von Werkzeugen auf dem Linux Betriebssystem:

```
meta-openembedded
├── contrib
├── COPYING.MIT
├── meta-filesystems
├── meta-gnome
├── meta-initramfs
├── meta-multimedia
├── meta-networking
├── meta-oe
├── meta-perl
├── meta-python
├── meta-webserver
├── meta-xfce
└── README
```

RAUC ist abhängig von den Layern `meta-filesystems`, `meta-oe` und `meta-python`.
Der Layer `meta-ptx` enthält Recipes für die Einrichtung und Installation von barebox:

```
meta-ptx
├── classes
│   ├── bootspec.bbclass
│   └── genimage.bbclass
├── conf
│   ├── distro
│   │   └── ptx.conf
│   └── layer.conf
├── COPYING.MIT
├── README
├── recipes-bsp
│   └── barebox
│       ├── barebox_2019.05.0.bb
│       └── barebox.inc
...
```

Der `meta-rauc` Layer enthält neben den für RAUC nötigen Recipes auch welche für die Installation von casync:

```
meta-rauc
├── classes
├── conf
├── COPYING.MIT
├── README
├── recipes-bsp
├── recipes-casync
├── recipes-core
├── recipes-devtools
├── recipes-kernel
├── recipes-support
└── scripts
```

Als Nächstes werden die neuen Layer im Projekt bekannt gemacht. Dafür werden in der Konfigurationsdatei `build/conf/bblayers.conf` folgende Einträge für die Variable `BBLAYERS` hinzugefügt:

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
    /yocto/meta \
    /yocto/meta-poky \
    /yocto/meta-yocto-bsp \
    /yocto/meta-thesis \
    /yocto/meta-openembedded/meta-oe \
    /yocto/meta-openembedded/meta-python \
    /yocto/meta-openembedded/meta-filesystems \
    /yocto/meta-ptx \
    /yocto/meta-rauc \
    "
```

Nun können die Recipes und Klassen der neuen Layer durch andere Recipes genutzt werden. Um RAUC und casync auf dem neuen Image zu installieren, wird die Variable `IMAGE_INSTALL_append` im `image-thesis.bb` Recipe um die Pakete `rauc`, `casync` und `e2fsprogs` erweitert (Variablen mit der Endung `_append` werden konkateniert):

```
require recipes-core/images/core-image-minimal.bb

IMAGE_INSTALL_append = " kernel-image"
IMAGE_INSTALL_append = " rauc casync e2fsprogs"
```

Das Paket `e2fsprogs` enthält Software zur Erstellung und Konfiguration der von Linux genutzten `ext2`, `ext3` und `ext4` Dateisystemen. RAUC benötigt die Werkzeuge zur Formatierung der Zielpartition bei einer Aktualisierung.
Jetzt kann die Konfiguration für RAUC angelegt werden. Diese wird im Layer `meta-thesis` unter dem Pfad `recipes-core/rauc` angelegt. Darin wird das Recipe `rauc_%.bbappend` und das Verzeichnis `files` erstellt:

```
$ mkdir -p meta-thesis/recipes-core/rauc
$ mkdir -p meta-thesis/recipes-core/rauc/files
$ touch meta-thesis/recipes-core/rauc/rauc_%.bbappend
```

Im `files` Verzeichnis werden die Konfigurationsdatei `system.conf` und der Schlüssel mit dem passenden Zertifikat angelegt. Um den Schlüssel und das Zertifikat im `files` Verzeichnis zu erstellen, wird folgender Befehl ausgeführt:

```
$ openssl req -x509 -newkey rsa:4096 -nodes \
                    -keyout meta-thesis/recipes-core/rauc/files/key.pem \
                    -out meta-thesis/recipes-core/rauc/files/cert.pem \
                    -subj "/O=rauc Inc./CN=rauc-demo"
```

Nach erfolgreichem Ausführen des Befehls befinden sich die Dateien `key.pem` und `cert.pem`, die jeweils den Schlüssel und das Zertifikat enthalten, im `files` Verzeichnis. Die Konfigurationsdatei `system.conf` wird genutzt, um das Zielsystem für RAUC zu beschreiben. Mit dieser Beschreibung kann RAUC entscheiden, wie es bei einer Aktualisierung des Systems vorzugehen hat. Die Konfiguration ist in verschiedene Sektionen unterteilt. Die Parameter jeder Sektion werden im Format `<parameter>=<wert>` angegeben. Im Folgenden werden häufig genutzte Sektionen und deren Parameter erläutert (Dateipfade sind aus Sicht des Zielsystems anzugeben):

`[system]` Sektion: Grundlegende Informationen zum System	
`compatible`:	Der Name des Zielsystems. Dieser String wird als rudimentärer Schutz gegen die Installation einer inkompatiblen Aktualisierung verwendet.
`variant-file`:	Der Pfad zu einer Datei mit dem Namen der Systemvariante. Dieser String dient zur Angabe einer gewissen Ausprägung des Systems. Bundles können dann mit Images verschiedener Ausprägungen ausgestattet werden, die jeweils nur auf der entsprechenden Ausprägung des Zielsystems installiert werden.
`bootloader`:	Gibt die zu nutzende Bootloaderschnittstelle an.
`max-bundle-download-size`:	Setzt die maximale Downloadgröße für Bundles, die bei der Installation über eine Netzwerkschnittstelle heruntergeladen werden.

`[keyring]` Sektion: Konfiguration der Authentizitätsprüfung	
`path`:	Der Pfad zum Zertifikat, mit dem die Authentizität von Aktualisierungsdaten geprüft werden soll.

`[handlers]` Sektion: Auszuführende Skripte vor und nach der Aktualisierung	
`pre-install`:	Gibt den Pfad zu einem ausführbaren Skript an, das durch RAUC vor dem Aktualisierungsprozess aufgerufen wird.
`post-install`:	Gibt den Pfad zu einem ausführbaren Skript an, das durch RAUC nach dem Aktualisierungsprozess aufgerufen wird.

`[slot.<class>.<index>]` Sektion: Beschreibung der Speichermedien des Systems. Jeder `slot` gibt dabei ein Ziel für die Aktualisierungdaten an. `<class>` ist eine beliebige Zeichenkette aus Buchstaben und Zahlen und `<index>` ist entsprechend eine Zahl. Die Zeichenkette `<class>` wird auch in der Manifest Datei genutzt, um ein Image aus einem Bundle dem richtigen Slot zuzuweisen	
`device`:	Gibt den Pfad zu einer Partition oder einem Speichermedium an. Dieser zeigt (typischerweise) auf eine Datei im Verzeichnis `/dev`.
`type`:	Gibt das auf der Partition oder dem Speichermedium genutzte Dateisystem an.
`bootname`:	Gibt den im Bootloader genutzten Namen zur Auswahl des Bootmediums an.

Für das zu Beginn beschriebene System kann die `system.conf` Konfigurationsdatei folgendermaßen aussehen:

```
[system]
compatible=qemux86-64
bootloader=barebox
max-bundle-download-size=134217728

[keyring]
path=/etc/rauc/cert.pem

[slot.rootfs.0]
device=/dev/vdb
type=ext4

[slot.rootfs.1]
device=/dev/vdc
type=ext4
```

Die durchschnittliche Dateigröße des Images beträgt 100 Megabyte, der Parameter `max-bundle-download-size` wird daher auf 134 Megabyte (2^{27} Byte) gesetzt. Die Konfiguration der `device` Parameter der jeweiligen Slots wird durch QEMU bestimmt. Dabei ist die Angabe der Reihenfolge des jeweiligen Laufwerks beim Ausführen von QEMU wichtig. Die Zuweisung der Laufwerke nutzt das Muster `'/dev/vdX'`, wobei `X` der Laufwerkindex, beginnend mit dem Index `'a'`, ist. Das erste Laufwerk `'/dev/vda'` wird durch die EFI Firmwarepartition belegt.
Das Recipe `rauc_%.bbappend` wird anschließend so parametriert, dass die Dateien aus dem `files` Verzeichnis auf dem Zielsystem unter `/etc/rauc` installiert werden:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
SRC_URI_append := " \
    file://system.conf \
    file://cert.pem \
"
FILES_${PN} += " \
    /etc/rauc/system.conf \
    /etc/rauc/cert.pem \
"
RAUC_KEYRING_FILE = "cert.pem"
do_install_append() {
    install -d ${D}${sysconfdir}/rauc/
    install -m 0666 ${WORKDIR}/system.conf ${D}${sysconfdir}/rauc/
    install -m 0666 ${WORKDIR}/cert.pem ${D}${sysconfdir}/rauc/
}
```

Der Schlüssel `key.pem` dient ausschließlich zum Verschlüsseln der Aktualisierungsdaten und wird daher nicht auf dem Zielsystem installiert. Die Variable `RAUC_KEYRING_FILE` gibt den Namen der Zertifikatsdatei an.
Um ein Bundle aus dem neu konfigurierten Image zu generieren, wird ein entsprechendes Recipe erstellt. Dieses wird im Verzeichnis `recipes-core/bundles` als `bundle-thesis.bb` angelegt:

```
$ mkdir -p meta-thesis/recipes-core/bundles
$ touch meta-thesis/recipes-core/bundles/bundle-thesis.bb
```

Durch die Konfiguration des Bundle Recipes wird eine Manifest Datei `manifest.raucm` erstellt, die in das Bundle geschrieben wird. Diese Manifest Datei beschreibt den Inhalt des RAUC Bundles und legt unter anderem fest, auf welchen Zielsystemen dieses Bundle installiert werden kann. Die `manifest.raucm` Datei nutzt die gleiche Struktur wie die `system.conf` Konfiguration. Die wichtigsten Sektionen und deren Parameter sind folgende:

`[update]` Sektion: Allgemeine Informationen über das Bundle	
`compatible`:	Name des Zielsystems. Dieser String wird mit dem in der `system.conf` gleichnamigen Parameter verglichen. Stimmen die Zeichenketten überein, wird die Aktualisierung auf dem Zielsystem durchgeführt.
`version`:	Die Version der Aktualisierungsdatei. Diese wird im Aktualisierungsprozess durch RAUC nicht geprüft. Entwickler können bei Bedarf mit diesem Parameter und benutzerdefinierten Skripten eine Versionskontrolle im Aktualisierungsprozess einführen.

`[image.<class>]` Sektion: Beschreibung der im Bundle enthaltenen Images. Jedes Image wird einer Klasse `<class>` zugewiesen. Diese Klasse richtet sich nach den in der `system.conf` Konfiguration gewählten Klassen der konfigurierten Slots. Damit werden die Images auf einem jeweils passenden Slot installiert.	
`filename`:	Dateiname eines im Bundle befindlichen Image. Basierend auf der Dateiendung und dem ZielSlot ermittelt RAUC, wie das Image zu extrahieren und zu installieren ist.

Wird ein Recipe zur Erstellung der `manifest.raucm` Konfiguration genutzt, müssen mindestens die folgenden Variablen im Recipe zugewiesen werden:

- `RAUC_BUNDLE_COMPATIBLE`:
  Die Zeichenkette für den `compatible` String der `[update]` Sektion.
- `RAUC_BUNDLE_SLOTS`:
  Die Zeichenkette für `<class>` der `[image.<class>]` Sektion.
- `RAUC_BUNDLE_SLOT_<class>`:
  Der Name des Image Recipe, welches in das Bundle gepackt werden soll.
- `RAUC_BUNDLE_SLOT_<class>[fstype]`:
  Das genutzte Dateisystem des Image.
- `RAUC_KEY_FILE`:
  Dateipfad zum Schlüssel, mit dem das Bundle verschlüsselt wird.
- `RAUC_CERT_FILE`:
  Dateipfad zum Zertifikat, mit dem das Bundle geprüft wird.

Demnach wird der Inhalt des Recipes `bundle-thesis.bb` folgendermaßen festgelegt:

```
inherit bundle

RAUC_BUNDLE_COMPATIBLE = "qemux86-64"
RAUC_BUNDLE_SLOTS = "rootfs"
RAUC_SLOT_rootfs = "image-thesis"
RAUC_SLOT_rootfs[fstype] = "tar.bz2"

RAUC_KEY_FILE = "${COREBASE}/meta-thesis/recipes-core/rauc/files/key.pem"
RAUC_CERT_FILE = "${COREBASE}/meta-thesis/recipes-core/rauc/files/cert.pem"
```

Das Recipe enthält neben den Zuweisungen der Variablen auch die Anweisung, die `bundle` Klasse zu nutzen. Die Variable `COREBASE` enthält den Pfad zum Stammverzeichnis des Betriebssystemprojekts.
Damit ist die Installation von RAUC und casync auf dem neuen Image `image-thesis` und die Einrichtung eines Bundle für dieses abgeschlossen. Die Verzeichnisstruktur des `meta-thesis` Ordners sollte an diesem Punkt folgendermaßen aussehen:

```
meta-thesis
├── conf
│   ├── distro
│   │   └── thesis.conf
│   └── layer.conf
└── recipes-core
    ├── bundles
    │   └── bundle-thesis.bb
    ├── images
    │   └── image-thesis.bb
    └── rauc
        ├── files
        │   ├── cert.pem
        │   ├── key.pem
        │   └── system.conf
        └── rauc_%.bbappend
```

Mit folgendem Aufruf von `bitbake` wird das Bundle erstellt:

```
$ bitbake bundle-thesis
```

Bei der Erstellung des Bundles wird das Image `image-thesis` automatisch gebaut, d. h. bei Änderungen am Image wird beim Aufruf von `bitbake` mit dem Bundle Recipe entsprechend das Image auch neu gebaut. Das erstellte Bundle wird unter `build/tmp/deploy/images/qemux86-64` als `bundle-thesis-qemux86-64.raucb` abgelegt.
Als Nächstes wird barebox für die Nutzung auf dem Zielsystem konfiguriert. Dazu wird in der Distributionskonfiguration `thesis.conf` barebox als Bootloader festgelegt:

```
require conf/distro/poky.conf

DISTRO_NAME = "Thesis Poky"

DISTRO_FEATURES_append = " systemd"
DISTRO_FEATURES_BACKFILL_CONSIDERED += "sysvinit"
VIRTUAL-RUNTIME_init_manager = "systemd"
VIRTUAL-RUNTIME_initscripts = ""

PREFERRED_PROVIDER_virtual/bootloader = "barebox"
```

Im Recipe `image-thesis.bb` werden die Pakete `barebox` und `ovmf` als Abhängigkeiten eingetragen:

```
require recipes-core/images/core-image-minimal.bb

IMAGE_INSTALL_append = " kernel-image"
IMAGE_INSTALL_append = " rauc casync e2fsprogs"
DEPENDS_append = " barebox ovmf"
```

Dadurch werden die Pakete beim Bauen des Image auch kompiliert und bereitgestellt. Das `ovmf` Paket enthält eine Open Source Implementation der EFI Firmware, die mit QEMU genutzt werden kann. Dieses wird benötigt, da barebox für die x86-64 Prozessorarchitektur nur als EFI Applikation kompiliert werden kann.
Für die Konfiguration von barebox wird im Layer `meta-thesis` das Verzeichnis `recipes-bsp/barebox` angelegt. Darin wird das Recipe `barebox_%.bbappend` und das Verzeichnis `files` erstellt:

```
$ mkdir -p meta-thesis/recipes-bsp/barebox
$ mkdir -p meta-thesis/recipes-bsp/barebox/files
$ touch meta-thesis/recipes-bsp/barebox/barebox_%.bbappend
```

Das Recipe `barebox_%.bbappend` wird mit folgendem Inhalt beschrieben:

```
COMPATIBLE_MACHINE = "qemux86-64"
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
SRC_URI_append := " \
    file://default_boot \
    file://detect \
    file://rootfs0 \
    file://rootfs1 \
"
do_configure_prepend() {
    cp ${S}/arch/x86/configs/efi_defconfig ${WORKDIR}/defconfig
}
do_configure_append() {
    cp ${WORKDIR}/default_boot ${S}/defaultenv/defaultenv-2-base/init/
    cp ${WORKDIR}/detect ${S}/defaultenv/defaultenv-2-base/boot/
    cp ${WORKDIR}/rootfs0 ${S}/defaultenv/defaultenv-2-base/boot/
    cp ${WORKDIR}/rootfs1 ${S}/defaultenv/defaultenv-2-base/boot/
}
do_deploy_append() {
    dd if=/dev/zero of=${S}/barebox.img bs=4M count=1
    /sbin/mkfs.fat ${S}/barebox.img
    mkdir -p ${S}/barebox-img/EFI/boot
    cp ${S}/barebox.efi ${S}/barebox-img/EFI/boot
    echo 'fs0:EFI\\boot\\barebox.efi' > ${S}/barebox-img/startup.nsh
    echo 'rootfs0' > ${S}/barebox-img/boot-device
    /usr/bin/mcopy -i ${S}/barebox.img -s ${S}/barebox-img/* ::/

    install -d ${DEPLOYDIR}
    install -m 0666 ${S}/barebox.img ${DEPLOYDIR}
}
```

Die Variable `COMPATIBLE_MACHINE` enthält die Bezeichnung des Zielsystems, in diesem Fall `qemux86-64`. Die Skripte `default_boot`, `detect`, `rootfs0` und `rootfs1` aus dem `files` werden dem Recipe mit der Variable `SRC_URI_append` bekannt gemacht. Diese werden für die Wahl des Bootmediums durch barebox genutzt.
Der Schritt `do_configure_prepend` wird vor der Konfiguration des Quellcodes von barebox ausgeführt. Darin wird die mit barebox gelieferte Standardkonfiguration `efi_defconfig` für die x86 Prozessorarchitektur in das Stammverzeichnis des Quellcodes von barebox kopiert, damit diese zur Konfiguration des Quellcodes genutzt wird.
barebox nutzt ein virtuelles Dateisystem, das beim Kompilieren des Quellcodes erstellt wird. Dabei wird die Verzeichnisstruktur mitsamt Dateien des `defaultenv-2-base` Verzeichnisses im barebox Quellcodeverzeichnis in das Kompilat geschrieben. Um die Skripte zur Wahl des Bootmediums in barebox verfügbar zu machen, werden diese im Schritt `do_configure_append`, welcher entsprechend nach dem Konfigurieren des Quellcodes ausgeführt wird, in das passende Verzeichnis kopiert.
Ist barebox kompiliert, muss die vorliegende EFI Applikation in einer FAT formatierten Partition für die EFI Firmware bereitgestellt werden. Dazu wird im Schritt `do_deploy_append`, der nach dem Kompilieren des Quellcodes ausgeführt wird, ein 4 Megabyte großes Image `barebox.img` mittels `dd` erstellt, das anschließend mit dem Programm `mkfs.fat` als FAT16 Dateisystem formatiert wird. Daraufhin wird das Verzeichnis `barebox-img` erstellt, in dem die Daten für das Image abgelegt werden. Die EFI Applikationen wird unter dem Standardpfad für Applikationen, `/EFI/boot`, gespeichert. Zusätzlich wird das Skript `startup.nsh` erstellt, dass zum Start der EFI Firmware ausgeführt wird. In diesem befindet sich die Anweisung zum Starten der barebox Applikation. Die Datei `boot-device` wird von den Skripten zur Wahl des Bootmediums durch barebox genutzt. Das Bootmedium ist zu Beginn `'rootfs0'`. Das Programm `mcopy` kopiert die Ordnerhierarchie und Daten des `barebox-img` Verzeichnisses in die `barebox.img` Datei. Zuletzt wird das erstellte Image im Verzeichnis, das durch die Va-

riable `DEPLOYDIR` angegeben wird, abgelegt. Der Wert der Variable ist in diesem Fall `build/tmp/deploy/images/qemux86-64`.
Die Skripte zur Wahl des Bootmediums werden benötigt, weil die interne Funktion `bootchooser` von barebox auf einem x86 System zum Zeitpunkt der Untersuchung im Rahmen dieser Arbeit nicht funktioniert. Mit den Skripten wird ein sehr rudimentärer Ersatz implementiert, der in Kombination mit einem zusätzlichen Skript für RAUC auf dem Zielsystem die aktualisierte Partition als Bootmedium wählt.
Das virtuelle Dateisystem von barebox steht unter dem Verzeichnis `/env` beim Start von barebox zur Verfügung. Im Verzeichnis `/boot` wird die EFI Partition zur Verfügung gestellt. Die Partition dient in diesem Fall auch als globale Ablage für Konfigurationen der beiden Betriebssystempartitionen. Skripte im `/env/init` Verzeichnis werden beim Initialisieren des Bootloaders ausgeführt. Das `env/init/default_boot` Skript wird entsprechend zum Start von barebox ausgeführt und belegt die Umgebungsvariable `global.boot.default` mit dem Wert `detect`. Diese Umgebungsvariable wird von barebox genutzt, um ein Skript nach dem erfolgreichen Initialisieren aus dem `/env/boot` Verzeichnis zu starten:

```
#!/bin/sh
global.boot.default=detect
```

Das `/env/boot/detect` Skript enthält Anweisungen zum Lesen der `/boot/boot-device` Datei, in der das zu startende Bootmedium steht, und das anschließende Starten des Betriebssystems aus diesem:

```
#!/bin/sh
readf /boot/boot-device device
boot $device
```

Der Befehl `readf` liest eine Zeile aus der Datei `/boot/boot-device` und legt diese in der Variable `device` ab. Dem `boot` Befehl wird anschließend die Variable als Argument übergeben, die den Namen eines Skriptes im Verzeichnis `/env/boot` enthält.
Die Skripte `/env/boot/rootfs0` und `/env/boot/rootfs1` enthalten jeweils Anweisungen, um die jeweilige Betriebssystempartition zu starten:

```
#!/bin/sh
# rootfs0
global.bootm.image=/mnt/disk0/boot/bzImage
global linux.bootargs.base="root=/dev/vdb"
```

```
#!/bin/sh
# rootfs1
global.bootm.image=/mnt/disk1/boot/bzImage
global linux.bootargs.base="root=/dev/vdc"
```

Damit nach einer Aktualisierung mit RAUC das aktualisierte Betriebssystem durch barebox gestartet wird, muss ein Skript für RAUC erstellt werden. Dieses wird im Verzeichnis `meta-thesis/recipes-core/rauc/files` als `set-boot-device` angelegt und enthält folgende Befehle:

```
#! /bin/sh
(mount | grep "/dev/vda") || mount /dev/vda /mnt
NUM=$RAUC_TARGET_SLOTS
eval DEVICE=\$RAUC_SLOT_DEVICE_${NUM}
if [ "$DEVICE" == "/dev/vdb" ]; then DEVICE=rootfs0; fi
if [ "$DEVICE" == "/dev/vdc" ]; then DEVICE=rootfs1; fi
echo "$DEVICE" > /mnt/boot-device
umount /mnt
```

Das Skript ermittelt die durch RAUC aktualisierte Partition und schreibt einen entsprechenden Wert in die Datei `boot-device`, welche sich auf der EFI Partition befindet.

Die EFI Partition wird unter Linux als `/dev/vda` registriert. Die Umgebungsvariablen `RAUC_TARGET_SLOTS` und `RAUC_SLOT_DEVICE_<index>` werden bei der Aktualisierung durch RAUC erstellt und Skripten, die beim Aktualisierungsprozess zusätzlich von RAUC ausgeführt werden, zur Verfügung gestellt. `RAUC_TARGET_SLOTS` enthält alle gültigen Indizes für die Variable `RAUC_SLOT_DEVICE_<index>`. Diese Indizes werden für gültige Ziele einer Aktualisierung angelegt. Für das genutzte System wird nur ein Index erstellt, denn neben der aktiven Betriebssystempartition existiert nur die redundante Partition als Slot für RAUC. `RAUC_SLOT_DEVICE_<index>` enthält den Pfad des aktualisierten Slots. Durch die Konfiguration in der `system.conf` Datei von RAUC kann die Umgebungsvariable nur die Pfade `/dev/vdb` oder `/dev/vdc` als Wert enthalten. Entsprechend dem Wert wird das Bootmedium bestimmt und in der Datei `boot-device` eingetragen.
Damit das Skript durch RAUC nach einer Aktualisierung ausgeführt wird, muss die Sektion `[hooks]` und der dazu gehörende Parameter `post-install` mit dem Pfad zum Skript in der `system.conf` Datei hinzugefügt werden (der Pfad zum Skript ist relativ zum Verzeichnis `/etc/rauc` auf dem Zielsystem):

```
[system]
compatible=qemux86-64
bootloader=barebox

[keyring]
path=/etc/rauc/cert.pem

[handlers]
post-install=set-boot-device

[slot.rootfs.0]
device=/dev/vdb
type=ext4

[slot.rootfs.1]
device=/dev/vdc
type=ext4
```

Im Recipe `rauc_%.bbappend` wird noch die Anweisung zur Installation des neuen Skriptes im Verzeichnis `/etc/rauc` auf dem Zielsystem hinzugefügt:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
SRC_URI_append := " \
    file://system.conf \
    file://cert.pem \
    file://set-boot-device \
"
FILES_${PN} += " \
    /etc/rauc/system.conf \
    /etc/rauc/cert.pem \
    /etc/set-boot-device \
"
RAUC_KEYRING_FILE = "cert.pem"
do_install_append() {
    install -d ${D}${sysconfdir}/rauc/
    install -m 0666 ${WORKDIR}/system.conf ${D}${sysconfdir}/rauc/
    install -m 0666 ${WORKDIR}/cert.pem ${D}${sysconfdir}/rauc/
    install -m 0766 ${WORKDIR}/set-boot-device ${D}${sysconfdir}/rauc/
}
```

Damit ist die Einrichtung von barebox komplettiert. Die Verzeichnisstruktur sieht zum Schluss folgendermaßen aus:

```
meta-thesis
├── conf
│   ├── distro
│   │   └── thesis.conf
│   └── layer.conf
├── recipes-bsp
│   └── barebox
│       ├── barebox_%.bbappend
│       └── files
│           ├── default_boot
│           ├── detect
│           ├── rootfs0
│           └── rootfs1
└── recipes-core
    ├── bundles
    │   └── bundle-thesis.bb
    ├── images
    │   └── image-thesis.bb
    └── rauc
        ├── files
        │   ├── cert.pem
        │   ├── key.pem
        │   ├── set-boot-device
        │   └── system.conf
        └── rauc_%.bbappend
```

Wird `image-thesis` oder das `bundle-thesis` mit `bitbake` gebaut, werden die Kompilate für barebox und ovmf im Verzeichnis `build/tmp/deploy/images/qemux86-64` entsprechend als `barebox.img` und `ovmf.code.qcow2` (Code der EFI Firmware) und `ovmf.vars.qcow2` (persistenter Speicher für Variablen der EFI Firmware) gespeichert.
Soll ein casync kompatibles Bundle erstellt werden, muss dieser Vorgang manuell mit RAUC ausgeführt werden, da es aktuell nicht möglich ist, diesen durch ein Recipe mit `bitbake` zu übernehmen. Zunächst wird RAUC für das Host-System mit `bitbake` kompiliert:

```
$ bitbake rauc-native
```

Das Programm wird im Verzeichnis `build/tmp/deploy/tools` als `rauc` abgelegt. Zusätzlich wird Host-seitig casync benötigt. Dazu wird es über den Paketmanager installiert:

```
$ apt install casync
```

Wird aus dem durch `bitbake` erstellten Bundle `bundle-thesis` ein casync Bundle erstellt, sieht der Aufruf von `rauc` folgendermaßen aus (das Arbeitsverzeichnis ist das Stammverzeichnis des Betriebssystemprojekts):

```
$ ./build/tmp/deploy/tools/rauc convert \
    --key=meta-thesis/recipes-core/rauc/files/key.pem \
    --cert=meta-thesis/recipes-core/rauc/files/cert.pem \
    --keyring=meta-thesis/recipes-core/rauc/files/cert.pem \
    build/tmp/deploy/images/qemux86-64/bundle-thesis-qemux86-64.raucb \
    build/tmp/deploy/images/qemux86-64/ca-bundle-thesis-qemux86-64.raucb
```

Mit dem Argument `'convert'` wird durch RAUC aus dem Bundle `bundle-thesis-qemux86-64.raucb` das casync kompatible Bundle `ca-bundle-thesis-qemux86-64.raucb` im `build/tmp/deploy/images/qemux86-64` Verzeichnis erstellt. Zum casync Bundle wird zusätzlich ein gleichnamiges Verzeichnis mit der Endung `.castr`, in diesem Fall `ca-bundle-thesis-qemux86-64.castr`, erstellt, in dem sich die Aktualisierungsdaten als casync Chunks befinden.
Im Kapitel 10 *"Praktische Anwendung des Firmware-Update Systems"* wird das hier konfigurierte System zum Durchführen von Firmware-Updates getestet.

10 Praktische Anwendung des Firmware-Update Systems

Dieses Kapitel beschreibt praktische Anwendungen des in Kapitel 9.4 *"Integration von RAUC, barebox und casync in das Yocto Projekt"* konfigurierten Systems zur Durchführung von Firmware-Updates. In den Anwendungen wird ein Server auf der Host-Maschine simuliert, der die Aktualisierungsdaten für das Zielsystem bereitstellt. Dafür wird das von der Interpretersprache Python zur Verfügung gestellte Modul `http.server` genutzt (verfügbar ab Python Version 3, alternativ kann das Modul `SimpleHTTPServer` von Python Version 2 genutzt werden). Zur Ausführung des Servers wird folgender Befehl im Verzeichnis der Aktualisierungsdaten (im Verzeichnis des Betriebssystemprojekts unter `build/tmp/deploy/images/qemux86-64`) ausgeführt:

```
$ python -m http.server
```

Der Server ist lokal unter der Adresse `http://0.0.0.0:8000` erreichbar (alternativ `http://localhost:8000`). Auf dem mit QEMU emulierten System ist die Serveradresse `http://10.0.2.2:8000`. Der Verlauf der Anfragen an den Server durch Clienten wird direkt in dem ausführenden Kommandozeilenprogramm angezeigt.
QEMU wird zum Starten des Zielsystems im Verzeichnis des Betriebssystemprojekts unter `build/tmp/deploy/images/qemux86-64` folgendermaßen aufgerufen:

```
$ qemu-system-x86_64 -enable-kvm -m 512 \
    -drive if=pflash,file=ovmf.code.qcow2,readonly=on \
    -drive if=pflash,file=ovmf.vars.qcow2 \
    -drive if=virtio,file=barebox.img,format=raw \
    -drive if=virtio,file=disk1.ext4,format=raw \
    -drive if=virtio,file=disk2.ext4,format=raw
```

Die Option `-enable-kvm` erlaubt QEMU die im Kernel implementierten Funktionen zur Beschleunigung eines emulierten Gast-Systems zu nutzen. Dem System werden mit der Option `-m 512` 512 Megabyte an RAM zugeteilt. Mit den ersten zwei `-drive` Optionen wird die EFI Firmware und der dazu gehörende Variablenspeicher in QEMU verwendet. Die folgenden `-drive` Optionen hängen entsprechend die EFI Partition mit barebox und die beiden Betriebssystempartitionen ein. Den Partitionen werden unter Linux folgende Gerätepfade zugewiesen:

- `barebox.img` → `/dev/vda`
- `disk1.ext4` → `/dev/vdb`
- `disk2.ext4` → `/dev/vdc`

Für die erste Betriebssystempartition wird das Image `image-thesis-qemux86-64.ext4` kopiert (sonst wird mit jedem Aufruf von `bitbake` für das Bundle auch das Image überschrieben). Die zweite Betriebssystempartition `disk2.ext4` muss noch erstellt werden. Dazu werden folgende Befehle im Verzeichnis der Aktualisierungsdaten ausgeführt:

```
$ cp image-thesis-qemux86-64.ext4 disk1.ext4
$ truncate -s128M disk2.ext4
$ mkfs.ext4 disk2.ext4
```

Damit wird eine Kopie des Image `image-thesis-qemux86-64.ext4` als `disk1.ext4` angelegt und anschließend ein 128 Megabyte großes Image `disk2.ext4` erstellt, das mit dem ext4 Dateisystem formatiert ist.
Um den Server vom emulierten System aus zu erreichen, müssen folgende Befehle auf diesem nach dem Startvorgang ausgeführt werden:

```
root@qemux86-64:~ ifconfig <schnittstelle> 10.0.2.3 up
root@qemux86-64:~ route add default gw 10.0.2.2
```

Die Netzwerkschnittstelle `eth0` des emulierten Systems wird beim Startvorgang durch einen Prozess von `systemd` umbenannt. Um die korrekte Schnittstellenbezeichnung für das Argument `<schnittstelle>` des `ifconfig` Befehls zu finden, kann im Systemlog mittels `dmesg` und `grep` nach der `eth0` Schnittstelle gesucht werden:

```
root@qemux86-64:~ dmesg | grep eth0
[    2.034606] e1000 0000:00:03.0 eth0: (PCI:33MHz:32-bit) 52:54:00:12:34:56
[    2.035722] e1000 0000:00:03.0 eth0: Intel(R) PRO/1000 Network Connection
[    3.461948] e1000 0000:00:03.0 enp0s3: renamed from eth0
```

In diesem Fall wurde die `eth0` Schnittstelle zu `enp0s3` umbenannt.

10.1 Durchführung einer Aktualisierung mit einem RAUC Bundle

Um eine Aktualisierung mit einem einfachen RAUC Bundle durchzuführen, wird das Programm `rauc` mit dem Argument `install` zusammen mit dem Pfad zum Bundle aufgerufen. Dieser Pfad kann auf eine lokale oder auf einem (entfernten) Netzwerk verfügbare Datei verweisen.
Zunächst wird für den späteren Vergleich die aktuell laufende Betriebssystempartition ermittelt:

```
root@qemux86-64:~ mount
/dev/vdb on / type ext4 (rw,relatime)
devtmpfs on /dev type devtmpfs (rw,relatime,size=244336k,nr_inodes=61084,mode=755)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
...
```

Der Ausgabe von `mount` ist zu entnehmen, dass die Partition des Geräts `/dev/vdb` als RootFS genutzt wird. Nun wird mit folgendem Aufruf Bundle `bundle-thesis-qemux86-64.raucb` direkt vom Server installiert:

```
root@qemux86-64:~ rauc install http://10.0.2.2:8000/bundle-thesis-qemux86-64.raucb
[...] bundle-thesis-qemux86-64.raucb: installing
[...] bundle-thesis-qemux86-64.raucb:   0% Installing
[...] bundle-thesis-qemux86-64.raucb:   0% Determining slot states
[...] bundle-thesis-qemux86-64.raucb:  20% Determining slot states done.
[...] bundle-thesis-qemux86-64.raucb:  20% Checking bundle
[...] bundle-thesis-qemux86-64.raucb:  20% Verifying signature
[...] bundle-thesis-qemux86-64.raucb:  40% Verifying signature done.
[...] bundle-thesis-qemux86-64.raucb:  40% Checking bundle done.
[...] bundle-thesis-qemux86-64.raucb:  40% Loading manifest file
[...] bundle-thesis-qemux86-64.raucb:  60% Loading manifest file done.
[...] bundle-thesis-qemux86-64.raucb:  60% Determining target install group
[...] bundle-thesis-qemux86-64.raucb:  80% Determining target install group done.
[...] bundle-thesis-qemux86-64.raucb:  80% Updating slots
[...] bundle-thesis-qemux86-64.raucb:  80% Checking slot rootfs.1
[...] bundle-thesis-qemux86-64.raucb:  90% Checking slot rootfs.1 done.
[...] bundle-thesis-qemux86-64.raucb:  90% Copying image to rootfs.1
[...] bundle-thesis-qemux86-64.raucb: 100% Copying image to rootfs.1 done.
[...] bundle-thesis-qemux86-64.raucb: 100% Updating slots done.
[...] bundle-thesis-qemux86-64.raucb: 100% Installing done.
[...] bundle-thesis-qemux86-64.raucb: idle
Installing `http://10.0.2.2:8000/bundle-thesis-qemux86-64.raucb` succeeded
```

Nach erfolgreichem Durchführen der Aktualisierung wird das System mit dem Befehl `reboot` neu gestartet. Das System wird jetzt von der zuvor unbeschriebenen Partition des Geräts `/dev/vdc` gestartet. Dies lässt sich mit Ausführen von `mount` bestätigen:

```
root@qemux86-64:~ mount
/dev/vdc on / type ext4 (rw,relatime)
devtmpfs on /dev type devtmpfs (rw,relatime,size=244336k,nr_inodes=61084,mode=755)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
...
```

10.2 Durchführung einer Aktualisierung mit einem casync Bundle

Für die Aktualisierung mit einem casync Bundle wird dem Image eine Modifikation eingeführt, mit der ermittelt werden kann, welche Datenmengen für das differenzielle Update durch casync übertragen werden müssen. Zunächst wird mit dem unveränderten Stand des Projekts aus dem RAUC Bundle `bundle-thesis-qemux86-64.raucb` das casync Bundle `ca-bundle-1.raucb` und der dazu gehörende Chunk Store `ca-bundle-1.castr` erstellt (das Arbeitsverzeichnis ist das Verzeichnis der Aktualisierungsdaten unter `build/tmp/deploy/images/qemux86-64`):

```
$ ../../tools/rauc convert \
    --key=../../../../../meta-thesis/recipes-core/rauc/files/key.pem \
    --cert=../../../../../meta-thesis/recipes-core/rauc/files/cert.pem \
    --keyring=../../../../../meta-thesis/recipes-core/rauc/files/cert.pem \
    bundle-thesis-qemux86-64.raucb \
    ca-bundle-1.raucb
```

Nun wird ein neues Recipe angelegt, durch das eine 1 Megabyte große Datei mit zufälligem Inhalt auf dem Image installiert wird. Der zufällige Inhalt der Datei vermindert die Kompressionsrate der Chunks, die für die Datei erstellt werden, und stellt damit das Worstcase-Szenario dar. Die Differenz der Datenmenge zwischen den Aktualisierungen sollte erkennbar sein und ungefähr der Größe der Datei entsprechen. Es wird das Verzeichnis `meta-thesis/recipes-core/dummy-file` und der darin enthaltene Ordner `files` und das Recipe `dummy-file.bb` angelegt:

```
$ mkdir -p meta-thesis/recipes-core/dummy-file
$ touch meta-thesis/recipes-core/dummy-file/dummy-file.bb
```

Die Datei `dummy-file` wird im `files` Verzeichnis mit `dd` erstellt. Als Quelle der Daten wird das vom Linux Kernel bereitgestellte Gerät `/dev/urandom` genutzt, dass eine kontinuierliche Folge von zufällig generierten Werten mit jedem Lesen zurückgibt:

```
$ dd if=/dev/urandom of=meta-thesis/recipes-core/dummy-file/files/dummy-file bs=1M count=1
```

Mit folgendem Recipe wird die Datei `dummy-file` auf dem Zielsystem unter `/etc/dummy` abgelegt:

```
SUMMARY = "Creates a dummy file."
LICENSE = "CLOSED"

FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
SRC_URI = "file://dummy-file"
S = "${WORKDIR}"

do_install() {
    install -d ${D}${sysconfdir}/dummy
    install -m 0666 dummy-file ${D}${sysconfdir}/dummy/
}
```

Anschließend wird das neue Paket auf dem Image installiert. Dazu wird in dem Image Recipe `image-thesis.bb` das Paket mit der Variablen `IMAGE_INSTALL_append` hinzugefügt:

```
require recipes-core/images/core-image-minimal.bb

DEPENDS_append = " ovmf barebox"
IMAGE_INSTALL_append = " kernel-image"
IMAGE_INSTALL_append = " rauc casync e2fsprogs"
IMAGE_INSTALL_append = " dummy-file"
```

Das modifizierte Bundle wird mit `bitbake` erstellt und im Anschluss mit `rauc` in das casync Bundle `ca-bundle-2.raucb` konvertiert:

```
$ bitbake bundle-thesis
$  ../../tools/rauc convert \
    --key=../../../../../meta-thesis/recipes-core/rauc/files/key.pem \
    --cert=../../../../../meta-thesis/recipes-core/rauc/files/cert.pem \
    --keyring=../../../../../meta-thesis/recipes-core/rauc/files/cert.pem \
    bundle-thesis-qemux86-64.raucb \
    ca-bundle-2.raucb
```

Die Differenz der erstellten casync Bundles kann mit durch Vergleichen der Chunk Stores ermittelt werden. Im Folgenden werden die Gesamtgrößen der Chunk Stores bestimmt:

```
$ ls -l $(find ca-bundle-1.castr/ -type f) | awk '{size+=$5} END {print size " Bytes"}'
30329484 Bytes
$ ls -l $(find ca-bundle-2.castr/ -type f) | awk '{size+=$5} END {print size " Bytes"}'
31381910 Bytes
```

Die Differenz beträgt demnach 1052426 Bytes. Zum Vergleich wird folgend die exakte Dateigröße der `dummy-file` Datei angegeben:

```
$ ls -l ../../../../../meta-thesis/recipes-core/dummy-file/files/dummy-file | \
    awk '{print $5 " Bytes"}'
1048576 Bytes
```

Die Differenz zwischen den Aktualisierungen ist wie erwartet ungefähr die Dateigröße der hinzugefügten Datei. Die Diskrepanz zwischen der Größe der Datei und der tatsächlichen Differenz kann verschiedene Gründe haben, wie zusätzliche Metadaten die durch die neue Datei im Dateisystem eingeführt wurden. Die für die Aktualisierung benötigten Chunks zur Abdeckung der Differenz zwischen den beiden Firmware-Versionen können folgendermaßen ermittelt werden:

```
$ find $( \
    diff -r ca-bundle-1.castr/ ca-bundle-2.castr/ | \
    grep ca-bundle-2 | \
    awk '{print "ca-bundle-2.castr/" $4}' ) -type f
ca-bundle-2.castr/0337/03376743890a21021eee54f8099d55144a97f54ea165806ca52a81d7cefe7cf1.cacnk
ca-bundle-2.castr/2172/21725598c92d637f1a88a00dc80c620827dea9f733063c3da18cfc851c75326b.cacnk
ca-bundle-2.castr/7861/7861e9c4cd75a173e7e1fce3865462057432a474d241e9bb7b69ff118984e437.cacnk
ca-bundle-2.castr/94cb/94cb801f5b2b3b785ac92315338fd08dbe0cb8b1bc56b03c4633675ea7100a8a.cacnk
ca-bundle-2.castr/9e20/9e2043a0b61d28320d2969e26d50cf6865f4bca210962b93b1cd7af23a7a8925.cacnk
ca-bundle-2.castr/a280/a28070bcc5d74de361c8f7825809100f652bfbc17134bfbd7ca22d07b8abb594.cacnk
ca-bundle-2.castr/ae82/ae826df46018cfbd9a20374d3cdbb42be8989dd0ba0246cbdcd4505a91ed3da8.cacnk
ca-bundle-2.castr/c5aa/c5aa6416dc1ad57f7c4090eaed1a9652b0897992879ed0b767d89b825e1fe9f9.cacnk
ca-bundle-2.castr/cc92/cc929d2a7e57ad73c77e4fb82b70262fbbcea18410fe38e0f548954ff6f0d7a0.cacnk
ca-bundle-2.castr/ccd9/ccd9619af2ef26d3d1a90cd37a73ef64086427b9cc71238c876db792b9977020.cacnk
ca-bundle-2.castr/cf08/cf0849ff95ed7e97660fac8687616a82c84890727239cb09f3d1603437761eda.cacnk
ca-bundle-2.castr/d386/d386d3b8142fc6117b9373c069b7b532aaea1b3318f7d493a91c2b73e0d6b068.cacnk
ca-bundle-2.castr/d781/d781f992778cefaabf865722785dc4b0f4b90dcb8e412d0be62e473745451d24.cacnk
ca-bundle-2.castr/e450/e450d21ac174336c9b2c927fbebf2e6a7adf012d1c5b5edd788e68d29324603c.cacnk
ca-bundle-2.castr/ed5c/ed5c55a42f564633cc49cf1ac3158d3a9fbf58683a8948d0054fbbf89dc64f2e.cacnk
ca-bundle-2.castr/fa74/fa744a8a458b3e940097466f2e5c96d996dcafb0c3e044a110d1bbcde0022bd0.cacnk
```

In diesem Fall werden 16 Chunks benötigt. Entsprechend kann die Gesamtgröße des differenziellen Updates ermittelt werden:

```
$ ls -l $(find $( \
    diff -r ca-bundle-1.castr/ ca-bundle-2.castr/ | \
    grep ca-bundle-2 | \
    awk '{print "ca-bundle-2.castr/" $4}' ) -type f) | \
    awk '{size+=$5} END {print size " Bytes"}'
1121279 Bytes
```

Für dieses Update müssen ca. 1,1 Megabyte heruntergeladen werden.
Die Installation des casync Bundles wird auf die gleiche Art wie die Installation eines normalen RAUC Bundles durchgeführt (beschrieben in Kapitel 10.1 *"Durchführung einer Aktualisierung mit einem RAUC Bundle"*). Man muss beachten, dass RAUC bei der Installation eines casync Bundles annimmt, dass der Pfad zum Chunk Store der gleiche wie der des Bundles ist.

10.3 Unerwartete Unterbrechung des Aktualisierungsprozesses

Die Konfiguration des Zielsystems soll unter anderem sicherstellen, dass durch den unerwarteten Abbruch des Aktualisierungsvorgangs das System trotzdem noch betriebsfähig bleibt. Für den Test der Funktionalität wird während dem Schreiben der Aktualisierungsdaten ein Systemreset durchgeführt (die Option ist im laufenden Fenster von QEMU unter dem Menü `Machine` → `Reset` verfügbar):

```
root@qemux86-64:~ mount
/dev/vdb on / type ext4 (rw,relatime)
devtmpfs on /dev type devtmpfs (rw,relatime,size=244336k,nr_inodes=61084,mode=755)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
[...]
root@qemux86-64:~ rauc install http://10.0.2.2:8000/bundle-thesis-qemux86-64.raucb
[...] bundle-thesis-qemux86-64.raucb: installing
[...] bundle-thesis-qemux86-64.raucb:   0% Installing
[...]
[...] bundle-thesis-qemux86-64.raucb:  80% Checking slot rootfs.1
[...] bundle-thesis-qemux86-64.raucb:  90% Checking slot rootfs.1 done.
[...] bundle-thesis-qemux86-64.raucb:  90% Copying image to rootfs.1

UEFI Interactive Shell v2.2
[...]

barebox 2019.05.0 #1 Wed Feb 12 16:39:03 UTC 2020
[...]
Booting entry 'rootfs0'
ext4 ext40: EXT2 rev 1, inode_size 128, descriptor size 32
mounted /dev/disk0 on /mnt/disk0

Loading MS-DOS executable '/mnt/disk0/boot/bzImage'


Thesis Poky 2.7 qemux86-64 ttyS0

qemux86-64 login: root
root@qemux86-64:~ mount
/dev/vdb on / type ext4 (rw,relatime)
devtmpfs on /dev type devtmpfs (rw,relatime,size=244336k,nr_inodes=61084,mode=755)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
[...]
```

Vor dem Update wird das Betriebssystem aus dem Speichermedium unter `/dev/vdb` ausgeführt. Der Aktualisierungsprozess wird beim Schreiben der Daten durch ein Systemreset unterbrochen. Ist das System nach dem Neustart hochgefahren, wird durch die Ausgabe des `mount` Befehls ersichtlich, dass das vor dem Aktualisierungsprozess genutzte Speichermedium `/dev/vdb` weiterhin als aktive Systempartition genutzt wird. Grund dafür ist, dass die Änderung des Eintrags der aktiven Systempartition erst nach dem erfolgreichen Durchführen der Aktualisierung erfolgt.

11 Fazit

Als Ziel dieser Arbeit wurde die Erstellung von Aktualisierungsdaten durch das Yocto Buildsystem für linuxbasierte, eingebettete Systeme gesetzt. Diese Aktualisierungsdaten sollten in einem möglichst effizienten Format in Hinsicht auf die zu übertragende Datenmenge erstellt werden. Um dies zu erreichen, wurde die Entwicklungsumgebung so konfiguriert, dass Aktualisierungen auf differenzieller Basis mithilfe der Projekte RAUC und casync ausgeführt werden, d. h. dass zur Durchführung einer Aktualisierung nur die Differenz an Daten von einem Firmware-Stand auf den anderen heruntergeladen werden. Um diese Differenz zwischen Firmware-Versionen zu verringern, wurde das Buildsystem für die Reproduzierbarkeit von Kompilaten eingerichtet, um Kompilate von nicht verändertem Quellcode binär kompatibel zur letzten Version zu halten und keine umgebungsabhängigen Änderungen einzuführen. Auch aus sicherheitstechnischer Sicht ist die gewählte Software in der Lage, das Endgerät vor unautorisiertem Zugriff durch kompromittierte Aktualisierungsdaten zu schützen.
Die Kombination der Software RAUC, barebox und casync eignet sich auch auf lange Frist wegen der Erweiterbarkeit zur Verwendung für den Aktualisierungs- und Verwaltungsprozess der Firmware eines Endgeräts. Die Nutzung von Skripten im Bootvorgang und bei der Aktualisierung eines Systems erlauben systemspezifische Implementierungen der Prozesse, ohne den Quellcode der Projekte zu modifizieren.

12 Quellenverzeichnis

[1] Canonical Ltd.
https://assets.ubuntu.com/v1/29985a98-ubuntu-logo32.png.

[2] Docker Inc.
https://docs.docker.com/engine/reference/builder.

[3] Docker Inc.
https://docs.docker.com/install/linux/docker-ce/debian.

[4] Docker Inc.
https://www.docker.com/sites/default/files/d8/2019-07/Moby-logo.png.

[5] Docker Inc.
https://www.slideshare.net/Docker/docker-birthday-3-intro-to-docker-slides. Seite 18.

[6] Docker Inc.
https://www.slideshare.net/Docker/docker-birthday-3-intro-to-docker-slides. Seite 18.

[7] F5 Networks Inc.
https://www.f5.com/content/dam/f5-labs-v2/article/articles/edu/20190709_what_is_the_cia_triad/cia_triad.png.

[8] gmacario.
https://hub.docker.com/r/gmacario/build-yocto.

[9] Joge Stolfi.
https://upload.wikimedia.org/wikipedia/commons/5/58/Hash_table_4_1_1_0_0_1_0_LL.svg.

[10] Larry Ewing.
https://isc.tamu.edu/~lewing/linux/sit3-shine.7.gif.

[11] Linux Foundation.
https://www.yoctoproject.org/wp-content/uploads/2017/10/yocto-logo.png.

[12] Linux Foundation.
https://yoctoproject.org/docs/2.7/brief-yoctoprojectqs/brief-yoctoprojectqs.html.

[13] Northern.tech AS.
https://mender.io.

[14] Ons JALLOULI.
https://www.researchgate.net/profile/Ons_Jallouli/publication/321123382/figure/fig1/AS:561500478808064@1510883554986/Symmetric-encryption-primitive.png.

[15] Ons JALLOULI.
https://www.researchgate.net/profile/Ons_Jallouli/publication/321123382/figure/fig2/AS:561500483002368@1510883555030/Asymmetric-encryption-primitive.png.

[16] opensource.org.
https://opensource.org/files/osi_logo_600X800_90ppi_0.png.

[17] Pengutronix e.K.
https://barebox.org.

[18] Pengutronix e.K.
https://rauc.io.

[19] Raul M. Silva.
https://www.debian.org/logos/openlogo-nd-100.png.

[20] Reproducible Builds.
https://https://reproducible-builds.org.

[21] Stefano Babic.
https://github.com/sbabic/swupdate.

[22] ThayerW.
https://www.archlinux.org/logos/archlinux-icon-crystal-64.svg.

[23] Victor Siame.
https://www.gnu.org/graphics/official%20gnu.svg.