



Embedded Linux

mit Yocto und OpenEmbedded

Von Marco Israel
Dezember 2019

Inhaltsverzeichnis

Inhaltsverzeichnis	iv
Abbildungsverzeichnis	v
Abkürzungsverzeichnis	vii
1 Einleitung	1
1.1 Über Yocto, OpenEmbedded, Poky, Meta-daten und Bitbake	1
1.1.1 Die Yocto Community	2
1.1.2 Die OpenEmbedded Community	2
1.2 Docker	2
1.3 Literatur	3
2 Grundlage	5
2.1 Bitbake	5
2.2 Bitbake Buildprozess	5
2.2.1 Fetch Task	6
2.3 Configurations-Dateien *.conf	6
2.4 Operatoren zum Verändern von Variablen	6
2.5 Yocto Tools	7
2.6 Entwicklungsrollen und Workflows	8

2.7	Recipes, Calsses, Packedgroups, Konfigurationsdateien, meta daten . .	9
3	Setup your host	11
3.1	Erforderliche Pakete	11
3.2	Host Konfiguration	12
4	Setup Eclipse	15
4.1	Installation, Einrichtung und Plugins	15
4.2	Cross Compile und Remote Debugging Einstellungen	15
5	QT5 einrichten	17
5.1	Arbeiten mit Qt Creator	17
6	Hardware und Linux distribution	19
6.1	Hardware (<i>machine</i>) Beschreibung	19
6.2	Linux Distribution definieren	20
6.3	Bootloader definieren	20
7	Kernel Module entwicklung	23
7.1	Kernel Module	23
7.1.1	Neues Kernel Modul	23
7.1.2	Kernel anpassen - features aktivieren	23
7.2	Software Module	23
7.2.1	Eigene Kernel modul erstellen und einbinden	24
8	Applikationsentwicklung	25
8.1	Einbinden von Applikationen in vorhandene Images	25
8.2	Existierende Pakete und Metadaten nutzen	25

8.2.1	Software dem Image hinzufügen	26
9	Hardware flashen	29
10	Commands Bitbake	31
10.1	Bitbake Dokumentationsquellen	31
10.2	Neuer <i>meta-layer</i>	31
10.3	meta-layer zum build hinzufügen	31
10.4	Active meta-layer auflisten	32
10.5	Anzeigen aller recipes - beispielsweise aller images	32
10.6	Anzeigen der Tasks eines recipes	32
10.7	Ausführen bestimmter Tasks eines Recipes	32
10.8	Umgebungsvariablen eines Recipes	33
10.8.1	Häufig benötigte Umgebungsvariablen	33
10.9	section name	33
10.10	Erzeugen von temporären Entwicklungs-Shells	33
11	Yocto devtool	35
11.1	devtool Kommandos	35
11.2	Arbeiten mit <i>devtool</i>	35
12	Bekannte Fehler - Pitfalls	37
12.1	Never do	37
12.2	Must do	37
12.3	Generelle Empfehlungen	38
13	Lösung für bekannte Fehler	41

13.1	ERROR: Fehler beim Bauen eines recipes>	41
13.1.1	Lösungsschritte	41
13.2	Logging in Recipes	41
13.3	Abhängigkeiten (Dependencies) visualisieren	42
14	Ausblick	43
14.1	Nächste Schritte	43
14.1.1	Entwicklungswerkzeuge als Docker Container	43
14.1.2	Erweiterung der Scripte	43
14.2	Security	44
14.3	Lizenzen	44
	Glossar	48
	Literatur	49

Abbildungsverzeichnis

Abkürzungsverzeichnis

SDK Software Development Kit

DTB Device Tree Blob

BSP Board Support Packed

TFTP Trivial File Transfer Protocol

NFS Network File System

FTP File Transport Protocol

WYSIWYG What you see is what you get

1 Einleitung

1.1 Über Yocto, OpenEmbedded, Poky, Meta-daten und Bitbake

Das *Yocto Project* bezeichnet eine Community Gruppe welche sich zur Aufgabe gemacht hat, das Erstellen von Linux Distributionen für eingebettete Systeme zu vereinfachen. Dabei werden Distributionen gezielt für einen Anwendungsfall zusammengestellt und für die jeweilige Zielhardware von Grund auf direkt aus den Quelldateien übersetzt. Letzteres ergibt sich aus der Vielfalt der unterschiedlichen Hardware und CPU bzw. Controller Architekturen.

Zusammen mit der *OpenEmbedded Community* pflegt das Yocto Project eine Software Build Umgebung mit dem Namen *Bitbake*, bestehend aus Python- und Shell-Skripten, welches das Erstellen von Linux Distributionen koordiniert. Parallel zu Bitbake entstehen und wachsen innerhalb der Yocto Community verschiedene Tools wie z.B. Wrapper, welche das Arbeiten mit Bitbake vereinfachen sollen.

Des Weiteren pflegen beiden Communities Metadaten (Metadatei die in Form von Regeln beschreiben, wie Software Pakete für unterschiedliche Distribution und Hardware durch Bitbake gebaut werden müssen. Diese Regeln werden in einer Python und GNU Make ähnlichen, eigenen, Bitbake Skriptsprache geschrieben. Sie werden *Recipes* (Rezepte) genannt.

Regeln bzw. diese Metadaten und somit auch das Build-System Bitbake arbeiten nach einem *Schichten Modell*. Dabei beschreiben Meta-Daten auf unterster Schicht allgemeine grundlegende Anleitungen zum Übersetzen der wichtigsten Funktionen eines Betriebssystems, aufbauend auf Schichten welche zunächst die Hardware oder den Bootloader beschreiben (Board Support Packed (BSP) Schichten). Höhere Schichten erweitern (detaillieren) oder überschreiben grundlegende, tiefere Rezepte. So setzt auf Beschreibungen der Hardware Software- und Anwendungsschicht auf. Ein solches

Schichtenmodell ermöglicht es, einzelne Schichten auszutauschen oder darunterliegende Einstellungen abzuändern. Ziel ist es, durch dieses Schichtenmodell eine eigene Linux Distribution zu erschaffen.

Ein Vereinfachtes Modell ist abgebildet unter [Gon18, S.26]

1.1.1 Die Yocto Community

Die Yocto community stellt Meta-Daten (Recipes) bereit, um ein minimalistisches Betriebssystem mit grundlegenden Linux Tools unter der Virtualisierungsumgebung *QEMU* für verschiedene Architekturen starten zu können.

Das Minimalisten Betriebssystem der Yocto Community wird *Poky* genannt.

Zudem pflegt es zusammen mit der OpenEmbedded community das Buildsystem *bitbake*.

1.1.2 Die OpenEmbedded Community

Die OpenEmbedded Community stellt Meta-Daten Rezepte bereit, die auf dieses minimalistische Betriebssystem aufsetzen und genutzt werden können um ein Betriebssystem nach eigenen Wünschen gestalten (zusammen setzen) und für Hardware Plattformen konfigurieren zu können. Hierzu gehören sowohl Hardware als auch open Source Software Beschreibungen. Zusammen mit der Yocto Community pflegen und erweitern das Build Umgebung *Bitbake*, welches verschiedene Aufgaben in geregelter Reihenfolge im Multicore Betrieb durchführt. Weiteres in Abschnitt 2.1 Seite 5; Bitbake .

1.2 Docker

Bitbake benötigt, neben einem Python Interpreter, verschiedene Tools beispielsweise *GIT* oder *wget*. Diese Tools, wie auch Python, sind in ihrer Version abhängig von der Bitbake Version sowie den Metadaten. Es ist **dringend** erforderlich den Versionsstand dieser Tools „Einzufrieren“.

An dieser Stelle kommt *Docker* ins Spiel. Alle Abhängigkeiten und Versionsstände in einem Docker Container zusammengefasst ermöglicht das Einfrieren bestimmter Versionen bei gleichzeitigem Parallelbetrieb von mehreren Versionen durch unterschiedlicher Container. Ein Container kann unter unschädlichen Betriebssystemen und Plattformen auf gleiche Weise genutzt (ausgeführt) werden, ohne das Neukonfigurationen oder Installationen nötig sind (abgesehen von Docker selbst).

Im Gegensatz zu anderen Virtualisierungstechniken hat Docker trotz Virtualisierungstechniken keinen großartigen Performance Verlust weshalb es vor anderen Virtualisierungen wie VirtualBox vorzuziehen ist.

Von dem Gebrauch klassischer Virtualisierungstechniken ist aus Performancegründen dringend abzuraten. Ein Beispiel ist VirtualBox.

1.3 Literatur

Nachfolgende Quellen sind zum Wissensaufbau oder als Nachschlagewerk zu empfehlen.

Host, IDS und Tool einrichten: [Phy19b], [Yocto_Eclipse_Plugin] und [Gon18]

Flash, HW Schnittstellen und DTB: [Phy19a, *Booting_the_System* und *Updating_the_System*]

Einführung und Beispiele [Gon18]

YouTube Tutorial Serie: [Live Coding with Yocto Project Yoc19a]

Nachschlagewerk: [Rif19]

Linux Treiber Entwicklung: [QK15] und [CRK05]

Linux / Unix SystemCall Referenz: [Ker10] und [RS13]

Phytec Mira Board Downloads: [Phy19c]

2 Grundlage

2.1 Bitbake

Die Buildumgebung *Bitbake*, im Kern bestehend aus Python- und Shellscripen die mit unterschiedlichen Softwareentwicklungswerkzeuge wie GIT, Make oder Autotools interagieren, führt in einer geregelten Reihenfolge verschiedene Aufgaben aus. Die Abbildung auf [Gon18, S. 20] zeigt abstract den Buildprozess:

- Übersicht über die zu Verarbeitenden Konfigurationsdateien
- Reihenfolge in der angepasste Konfigurationsdateien (.conf) verarbeitet werden.
- Build schritte (Siehe Abschnitt 2.2 Seite 5; Bitbake Buildprozess)
- Packetieren in unterschiedliche Formaten
- Generieren und bereitstellen von Software Development Kits
- Aufräum- und Nacharbeiten

2.2 Bitbake Buildprozess

Jedes *recipe* erbt eine Reihe von standard Build-Tasks. Hierzu gehören u.a.

- Herunterladen (anfordern, *fetchen*) und Sammeln von Quell (Source) Dateien (z.B. Git Repositorien, ftp Servern oder lokalen Dateien.)
- Übersetzen, konfigurieren, patchen, installieren, verifizieren usw. von Paketen bei Verwendung von mehreren Prozessor Kernen
- Bereitstellen von Entwicklungs- und Verwaltungswerkzeugen

Eine Auflistung der Standard-Task kann, mitsamt kurzer Beschreibung, entnommen werden [Gon18, S. 171-172]

2.2.1 Fetch Task

Eine der Hauptaufgaben von Bitbake besteht darin, die jeweiligen Softwarepakete in Ihren benötigten Versionen und Revisionsständen aus unterschiedlichen Quellen zu sammeln und dem Buildsystem zur Verfügung zu stellen. Solche Quellen können sein:

- Lokaler Bitbake-Download ordner/cache. Er enthält bereits einmal heruntergeladene Softwarepakete in jeweiligen Revisionsständen.
- Lokaler Pfad zu Quelldateien; z.b. in einem eclipse oder QT Workspace
- Lokale Repository (z.B. lokales GIT Repository)
- Netzwerkpfad zu einem Client oder Server im lokalen Netzwerk. Z.B. über Freigaben oder einem lokalen FTP Server
- Online Repository oder TFTP Server.
- Alternatives lokales oder Remote (online) Repository.

Die Reihenfolge in welcher nach Source-Dateien gesucht werden soll ist zum einen definiert durch:

- Das Recipe selbst welches das Softwarepakete innerhalb Bitbake bauen soll
- durch Konfigurationsdateien wie *./conf/local.conf*
- sowie durch eine allgemein fest vorgegeben Reihenfolge innerhalb bitbakes. Siehe [Gon18, S.53]

2.3 Configurations-Dateien *.conf

Bitbake, sowie die Buildprozesse der einzelnen Recipes werden gesteuert durch unterschiedliche **recipe-lokale** und **globale** Konfigurationsdateien.

2.4 Operatoren zum Verändern von Variablen

Nachfolgende Operatoren stehen in unterschiedlichen Dateien zur Verfügung. Dabei sind sie (teils ausschließlich) in *.conf dateien oder in recipes anwendbar.

Standard operatoren wie +=, ?=, ... Eine Auflistung der standard Operatoren ist zu finden unter: [Gon18, Seite 160]

_append, _prepena, _removed Werden überwiegend in *.conf Dateien verwendet. Genau wie die Standard Operatoren erweitern Sie eine Variable. Sie werden jedoch zu einem späteren Zeitpunkt als die standard Operatoren verarbeitet, was einen erheblichen Einfluss auf den Buildprozess und das Ergebnis haben kann. Weiteres ist zu finden unter: [Gon18, Seite 160]

INHERRIT Genutzt ausschließlich in *.conf Dateien zum Erben (inkludieren) von Konfigurationsklassen.

inherit Genutzt ausschließlich in recipes *.bb und *.bbappend zum Erben (inkludieren) von Bitbake-Klassen. Weiteres unter [Rif19]

include Wenn Datei vorhanden, dann include ihren Inhalt. Andernfalls ignoriere den Befehl. *include* lässt sich sowohl in rKconfigurationsxxid-Dateien als auch recipes verwenden um alle anderen Arten von Dateien zu inkludieren (andere Recipes, Configs, ...) Weiteres unter [Rif19]

require Wie include nur dass die Datei vorhanden sein muss, ansonsten beendet bitbake mit einem Fehler. Weiteres unter [Rif19]

2.5 Yocto Tools

Zum Arbeiten an Bitbake recipes, meta-layern, bitbake classes, bitbake Paketgruppen oder an Konfigurationsdateien stehen verschiedene Werkzeuge bereit.

- *bitbake-layers* help
- *yocto-layers* help
- *recipe-tool* help
- *devtool* -help
- *bitbake -c devshell <recipe>*
- *bitbake -c devpyshell <recipe>*

- GNU Tools wie *grep*, *awk*, *set*, *diff*, *cp*, *vim*, *usw*

Dabei können gleiche Aufgaben mit verschiedenen Werkzeugen oder auch manuell bearbeitet werden können und zum selben Ergebnis führen.

Nicht jedes Tool ist dabei für jeden *Workflow* geeignet. Zum Teil ist es einfacher und weniger Fehleranfälliger einzelne Schritte manuell durchzuführen. Etwa das Integrieren neuer Softwarekomponenten in ein Image.

2.6 Entwicklungsrollen und Workflows

Generell ist das Arbeiten an einer Linux Distribution und das Arbeiten mit Bitbake in verschiedene Aufgabenpakete zu unterteilen:

- Anpassen eines Linux Kernels und Entwicklung neuer Kernel Module
- Anpassen von Hardware-Beschreibungen wie Device Trees und Bootloader
- Anpassen von meta-layern auf u.a. auf BSP und Applikationsschicht. Sowie erstellen von neuen meta-layern, recipes, bitbake-paketen, ...
- Entwickeln von Softwarekomponenten; Hardwarenah; nutzen von Systemcalls
- Entwickeln von Softwarekomponenten; Abstract; nutzen von höheren Funktionsbibliotheken
- Einbinden und patchen / erweitern von Softwarepakete in den Buildprozess,

Je nach Aufgabengebiet unterscheidet sich auch die Arbeitsweise:

- Externe Software- oder Modulentwicklung, etwa in *eclipse* oder *qt-creator* sowie externes Testen der Komponenten. Bitbake kann für diese Aufgabe ein Software Development Kit bereitstellen, welches die Cross- Entwicklung für die Zielplattform ermöglicht. Siehe hierzu ?? Seite ??; ?? .
- Patchen oder erweitern von Softwarekomponenten die Bereits in den Linux Kernel und Bitbake integriert sind, lassen sie am einfachsten durch Tool bereitgestellt, die von der Yocto / OpenEmbedded Community bereit gestellt werden. Hierzu zählt z.B. das s.g. *devtool*. Siehe hierzu Kapitel 11 Seite 35;

Yocto devtool . Aber auch eine temporäre Bitbake-bash oder bitbake-python *development-shell* steht bei einfachen Anpassungen zu verfügung.

- Beim Arbeiten an recepis und meta-layern und somit dem erweitern und pflegen des Buildssystems, können helfen:
- *bitbake-layers* help
- *yocto-layers* help
- *recipe-tool* help
- *devtool* -help

2.7 Recipes, Calsses, Packedgroups, Konfigurationsdateien, meta daten

Im laufe des Arbeitens mit Bitbake tauchen verschiedene Begrifflichkeiten auf. Das wichtigste Vokabular beschreibt einleitend kurz und knapp [Phy19d]

Recipes Die Regeln in einer eigenen Skriptsprache, beschreiben wie ein Softwarepaket zu bauen und in das Gesamtsystem zu Integrieren ist.

Classes Verallgemeinerte, übergeordnete Rezepte die durch Rezepte inkludiert werden können.

Packedgroups Zusammenfassung von Softwarepakete die z.B. in Abhängigkeit zueinander stehen.

Konfigurationsdateien Eine Übersicht über die relevanten Konfigurationsdateien liefert [Gon18, Seite 19, 20]

Meta daten Zusammenfassung von Recipes und Konfigurationsdateien als Softwarepaket, Softwareschicht, Hardware- oder Distributionsbeschreibung.

3 Setup your host

3.1 Erforderliche Pakete

Das *Dockerfile* „Yocto“ zeigt eine Liste aller nötigen Ubuntu Pakete die zum Arbeiten mit der Yocto / OpenEmbedded Build Umgebung auf einem Host benötigt werden. Beim Arbeiten mit dem Docker Image sind alle nötigen Tools bereits nach dem erstellen des Docker Container in dem Container in kompatiblen Versionen enthalten. *Bitbake* und andere Tools können direkt genutzt werden.

Der Container lässt sich bauen mit *docker build -t Dockerfile*.

Zusätzlich sind die nachfolgenden Pakete zum Arbeiten auf dem lokalen Host nötig oder empfehlenswert. Nähere Informationen zu den Paketen, Quellen, Konfigurationen usw. sind auf verschiedenen Internetseiten zu finden.

GIT: Manuels Herunterladen von Metadaten oder anderen GIT Repositories

Docker: Zwingend erforderlich um einen Docker Container bauen und ausführen zu können.

TFTP Server Bereitstellen des fertigen Linux Images, Root File Systems sowie des Device Tree Blob

NFS Server Bereitstellen eines globalen Dateisystems über das Netzwerk

microcom Werkzeug zum verbinden mit einer serielle Schnittstelle

eclipse Entwicklungs IDE mit Unterstützung zum Cross-Kompilieren als auch zum Remote debuggen

QT5 und QT5 Creator (Cross-)Entwicklung und Remote debugging IDE für grafische Oberflächen in C++ als WYSIWYG Editor.

Openssh-Server Verbinden und Übertragen von Dateien mit/zu dem eingebetteten Linux auf der Zielhardware.

3.2 Host Konfiguration

Nachfolgende Pakete benötigen weitere Konfigurationen

Docker:

- Docker Service starten
- Docker-yocto Image bauen: ***docker build -t Dockerfile*** .
- Hilfe liefert docker --help oder die **Internetseite**
- Image starten durch ausführen von „./run.sh bash “

TFTP Server:

- TFTP Austausch Ordner anlegen und Zugriffsrechte definieren
- stat-alone daemon (/etc/default/tftpd-hpa) oder xinitd Service (/etc/xinitd.d/tftp) konfigurieren
- Server neu starten
- **BEISPIEL** im „BSP Manual“ unter phytec.de; Stichwort „Bootig the Kernel from Network“(Booting _ the _ Kernel _ from _ Network) [Phy19a] oder unter [Gon18, Seite 44]

NFS Server:

- NFS Server konfigurieren (/etc/exports)
- NFS Server neu starten
- **BEISPIEL** im „BSP Manual“ unter phytec.de; Stichwort „Bootig the Kernel from Network“(Booting the Kernel from Network) [Phy19a] oder unter [Gon18, Seite 45]

Microcom

- Der Parameter --port Definiert die Serielle Schnittstelle.
- Weiteres ist unter Manual Seite zu finden.

Eclipse • Weiters im Kapitel Kapitel 4 Seite 15; Setup Eclipse

QT5Creator

- Weiters im Kapitel Kapitel 5 Seite 17; QT5 einrichten

Yocto Areitsverzeichnis

- Erstellen eines globalen Arbeitsverzeichnisses. Z.B. /opt/yocto
- Setzen der Rechte **rwX** Rechte für alle user („others“).

Python Version als standart interpreter Yocto/Openembedded Tools bauen auf je nach Version auf Python Version 2 oder 3 auf. Daher ist es nötig einen Symlink oder Alias auf die jeweilige Python Version zu setzen. Z.b. *alias python=python2*

Proxy und Routen Je nach Netzwerkinfrastruktur müssen Proxy und Netzwerkrou-ten auf dem lokalen Host gesetzt werden. Beispielsweise für die Tools:

- GIT
- wget
- apt-get
- https_proxy und http_proxy

Informationen hierzu liefert die das Yocto Manual oder das Yocto Wiki unter dem Stichwort „**Working Working Behind a Network Proxy** “(Working Behind a Network Proxy)

4 Setup Eclipse

4.1 Installation, Einrichtung und Plugins

Zur Entwicklung von C/C++ kann Eclipse CDT verwendet werden.

Zudem sind zum Cross compilieren und Remote Debuggen sowie zum remote Deployen nachfolgende Plugins bzw zusätzliche Eclipse-Softwaremodule nötig: (*Help -> Install new Software*)

- C/C++ Remote (Over TCF/TE) Run/Debug Launcher.
- Remote System Explorer User Actions
- TM Terminal via Remote System Explorer
- TCF Target Explorer

Des weiteren stellt die Yocto download Seite downloads.yoctoproject.org ein SDK Plugin bereit, welches das Konfigurieren von Remote Einstellungen für jeweilige Entwicklungsprojekt vereinfacht und zusammenfasst. Zum installieren muss beispielsweise zu den Installations-Quellen in eclipse hinzu gefügt werden:

<http://downloads.yoctoproject.org/releases/eclipse-plugin/2.6.1/oxygen/>

4.2 Cross Compile und Remote Debugging Einstellungen

Eine Anleitung mit Beispielen zum Cross Compile und Remote Debugging unter eclipse liefern:

- **[Phy19b, Working with Eclipse]** Anleitung wie o.g. ohne Plugin unter Verwendung eines SDKs in eclipse eingerichtet werden kann
- **[Gon18, Seite 249-269]** Anleitung wie das Yocto Plugin für o.g. verwendet werden kann.

- **[var19]** Wie oben, zeigt die Verwendung des Yocto Plugins.

5 QT5 einrichten

Wie sich QT zur Cross-Entwicklung und Remote debugging einrichten und in Kombination mit einem bitbake-SDK nutzen lässt ist beschrieben unter

- [Gon18, Seite 269-276]
- [Phy19b, Working with Qt Creator]

5.1 Arbeiten mit Qt Creator

Der Workflow, wie sie QT-Anwendungen mittels Qt Creator entwickeln und in Bitbake einbinden lassen, ist beschrieben unter: [Gon18, Seite 277-285]

6 Hardware und Linux distribution

Jeder Metadaten-layer aber auch jedes einzelne Recipe lässt sich gezielt Hardware Beschreibungen bzw Hardware Gruppen oder einer (eigenen) *Linux-Distributionen* zuordnen, womit jener Meta-Layer oder jenes Recipe nur für solche Hardware oder Distribution Anwendung findet.

Die `./conf/local.conf` definiert für einen Buildprozess die zu verwendende Hardware Gruppe und Linux Distribution.

6.1 Hardware (*machine*) Beschreibung

Damit ein Linux System und sine Software Komponenten auf einer Ziel Hardware lauffähig sind, muss diese Hardware sowohl für das Linux Betriebssystem, als auch für Bitbake beschrieben werden. Dieses geschieht mittels einer Konfiguration:

```
<hw-metalayer>/conf/machine/<machine>.conf
```

So muss beispielsweise einem Linux Kernel mittels eines Device Tree Blob die Hardware beschrieben bzw. bekannt gemacht werden. Bitbake muss unter anderem wissen über welche s.g. *Features* eine Hardware verfügt. Ein Feature ist beispielsweise die Existenz eines Displays oder einer nicht- standard Pheripheral wie eine PCI-Port oder wifi.

Eine Lister möglicher *Machine Features* ist zu finden unter:

- [Abschn. Machine_Features Yoc19b]
- [Abschn. Machine_Features Rif19]

Zudem definiert die Maschinenkonfiguration

- welche Linux distribution verwendet werden soll bzw. kann. Siehe hierzu Abschnitt 6.2 Seite 20; Linux Distribution definieren

- Welcher Bootloader Anwendung finden soll. Siehe hierzu Abschnitt 6.3 Seite 20; Bootloader definieren
- Welche Softwarepakete, Treiber, Kernelmodule, Konfigurationen, usw. für eine Zielplattform zwingend zum starten erforderlich oder etwa wünschenswert sind.

Die Maschinenkonfiguration ist somit eine Zusammenfassung aller nötigen und optionalen Komponenten, damit eine Software mitsamt Betriebssystem auf einer Zielhardware lauffähig und verwendbar wird.

Sollte es Änderungen oder Erweiterungen an einer bestehenden Hardware-Plattform geben, so müssen unter Umständen auch diese bekannt gemacht werden. Hierzu kann entweder eine bestehende Beschreibung verändert werden, oder eine neue Maschinenbeschreibung erstellt werden. [Gon18, S. 91-95]

6.2 Linux Distribution definieren

Eines der **grundlegenden Ziele** von Bitbake, bzw. Ziel der Yocto und OpenEmbedded Community ist die Möglichkeit eigene Linux Distributionen gezielt für einen Anwendungsfall zu definieren. Dank seines Schichtenmodells ist eine Yocto-Distribution unabhängig von einer Hardware. Eine Distribution wird für einzelne Hardwareplattformen immer neu übersetzt. Hierzu wird auch die Toolchain im Vorfeld aus Quellen vollständig erstellt, sollte diese nicht vorhanden sein. Eine definierte Distribution beinhaltet für jede Zielplattform die selben Tools, Anwendungen, Konfigurationen oder andere konfigurierte Features, insofern die jeweilige Ziel-Hardware diese Features (in Hardware) unterstützt.

6.3 Bootloader definieren

Ein bootloader ist ein Stück Software das die Schnittstelle zwischen Software und Hardware darstellt. Er hat die Aufgabe ein Betriebssystem oder baremetal Software zu starten oder zu aktualisieren. Er bestimmt etwa, wo der Einstiegspunkt einer Software oder eines Betriebssystems zu finden ist (Speicherkomponente, Sektor, ...) und übernimmt die Aufgabe solche Software über eine definierte Schnittstelle zu aktualisieren.

Einem Betriebssystem liefert es u.a. eine Beschreibung der Hardware in Form eines

Device Tree Blob (DTB).

Yocto / OpenEmbedded setzt i.d.R. auf einen der nachfolgenden verfügbaren

- U-Boot
- Barebox (eine Neuauflage des U-Boot Bootloaders)

Ähnlich wie die Hardware und Zielplattform (Siehe Abschnitt 6.1 Seite 19; Hardware (*machine*) Beschreibung sowie eine Betriebssystemdistribution muss auch der Bootloader mittels Configurationsdateien und Recipes in der Bitbake Buildumgebung für eine Zielplattform definiert werden.

Nötige Konfigurationen sind beispielhaft beschrieben unter [Gon18, S. 93–95, 98–99, 100–108] und gehen i.d.R. mit der Maschinendefinition einher.

7 Kernel Module entwicklung

7.1 Kernel Module

7.1.1 Neues Kernel Modul

7.1.2 Kernel anpassen - features aktivieren

Wie unter [Gon18, Seite 109] beschrieben, existieren verschiedene Möglichkeiten ein bereits im Linux Kernel vorhandenes Modul oder Feature zu aktivieren. Z.B.

- manuelles Anpassen der Kernel *.config* Datei
- GUI oder Menü basiert mittels
 - menuconfig
 - xconfig
 - gconfig
- *bitbake -c menuconfig virtual/kernel*

Seite [Gon18, Seite 114 und Seite 118] zeigt letztes beispielhaft und beschreibt wie **Änderungen dauerhaft gespeichert werden können**

7.2 Software Module

Neue, selbst erstellte Kernel Module lassen sich durch die nachfolgenden Konfigurationsvariablen zum Image *rootfs* hinzufügen.

Die Variablen können beispielhaft in einem Image-Recipe oder in einer der Konfigurationsdateien wie *./conf/local.conf* oder *machine/<yourMachine.conf>* definiert werden.

- `MACHINE_ESSENTIAL_EXTRA_RDEPENDS_append += <module-recipe>`

- `MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS_append += <module-recipe>`
- `MACHINE_EXTRA_RDEPENDS_append += <module-recipe>`
- `MACHINE_EXTRA_RRECOMMENDS_append += <module-recipe>`

Um das Modul *automatisch, beim booten des Kernels* zu laden, muss zusätzlich definiert werden:

- `KERNEL_MODULE_AUTOLOAD_append = „< ><kernel-Modul> “`
- `KERNEL_MODULE_AUTOLOAD_append += <kernel-Modul>`
- `KERNEL_MODULE_AUTOLOAD += <kernel-Modul>`

7.2.1 Eigene Kernel modul erstellen und einbinden

Seite [Gon18, S. 121–125] zeigt beispielhaft, wie sich ein eigene Kernelmodule erstellen und in die Buildumgebung Bitbake einbinden lässt. Der *poky* meta-layer beinhaltet zudem ein beispielhaftes *hello-world* Kernel Modul, mitsamt beispielhafter *Makefile* und *recipe.bb* Datei.

Ein umfangreiches Handbuch zur Linux Kernel und Modulentwicklung stellen die beiden Bücher dar:

- [QK15]
- [CRK05]

Nachschlagewerke zum Thema Linux/Unix Kernel Schnittstellen (System-Calls) sind die Bücher:

- [Ker10]
- [RS13]

8 Applikationsentwicklung

8.1 Einbinden von Applikationen in vorhandene Images

Die nachfolgende Schritte sind nötig um ein existierendes Softwarepaket zu einem Image hinzu zu fügen.

8.2 Existierende Pakete und Metadaten nutzen

1. Image Branches herausfinden: `bitbake -e <recipes/image> | grep ^LAYERSERIES_COMPAT`
2. Seite [ope19] aufrufen
3. **Branch** auswählen und ggf. Reiter **recipes** neu auswählen.
4. **Suchen** nach dem gewünschten Softwarepaket in der Suchleiste. Dies liefert als Ergebnis den jeweiligen Metadaten Layer, welcher das Recipes enthält.
5. Verifizieren, ob der Metadaten-layer bereits existiert: Z.b. durch
 - Manuelles Suchen in der lokalen (meta-layer) Verzeichnisstruktur
 - `find -name <new-meta-layer>`
 - `bitbake-layers show-layer | grep «new-meta-layer»`
6. Layer Packet herunterladen in das lokale Yocto Arbeitsverzeichnis, in welchem sich bereits *Poky* und andere OpenEmbedded Metadaten-Layer befinden.:
 - Manuelles Herunterladen und einfügen in `./conf/bblayersi.conf` `BBLAYER += «pfad/zum/meta-data-layer»`
 - `git clone <url-to-meta-layer>` und manuelles einfügen in `bblayersi.conf`
 - `bitbake-layers layerindex-fetch <layer-name>`

7. Beim manuellen Herrunterladen, den neuen Layer zu `./conf/bblayers.conf` hinzufügen. Dieser Schritt wird durch `bitbake-layers layerindex-fetch <metadatalayer-name>` bereits vollzogen.
8. Abhängigkeiten Metadaten-layer herunterladen und ebenfalls dem Projekt hinzufügen. Abhängigkeiten lassen sich ermitteln oder ausgeben durch:
 - *anklicken des Recipie names* unter [ope19] einsehen und von dort, wie oben beschriebe manuell oder tool gestützt herunterladen
 - direkt ausgeben durch: `bitbake-layers layerindex-show-depends <layer-name>` und anschließend herunterladen und hinzufügen durch `bitbake-layers layerindex-fetch <layer-name>`
9. Falls nicht im Vorfeld ausgewählt, den nötigen *Branch auschecken* mit `git checkout <branch>` in jedem neuen Metadaten-layer.
10. Auf Fehler überprüfen (z.B. auf fehlende Abhängigkeiten):
`bitbake-layers show-layers`

8.2.1 Software dem Image hinzufügen

Um ein Softwarepaket einem Image hinzu zu fügen muss die Variable `IMAGE_INSTALL` gesetzt oder erweitert werden. Hier gilt zu entscheiden ob das Hinzufügen *global* für alle Image Recipes oder *image-lokal* für gezielt einzelne Images (sprich Distributionen) geschehen soll:

- Setzen oder erweitern in `./conf/local.conf`
- Abändern des `<image-recipes>.bb` files durch setzen / erweitern der Variablen
- Image erweitern durch erstellen eines `<image-recipes>.bbappend` files
- Image Variable gezielt für ein Image/Target erweitern (s.u.)

Um eine **Variable zu erweitern** bieten sich folgende mögliche Operatoren an :

spätes Anhängen `IMAGE_INSTALL_append = « ><your-sw-packed> »`

spätes Anhängen `IMAGE_INSTALL_prepend += « <your-sw-packed> »`

direktes Anhängen `IMAGE_INSTALL += «>your-sw-packed>»`

direktes Anhängen `IMAGE_INSTALL .= «>your-sw-packed>»`

mehrere Pakete direkt anhängen `IMAGE_INSTALL_append = «>your-sw-packed1> <your-sw-packed2> <...>»`

`IMAGE_INSTALL_prepend += «your-sw-packed1> <your-sw-packed2> <...>»`

`IMAGE_INSTALL_append_<your-image-recipes> = «>your software packed>»`

`IMAGE_INSTALL_<your-image-recipes> += «your software packed>»`

ACHTUNG:

Es sind die Varianten mit Operator ***append*** und ***prepend*** vorzuziehen!

append und **prepend** werden verarbeitet **nachdem** alle Recipes von Bitbake geparsed und erstmalig verarbeitet wurden. Die Operatoren `+=` und `-=` werden direkt beim Parsen eines Recipes verarbeitet. Aufgrund des unterschiedlichen (früheren) Bearbeitungszeitpunktes kann bei den Operatoren `+=` und `.=` nicht garantiert werden, zu welchem Zeitpunkt und in welcher Reihenfolge eine Variable erweitert wird. **Dieses kann in einigen Fällen zu Fehlern führen.** Siehe hierzu auch [Gon18].

Generelle Unterscheidung:

_append **_prepend** für `.conf` Dateien. Diese Dateien werden vor recipes verarbeitet. Durch diese Operatoren wird dennoch erreicht, dass die Konfiguration abschließend Anwendung findet

operatoren `+=` `.=` `..` Sind für recipes

ACHTUNG ACHTUNG: beachte das `< >`, welches ein **führendes Leerzeichen** symbolisiert und bei dieser Art der Variablenerweiterung **zwingend** erforderlich ist.

9 Hardware flashen

Es existieren verschiedene Wege einen Linux Kernel über das Netzwerk zu booten oder in den Speicher zu schreiben.

- Netzwerk
- sdcard
- usb
- spi
- serial
- weitere

Wie sich **die unterschiedlichen Flash Bausteine** auf die o.g. Wege mithilfe des Bootloaders *barebox* be- bzw. überschreiben lassen, ist beschrieben unter [Phy19a, Updating the Software]

10 Commands Bitbake

ACHTUNG: Ein *Image* ist in *Bitbake* ebenso ein *recipe* und wird entsprechend gleich behandelt. Alle Befehle die für ‚einfache‘ recipes gelten (z.b. Kernel Modules oder Applikationen), gelten ebenso für Image-Recipes und werden als solche in Dokumentationen benannt.

10.1 Bitbake Dokumentationsquellen

Eine die offizielle Dokumentation zu *Bitbake* und *Bitbake-Layers* sowie derer Kommandos ist zu finden unter [RB19].

Eine Übersicht über die Kommandos von Bitbake-Layers ist zu finden unter [Gon18, S. 156] und im Anschluss in Anwendung gezeigt.

10.2 Neuer *meta-layer*

Nachfolgende Möglichkeiten stehen bereit, um einen **neuen Meta-daten layer** zu erstellen:

- `yokto-layer create <myLayer>` (**Empfohlen**)
- `bitbake-layers create-layer <mylayer>`

Wie sich einer neuer meta-layer **zum Build hinzufügen lässt**, ist nachzulesen unter Abschnitt 10.3 Seite 31; meta-layer zum build hinzufügen

10.3 meta-layer zum build hinzufügen

Um einen neuen Metadaten-layer zum Buildsystem hinzu zu fügen, muss dieser **in die Datei `/conf/bblayers.conf` eingetragen und so Bitbake bekannt gegeben werden**. Hierzu stehen nachfolgende Möglichkeiten bereit:

- Manueles Eintragen des Pfades zum Meta-layer (**Empfohlen**)

- *bitbake-layers add-layer <layername>*

10.4 Active meta-layer auflisten

Um alle aktiven meta-layer aufzulisten, die im build einbezogen werden, stehen nachfolgenden Möglichkeiten bereit:

- Manuelles Einsehen der Datei *conf/bblayers.conf*
- *bitbake-layers show-layers* (**Empfohlen**)

10.5 Anzeigen aller recipes - beispielsweise aller images

Recipes die während eines Builds aktive sind und ausgeführt werden lassen sich wie folgt anzeigen:

- *bitbake-layer show-recipes [<recipes>]*
- *bitbake-layer show-recipes [<recipes>] | grep image*

10.6 Anzeigen der Tasks eines recipes

Jedes Recipes besitzt eine eine Anzahl von standard Tasks die es direkt oder indirekt implementiert hat (überschreibt) oder erbt.

Alle Task die eine recipe besitzt und somit während eines builds durchlaufen werden lassen sich anzeigen durch:

- *bitbake -c listtasks <recipes>*

Wie gezielt einzelne bzw. bestimmte Task eines Recipes ausgeführt werden ist beschrieben unter Abschnitt 10.7 Seite 32; Ausführen bestimmter Tasks eines Recipes

Alle standard Tasks sind aufgeführt unter[Gon18, Seite 171-172]

10.7 Ausführen bestimmter Tasks eines Recipes

Der Parameter *bitbake -c* (bitbake -command) ermöglicht das Ausführen bestimmter Tasks eines Recipes:

- *bitbake -c <cmd> <recipes>*

10.8 Umgebungsvariablen eines Recipes

Die Umgebungsvariablen welche eine Recipe dauerhaft oder temporär setzt, erweitert oder löscht, lassen sich wie folgt anzeigen.

- *bitbake -e <recipes>*

Es kann gezielt nach einzelnen Variablen gefiltert werden:

- *bitbake -e <recipes> | grep ^ENVVARIABLE>*

10.8.1 Häufig benötigte Umgebungsvariablen

Originale Source-Quelle *bitbake -e virtual/kernel | grep ^SRC_URI=*

Linux Kernel Anbieter *bitbake -e virtual/kernel |
grep ^PREFERRED_PROVIDER_virtual/kernel"*

Image Distributionsversion (Branch) *bitbake -e <recipes/image> |
grep ^LAYERSERIES_COMPAT*

10.9 section name

10.10 Erzeugen von temporären Entwicklungs-Shells

Bitbake ist in der Lage temporäre, gekapselte und vorkonfigurierte Entwicklungs-Shells zu erzeugen und zu öffnen:

- *bitbake -c devshell <recipes>*
- *bitbake -c devpyshell <recipes>*

Dabei unterscheidet bitbake zwischen einer normalen Bash-Shell und einer Python-Shell (Die beiden Skriptsprachen welche innerhalb bitbake Anwendung finden. Beispielsweise wahlweise Python als auch Bash-Shell commandos (kombiniert) innerhalb eines Recipe verwenden.

11 Yocto devtool

Des Yocto *devtool* unterstützt dabei, neue recipes zu erstellen, vorhandene anzupassen oder etwa Sourcedateien zu patchen.

11.1 devtool Kommandos

Eine Übersicht über die Kommandos des Yocto *devtool* liefert [Gon18, Seite 245]

11.2 Arbeiten mit *devtool*

Verschiedene Workflows zum arbeiten mit *devtool* sind zu finden unter [Gon18, Seite 239-249]

12 Bekannte Fehler - Pitfalls

12.1 Never do

Nachfolgende actionen führen in jedem Fall zu einem Fehler

Unterstrich „_“ im Namen Vergebe niemals einen Namen mit einem Unterstrich (_); z.B. für ein Recipe oder einen Ordner

kein CamelCase Namen für Recipes oder beispielsweise Ordner müssen klein geschrieben werden. CamelCase oder Großschreibung ist nicht zulässig.

Umbenennung von Dateien oder Pfaden Benenne niemals einen Ordner, einen Pfad oder ähnliches um. Anderfalls ist ein kompletter Neuanfang mit leerer *Shell* und leerem *temp* und *sstate_cache* Ordner nötig.

Updaten, austauschen, ... von meta-layern, paketen, Konfigurationen ... Neue Abhängigkeiten und veränderungen führen oft zu Fehlern.

12.2 Must do

- Es **muss** der *sstate_cache* manuell gelöscht werden wenn:
 - Kernel Konfigurationen verändert wurden. Z.B. durch *menuconfig* oder *bitbake -c configure virtual/kernel*
 - Module zum Kernel-Autoload hinzu gefügt wurden. Siehe hierzu Abschnitt 7.2 Seite 23; Software Module
 - Eine **Maschinenkonfiguration** *machine/mymachine.conf* verändert wurde.
 - Eine **Distributionskonfiguration** *distro/mydistro.conf* verändert wurde.

- Sollte alles nichts helfen, muss auch der *tmp* Ordner gelöscht werden oder ein *bitbake -force <command_and_recipe* aufgerufen werden. Siehe hierzu auch Unterabschnitt 13.1.1 Seite 41; Lösungsschritte

12.3 Generelle Empfehlungen

Beim Arbeiten mit Bitbake ist zu empfehlen:

- Kleine Änderungen durchführen und testen.
- Eigene Meta-dateien und Änderungen in bestehenden Konfigurationen und Metadateien in einem Repository Versionieren. In einem Fehlerfall ist so ein Roll-back möglich.
- Sichere regelmässig local die Ordner
 - *sstate-cache*
 - *tmp*
 - *deploy* bzw. *tmp/deploy*
- Halte immer ein Lauffähiges *Image*, *rootfs*, und einen funktionierenden *device tree blob* und ggf. ein funktionierendes *SDK* bereit; u.a. um Fehler der Hardware ausschließen zu können.
- Vorgehensweise beim erstellen einer neuen Konfiguration:
 - Eine Bestehende Image / Distro / Machine Configuration nutzen und nach eigenen Anforderungen abändern. Änderungen jeweils in kleinen schritten durchführen.
 - Auf einer minimalen poky Konfiguration neu aufbauen. Ggf. an anderen Beispielen orientieren.
 - Eine Bestehende größere Konfigurationsstruktur zu erweitern ist *nicht empfehlenswert* da Fehleranfällig und schwer zu verstehen.
 - Must-do Abschnitt 12.2 Seite 37; Must do beachten.
 - Never-do Abschnitt 12.1 Seite 37; Never do beachten.
- Variablen sollten wie folgt erweitert werden:

In Konfigurationsdateien durch die Operatoren `_append` oder `_prepend`.

In Recipes durch Operatoren `+=` oder `.=`.

Weitere Informationen und Hintergründe unter [Gon18, Seite 160]

13 Lösung für bekannte Fehler

13.1 ERROR: Fehler beim Bauen eines recipes>

13.1.1 Lösungsschritte

1. Ausführen von *bitbake -c cleanall <recipes>*
2. Force kompilieren mit *bitbake -f <recipes>*
3. Sollte nichts helfen, je nach situation einen oder alle der nachfolgenden Ordner löschen.
 - *sstate* Ordner
 - *deploy* Ordner
 - *tmp* Ordner
 - Alles weitere im Projekt Ordner, mit Ausnahme des *conf* Ordners.

13.2 Logging in Recipes

In Bitbake recipes lässt sich **python** oder **bash loggig** nutzen. Dabei werden durch die Yocto und OpenEbedded Community bereits vordefinierte *Logging Classes* (*.bbclass) bereit gestellt die unterschiedliche Informationen loggen können, angefangen von:

- *plain* und *notes*
- *warnings* und (*fatal-*) *errors*
- *debug* informationen

Eine Übersicht ist gelistet unter [Gon18, S.79-80].

13.3 Abhängigkeiten (Dependencies) visualisieren

Bitbake ist in der Lage, Abhängigkeiten zwischen Recipes bzw. zwischen Software Komponenten als Graphen zu visualisieren. Hierbei wird mittels *Graphviz* .dot files zwischen den Abhängigkeiten generiert. *.dot Dateien lassen sich anschließend in eine Grafik oder PDF umwandeln. Beispielhaft:

Listing 13.1: Dependency Graph

```
1 [user@host]/yocto$: source ./[<path>]environment-setup-*.sh
   sh
[user@host]/yocto$: bitbake -g <recipes> [-l <dependencies>
   ignore >]
3 [user@host]/yocto$: dot -Tpng *.dot -o dependency.png
[user@host]/yocto$: dot -Tpdf *.dot -o dependency.pdf
```

Alternativ stellt Yocto/Openembedded einen *dependency explorer* bereit. Hierzu Bitbake aufrufen mit

- bitbake -g -u taskexp <recipe>

Weitere Abhängigkeits-Probleme zwischen Dateien/Versionen lassen sich ermitteln über die Dateien:

- ./tmp/stamps/sigdata
- ./tmp/stamps/siginfo

Hierbei handelt es sich um *Python-Datenbanken* die sich wie folgt ausgeben und mit dem aktuellen Abseitsstand vergleichen lassen:

- bitbake-dumpsig [sigdata / siginfo]
- bitbake-diffsig [sigdata / siginfo]

14 Ausblick

14.1 Nächste Schritte

14.1.1 Entwicklungswerkzeuge als Docker Container

Es wäre sinnvoll die Entwicklungstools in einem oder mehrere Docker container vor-konfiguriert zur verfügug zu stellen. Hierzu zählen unter anderem:

Eclipse Vorkonfiguriertes Eclipse inclusive Plugins, Cross-Compile und Remote de-bugging Einstellungen.

QT5 Vorkonfiguriertes QT mit Crosscompile und Remot Debugging

TFTboot und NFSROOT Server in einem Container vorkonfiguriert bereitstellen

14.1.2 Erweiterung der Scripte

Das *run.sh* script sollte so erweitert werden, das es ‚post‘ oder ‚pre‘ Aufgaben vor oder nach dem aufrufen der *dockerjobs.sh* durchführt oder andere postbuild oder prebuild scripte aufruft. Denkbar wären:

- Kopieren des zImages und Device Tree Blob (DTB) in das *TFT-boot* Verzeichnis
- Kopieren und extrahieren des rootfs in das nfs-rootfs Verzeichnis

Das *run.sh* script erzeugt das *dockerjobs.sh* Script sollte dem run.sh script parame-ter übergeben werden. Anschließend startet das run.sh script den Docker container in definierter version (gesetzt über Parameter oder direkt innerhalb des run.sh scripts). Das dockerjobs.sh wird innerhalb docker durch das image aufgerufen und enthält alle aufgaben welche durch den Container in batchmode erfüllt werden sollen. Das docker-jobs.sh script lässt sich manuell erweitern / erstellen. Es wird nur überschrieben, wenn dem run.sh Ausführungsbefehle übergeben werden.

14.2 Security

Gerade zu Beginn der Entwicklung bietet sich an, zunächst auf viele Sicherheitsfunktionen zu verzichten, da der gesamte Entwicklungsprozess bereits komplex ist und ausreichend potentielle Fehlerquellen besitzt.

Dennoch ist mindestens zum Ende eines Projektes, vor Veröffentlichung, das Sicherheitskonzept überarbeitet werden. So müssen beispielsweise nachfolgende Themen bewertet und bearbeitet werden

- Linux Kernel härten
- SELinux
- Gesamt System härten
- SMACK
- Benutzer, Passwörter, Zugriffsrechte, ACLs
- Netzwerkschnittstellen und Kommunikation absichern. Beispielweise durch verschlüsselte Datenübertragung
- Debugging, Flashing, Tracing Schnittstellen entfernen oder einschränken

14.3 Lizenzen

Eine wichtiges Thema ist die Lizenzierung neuer Softwarekomponenten, welche zusammen mit Open-Source paketen (i.d.R. mindestens dem Linux Kernel) genutzt werden und mit diesen Kompatible sein muss. So muss ich über folgende Kombinationen gedanken gemacht werden.

Lizenverwaltung und Kompatibilität von:

- Neuen Softwarekomponenten
- Bestehenden Softwarekomponenten
- Verwendeten / Eingebundenen Softwarepaketen, z.B. über genutzte meta-datan Layer aus Community Quellen.

Glossar

B | D | F | G | M | N | O | P | Q | R | T | W | Y

B

Bitbake Bitbake ist ein Framework ähnlich wie GNU Make, bestehend aus Python Sripten welche das Erstellen von Linux Distributionen mittels Metadaten koordiniert.. 1

D

Distribution Als Distribution bezeichnet man eine Zusammenstellung von (Software-) Paketen, Versionen, Konfigurationen, Einstellungen, usw. die als Gesamtpaket veröffentlicht sind oder werden und in sich ohne weiteres Zutun eine definierte Aufgabe erfüllen. Beispielsweise ist ein Betriebssystem eine solche Zusammenstellung das ohne weiters Zutun für einen Satz von Anwendungsfällen genutzt werden kann. . 19

Docker Docker ist eine Software zur virtualisierung von einzelnen Anwendungen. 3, 45

Dockerfile Ein Dockerfile beschreibt wie ein Docker Container (Image) gebaut werden soll, genauer welche Tools und Konfigurationen der Container enthalten soll.. 11

F

Feature Merkmal oder Eigenschaft, wie eine besondere Funktionalität. 19, 23

G

GIT Git ist eine freie Software zur verteilten Versionsverwaltung von Dateien, die durch Linus Torvalds initiiert wurde. Es arbeiten im Gegensatz zu anderen Versionsverwaltungen mit lokalen als auch externen Repositories. . 11

M

Metadaten Metadaten oder Metainformationen sind Informationen über andere Daten. 1

Metadaten Metadaten oder Metainformationen sind Informationen über andere Daten. 46

Metadatei Eine Datei, welche Metadaten (Informationen zu anderen Daten) über andere Daten enthält.. 1

N

NFS Das Network File System (NFS), (auch Network File Service) ist ein Netzwerkprotokoll, das den Zugriff auf Dateien über ein Netzwerk ermöglicht. Dabei werden die Dateien nicht wie z. B. bei File Transport Protocol (FTP) oder Trivial File Transfer Protocol (TFTP) übertragen, sondern die Benutzer können auf Dateien, die sich auf einem entfernten Rechner befinden, so zugreifen, als ob sie auf ihrer lokalen Festplatte abgespeichert wären. Unter Unix Betriebssystemen lassen sich diese Netzwerkfreigaben direkt in das Dateisystem einhängen entsprechend wie z.B. Festplatten. . 11

O

OpenEmbedded Eine Community Gruppe welche eine Software Buildumgebung namens Bitbake pflegt und Metadaten für dieses Buildsystem bereitstellt, welche von dieser Buildumgebung verwendet wird um Softwarepakete zu übersetzen. 1, 48

P

Poky Minimalistisches Betriebssystem der Yocto Community das mitsamt seiner Tools in einer QEMU virtuellen Maschine lauffähig ist. . 2, 48

Q

QEMU QEMU ist (von englisch „Quick Emulator“) ist eine freie Virtualisierungssoftware, die die gesamte Hardware mitsamt Prozessorinstruktionen einer einer Zielhardware emuliert. Dabei ist es sehr Recourcenschonend und performant.. 2, 46, 47

R

Recipies Rezepte oder Anleitungen. 1

T

TFTP Das Trivial File Transfer Protocol (TFTP) ist ein im Funktionsumfang stark vereinfachtes File Transport Protocol und reduziertes Protokoll, das lediglich das senden und das empfangen von Dateien ermöglicht.. 11

Tool Ein Tool (Informationstechnik, zu Deutsch *Werkzeug*) beschreib beschreibt ein Stück Software das Hilfsaufgaben ausführt um eingesetzt bei anderen Aufgaben zu unterstützen. 1, 47

W

Workflow Ein Wokflow ist ein Arbeitsablauf und beschreibt Schritte in ihrer Reihenfolge die nötig sind, um eine Aufgabe, Arbeitspaket oder etwa eine Anweisung zu erfüllen. Dabei sind die Arbeitsschritte häufig wiederkehrend in anderen Arbeitsabläufen. . 35

Wrapper Ein Wrapper ist ein Softwareinterface das Interaktion mit einem anderen Softwarepaket kapselt und i.d.R. nach außen hin vereinfacht. . 1

Y

Yocto Das Yocto Project bezeichnet eine Community Gruppe, welche sich zur Aufgabe gemacht hat, dass erstellen von Linux Distributionen für eingebettet System zu vereinfachen. Die Yocto community stellt Meta-Daten (Recipes) bereit, um ein minimalistisches Betriebssystem mit grundlegenden Linux Tools unter der Virtualisierungsumgebung *QEMU* für verschiedene Architekturen starten zu

können. Das Minimalisten Betriebssystem der Yocto Community wird *Poky* genannt. Zudem pflegt es zusammen mit der OpenEmbedded community das Buildsystem *bitbake*. . 1, 46

Literatur

Books

- [CRK05] Jonathan Corbet, Alessandro Rubini und Greg Kroah-Hartman. *Linux Device Drivers* -. 3. Aufl. Sebastopol: Ö'Reilly Media, Inc.", 2005. ISBN: 978-0-596-00590-0.
- [Gon18] Alex Gonzalez. *Embedded Linux Development Using Yocto Project Cookbook* -. 2nd Revised edition. Birmingham: Packt Publishing, 2018. ISBN: 978-1-788-39921-0.
- [Ker10] Michael Kerrisk. *The Linux Programming Interface* -. 1. Aufl. München: No Starch Press, 2010. ISBN: 978-1-593-27220-3.
- [QK15] Jürgen Quade und Eva-Katharina Kunst. *Linux-Treiber entwickeln - Eine systematische Einführung in die Gerätetreiber- und Kernelprogrammierung - jetzt auch für Raspberry Pi*. 4. akt. und erw. Aufl. Heidelberg: Dpunkt.Verlag GmbH, 2015. ISBN: 978-3-864-90288-8.
- [RS13] Stephen A. Rago und W. Richard Stevens. *Advanced Programming in the UNIX Environment* -. 3rd edition. Amsterdam: Addison-Wesley, 2013. ISBN: 978-0-321-63773-4.

Online resources

- [ope19] openEmbedded. *Open Embedded Recipe Indexer*. 2019. URL: <https://layers.openembedded.org/layerindex/branch/master/recipes/> (besucht am 01. 12. 2019).
- [Phy19a] PhyTec. *PhyTec Mira BSP Manual*. Device Booting & Flashing, Peripherals Mira6. 2019. URL: <https://www.phytec.de/documents/?title=1-814e-4-imx6-bsp-manual> (besucht am 01. 12. 2019).
- [Phy19b] PhyTec. *PhyTec Mira Development Guide*. Host & Tool Setup. 2019. URL: <https://www.phytec.de/documents/?title=1-833e-8-development-environment-guide> (besucht am 01. 12. 2019).
- [Phy19c] PhyTec. *PhyTec Mira Downloads*. Downloads Mira6 Board. 2019. URL: <https://www.phytec.de/produkt/system-on-modules/phycore-imx-6-phyboard-mira-download/> (besucht am 01. 12. 2019).
- [Phy19d] PhyTec. *PhyTec Mira Yocto Manual*. Conclusion Yocto Mega Manual. 2019. URL: <https://www.phytec.de/documents/?title=1-813e-7-yocto-reference-manual> (besucht am 01. 12. 2019).
- [RB19] Chris Larson Richard Purdie und Phil Blundell. *BitBake User Manual*. 2019. URL: <https://www.yoctoproject.org/docs/3.0/bitbake-user-manual/bitbake-user-manual.html> (besucht am 01. 12. 2019).
- [Rif19] Scott Rifenbark. *Yocto Mega Manual*. 2019. URL: <https://www.yoctoproject.org/docs/current/mega-manual/mega-manual.html> (besucht am 01. 12. 2019).

-
- [var19] variwiki. *Yocto Programming with Eclipse*. 2019. URL: http://variwiki.com/index.php?title=Yocto_Programming_with_Eclipse (besucht am 01. 12. 2019).
- [Yoc19a] Yocto. *Live Coding with the Yocto Project*. 2019. URL: https://youtube.com/channel/UC2_mG3h-AxxK2oz08j2bz2g (besucht am 01. 12. 2019).
- [Yoc19b] Yocto. *Yocto Project Reference Manual*. 2019. URL: <https://www.yoctoproject.org/docs/3.0/ref-manual/ref-manual.html> (besucht am 01. 12. 2019).