# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Abkürzungsverzeichnis

**SDK** Software Development Kit

**DTB** Device Tree Blob

# 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, dass erstellen von Linux Distributionen für eingebettet System zu vereinfachen.

Zusammen mit der OpenEmbedded Community pflegt das Yocto Project eine Software Build Umgebung mit dem Namen Bitbake, bestehend aus Python- und Shell-Scripten, welches das Erstellen von Linux Distributionen koordiniert. Parallel zu Bitbake entstehen und wachsen innerhalb der Yocto Community verschiedene tools wie z.B. Wrapper, welchen das Arbeiten mit Bitbake vereinfachen sollen, indem diese Tools beispielsweise einzelne Workflows abbilden.

Des Weiteren pflegen beiden Communities Dateien mit Metadaten (Metadatendatei die in Form von Regeln beschreiben wie Software Pakte innerhalb für unterschiedliche Distribution und Hardware durch Bitbake gebaut werden müssen. erforderlich sind.

Diese Regeln werden Recipies genannt. Regeln bzw. diese Metadaten und somit auch das Build-System Bitbake arbeiten nach einen Schichten Modell. Dabei beschreiben Meta-Daten auf unterster Schicht allgemeine grundlegende Anleitungen zum Übersetzten der wichtigsten Funktionen eines Betriebssystems. Höhere Schichten erweitern (detaillieren) oder überschreiben diese grundlegenden Rezepte in immer höheren Schichten. So setzt auf Beschreibungen der Hardware (BSP, Board Support packed Schichten) Schichten der Software- und Anwendungsschicht auf. Ein solches Schichtenmodell ermöglicht es, einzelne Schichten auszutauschen oder darunterliegende Einstellungen abzuändern. 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 aufsetzten und genutzt werden können um ein Betriebssystem nach eigenen Wünschen gestalten ( zusammen setzten) 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 ??, Seite ??.

## 1.2 Docker

Bitbake benötigt, neben einem aktuellen Python (2) Interpreter, verschiedene Tools. So u.a. Git zum Herunterladen von Source Detain. Alle diese Abhängigkeiten wurden in einem Docker Container zusammengefasst. Bitbake kann unter unschädlichen Betriebssystemen und Plattformen auf gleiche Weise genutzt werden, ohne dass es und seine Abhängigkeiten neu konfiguriert werden müssen. Im Gegensatz zu anderen Virtualisierungstechniken hat Docker trotz Virtualisierungstechniken keinen großartigen Performance Verlust. Von einem Gebrauch von klassischen Virtualisierungstechniken ist aus Performance Gründen dringend abzuraten.

# 2 Grundlage

## 2.1 Bitbake

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

- Reihenfolge der angepassten config files gepasst werden.
- Parsen der config files.
- Buildschritte (Siehe 2.2 Seite 3)
- Packetieren in unterschiedliche Pakettypen
- 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 (fetchen) und Sammeln von Source Dateien (z.B. Git Repositorien, ftp Servern oder lokalen Dateien.)
- Übersetzten, konfigurieren, patchen, installieren, verifizieren usw. von Paketen auf mehreren Prozessor Kernen
- Bereitstellen von Entwicklung und Verwaltungstools

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

### 2.2.1 Fatch Task

Eine der Hauptaufgaben von Bitbake besteht dadrinne, die jeweiligen Softwarepakte in Ihren benötigten Versionen und Revisionsständen zusammen aus unterschidlichen Quellen zusammen 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 Revisionständen.
- 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 Reihefolge 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 Reihefolge innerhalb bitbakes. Siehe [Gon18, S.53]

## 2.3 Configurations-Dateien *.conf

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

# 3 Setup your host

## 3.1 Erforderliche Pakete

Das yocto Docker file zeigt eine Liste aller nötigen Ubuntu Pakte die zum Arbeiten mit der Yocto / OpenEmbedded Build Umgebung auf einem Host benötigt werden, sollte nicht mit dem Docker Image gearbeitet werden wollen.

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

- git
- docker
- TFTP Server
- NFS Server
- microcom
- eclipse
- qt5
- qt5Creator
- openssh-server

## 3.2 Host Konfiguration

Nachfolgende Pakete benötigen weitere Konfigurationen

**Docker:**

- Docker Service starten
- Docker-yocto Image bauen: „docker build -t yocto . “
- 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 deamon (/etc/default/tftpd-hpa) oder xinitd Service (/etc/-xintd.d/tftp) konfigurieren
- Server neu starten
- **BEISPIEL** im „BSP Manual“unter phytec.de; Stichwort „Booting the Kernel from Network“(Booting_the_Kernel_from_Network) [Phy19] oder unter [Gon18, S. 44]

**NFS Server:**

- NFS Server konfigurieren (/etc/exports)
- NFS Server neu starten
- **BEISPIEL** im „BSP Manual“unter phytec.de; Stichwort „Booting the Kernel from Network“(Booting_the_Kernel_from_Network) [Phy19] oder unter [Gon18, S. 45]

**Microcom**

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

**Eclipe**

- Weiters im Kapitel 4; Seite 9

**QT5Creator**

- Weiters im Kapitel ??; Seite ??

**Yocto Areitsverzeichnis**

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

**Python2 als standart interpreter** Yocto/Openembedded Tools bauen auf Python2 auf. Daher ist es nötig einen symlink oder alias auf Python2 zu setzten. Z.b. *alias python=python2*

**Proxy und Routen** Je nach Netzwerkinfrastruktur müssen Proxy und Netzwerkrouten 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 **http://downloads.yoctoproject.org/releases/eclipse-plugin/2.6.1/oxygen/** hinzu gefügt werden.

## 4.2 Cross Compile und Remote Debugging Einstellungen

TODO

# 5 Module entwicklung

## 5.1 Kernel Module

### 5.1.1 Neues Kernelmodul

### 5.1.2 Kernel anpssen - features aktivieren

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

- manuelles Anpassen der .config datei
- GUI/Menü basiert mittels Befehl
  - menuconfig
  - xconfig
  - gconfig
- bitbake -c menuconfig virtual/kernel

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

## 5.2 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

### 5.2.1 Eigene Kernel modul erstellen und einbinden

Seite [Gon18, S. 121–125] zeigt beipielhaft, wie sich ein eigenes Kernelmodul erstellen und in die Buildumgebung Bitbake einminden 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:

- ??
- ??

## 5.3 Flashen des image

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

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

# 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 Revipe 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 <hw-metalayer>/conf/machine/<machine> Konfigurationsdatei.

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 Yoc19]
- [Abschn. Machine_Features Rif19]

Zudem definiert die Maschinenkonfiguration

- welche Linux distribution verwendet werden soll bzw. kann. Siehe hierzu ??, Seite ??.
- Welcher Bootloader Anwendung finden soll. Siehe hierzu ?? Seite 14.

- Welche Softwarepakete, Treiber, Kernelmodule, Konfigurationen, usw. für eine Zielplattform zwingend zum starten erfoderlich 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-Platform 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 Hardwareplatformen immer neu übersetzt, beinhaltet jedoch die selben Tools, Anwendungen, Konfigurationen oder sonstigen 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 Stiegspunkt einer Software oder eines Betriebssystems zu finden ist (Speicherkomponente, Sektor, ...) und übernimmt die Aufgabe solche Software über eine definierte Schhnittstelle 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)

Änlich wie die Hardware und Zielplattform (Siehe 6.1 Seite 13) sowie eine Betriebssystemdistribution muss auch der Bootlooder 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 Maschinen definition einher.

# 7 Lösung für bekannte Fehler

## 7.1 ERROR: Fehler beim Bauen eines recipes>

### 7.1.1 Lösung

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.

## 7.2 Logging in Recipes

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

- *plain* und *notes*
- über *warnings* und *(fatal-) errors*
- bis bin zu *debug* informationen

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

### 7.3 Abhängigkeiten (Dependencies) visualisieren

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

Listing 7.1: Dependency Graph

```
1 [user@host]/yoctopath $: source ./[<path>]enviroment-setup-*.sh
  [user@host]/yoctopath $: bitbake -g <recipes> [-I <dependencie to ignore> ]
3 [user@host]/yoctopath $: dot -Tpng *.dot -o dependancy.png
  [user@host]/yoctopath $: dot -Tpdf *.dot -o dependancy.pdf.
```

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

- bitbake -g **-u taskexp** <recipe>

Weitere Abhängikeits-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]

# 8 Ausblick

## 8.1 Nächste Schritte

### 8.1.1 Entwicklungswerkzeuge als Docker Container

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

**Eclipse** Vorkonfiguriertes Eclipse inclusieve Plugins, Cross-Compile und Remote debugging Einstellungen.

**QT5** Vorkonfiguiertes QT mit Crosscompile und Remot Debugging

TFTboot und NFSROOT Server in einem Container vorkonfiguriert bereitstellen

### 8.1.2 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 parameter 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 dockerjobs.sh script lässt sich manuell erweitern / erstellen. Es wird nur überschrieben, wenn dem run.sh Ausführungsbefehle übergeben werden.

## 8.2 Security

Gerade zu Beginn der Entwicklung bietet sich an, zunächst auf viele Sicherheitfunktionen 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

## 8.3 Lizenzen

Eine wichtiges Thema ist die Lizensierung 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 Sofwarekomponenten
- Bestehenden Sofwarekomponenten
- Verwendeten / Eingebundenen Softwarepaketen, z.B. über genutzte meta-datan Layer aus Community Quellen.

# Glossar

B | D | F | M | R | W | Y

## B

**Bitbake** Bitbake ist ein Framework ähnlich wie GNU Make, bestehend aus Phython Srcipten 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. . 13, 21

**Docker** Docker ist eine Software zur virtualisierung von einzelnen Anwendungen. 2

## F

**Feature** Merkmal oder Eigenschaft, wie eine besondere Funktionalität. 11, 13

## M

**Metadaten** Metadaten oder Metainformationen sind Informationen über andere Daten. 1

**Metadaten** Metadaten oder Metainformationen sind Informationen über andere Daten. 21

**Metadatendatei** Eine Datei, welche Metadatens (Informationen zu anderen Daten) über ndere Daten enthält.. 1

## R

**Recipies** Rezepte oder Anleitungen. 1

## 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. . 1

## Y

**Yocto Project** Eine Community Gruppe welche eine Software Buildumgebung pflegt und weiterentwickelt, sowie Metadaten für diese Buildumgebung zur verfügung stellt um ein minimalistisches Linux system mit grundlegenden Tools unter der virtuallisierungsumgebung QEMU zu starten.. 1

# Literatur

## Books

[Gon18] Alex Gonzalez. *Embedded Linux Development Using Yocto Project Cookbook* -. 2nd Revised edition. Birmingham: Packt Publishing, 2018. ISBN: 978-1-788-39921-0.

## Online resorces

[Phy19] PhyTec. *PhyTec Mira BSP Manual*. Device Booting & Flashing, Peripherals Mira6. 2019. URL: https://www.phytec.de/documents/?title=l-814e-4-imx6-bsp-manual (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).

[Yoc19] Yocto. *Yocto Project Reference Manual*. 2019. URL: https://www.yoctoproject.org/docs/3.0/ref-manual/ref-manual.html (besucht am 01. 12. 2019).