Embedded Linux

danke Yocto und OpenEmbedded

Von Marco Israel

Dezember 2019

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.

1.3 Literatur

Nachfolgende Quellen sind zu empfehlen

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

Phytec Hardware Flash, Grundlagen zu PhyTec HW Schnittstellen und Device Tree B
[Kapitel *Booting_the_System* und *Updating_the_System* Phy19a]

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 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 5)
- 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.

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 Wird übergehend in *.conf Dateien verwendet. Genau wie die Standardoperatoren erweitern Sie eine Variable. Sie werden jedoch zu einem späeren Zeitpunkt als die standard Operatoren verarbeitet. Weiteres ist zu finden unter: [Gon18, Seite 160]

INHERRIT Genutzt ausschließlich in *.conf Dateien zun erben (includieren) von configurations Klassen.

inherrit Genutzt ausschließlich in recipes *.bb und *.bbappend zum erben (inkludieren) von Bitbake-Classen.

include Wenn Datei vorhanden, dann include ihren Inhalt. Ansonsten ignoriere den Befehl. *include* lasst sich sowohl in configurations-Dateien als auch recepes verwenden um alle andren Arten von Dateien zu inkludieren (andere Recipes, Configs, ...)

require Wie include nur das die Datei vorhanden sein muss, ansonsten beendet bitbake mit einem error.

2.5 Yocto Tools

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

- *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 manuel 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 ?? ?? ?? .

- Patchen oder erweitern von Softwarekomponenten die Bereits in den Linux Kernel und Bitbake integriert sind, lassen sie am einfachsten durch Tool bewerkstelligen, die von der Yocto / OpenEmbedded Community bereit gestellt werden. Hierzu zählt z.B. das s.g. *devtool*. Siehe hierzu Kapitel 10 Yocto devtool 35 . 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

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) [Phy19a] 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) [Phy19a] 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 15

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 QT5 einrichten

5.1 Qt Creator 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]

5.2 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 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 Yoc19b]
- [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 20.

- 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 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 19) 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 Module entwicklung

7.1 Kernel Module

7.1.1 Neues Kernelmodul

7.1.2 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**

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

7.2.1 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:

- [QK15]
- [CRK05]

Nachschlagewerke zum Thema Linux/Unix Kernel Schnitstellen (System-Calls) sind die Bücker:

- [Ker10]
- [RS13]

7.3 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

8 Applikationsentwicklung

8.1 Einbinden von Applikationen in vorhandene Images

h

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 ÎAYERSERIES_COMPAT*
2. Seite ?? aufrufen
3. **Branch** auswählen und ggf. Reiter **recipies** 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 metadaten-layer bereits existiert: Z.b. durch z.B.
   - Manuelles suchen
   - *find –name <new-meta-layer>*
   - *bitbake-layers show-layer | grep «new-meta-layer>*
6. Layer packet herunter laden 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 Metadatenlayer herunterladen und ebenfalls dem Projekt hinzufügen.
   - Abhängigkeiten lassen sich durch *anklicken des Recipie names* unter ?? einsehen.
   - oder direkt ausgeben durch: *bitbake-layers layerindex-show-depends <layer-name>* und anschließend herunterladen und hinzufügen durch *bitbake-layers layerindex-fetch <layer-name>* bitbake-layers add-layer <path/to/your/meta-data-layer>

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. fehlende Abhängigkeiten): *bitbake-layers show-layers*

8.2.1 Sotware dem Image hinzufügen

Um ein Softwarepaket einem Image hinzu zu fügen muss die Variable *IMAGE_INSTALL* gesetzt oder erweitert werden. Hier gilt zu entschieden ob dieses *global* oder *image-lokal* geschehen soll:

- Setzten oder erweitern in *./conf/loccal.conf*
- Abändern des <image-recipes>.bb files durch setzten / 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 opratoren 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 Operaturen += 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, das die Konfiguration abschließend Anwendung findet

operatoren += .= .. Sind für recipes

ACHTUNG ACHTUNG: beachte das < >, welches ein führendes Leerzeichen symbolisiert und zwingend erforderlich ist.

9 Commands Bitbake

NOTE: Ein *Image* ist in der Bitbake welt ebenso ein *recipie* und wird entsprechend gleich behandelt. Alle befehle die für ‚einfache'recipes gelten (z.b. Kernel Modules oder Applikationen), gelten ebenso für image-recipes.

9.1 Bitbake Dokumentations Quellen

Eine die offizielle Dokumentation zu *Bitbake* und *Bitbake-Layers* mitmitsamt 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.

9.2 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 9.3, Seite 31.

9.3 meta-layer zum build hinzufügen

Um einen neuen Metadaten-layer zum Buildsystem hinzu zu fügen, muss dieser in die datei */conf/bblaysers.conf* Bitbake bekant gegeben werden. Hierzu stehen nachfolgende Möglichkeiten bereit:

- Manueles Eintragen der des Pfades zum meta-layer (**Empfohlen**)

- bitbake-layers add-layer <layername>.

9.4 Active meta-layer auflisten

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

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

9.5 Anzeigen aller recipes - beispielsweise aller images

Recipies 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

9.6 Anzeigen der Tasks eines recipes

Jedes Recipes besitzt eine eine Anzahl von standard Tasks die es direkt oder indirekt implementiert hat 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 unter 9.7, Seite 32 beschrieben.

9.7 Ausführen bestimmter Tasks eines Recipes

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

- bitbake -c <cmd> <recipes>

9.8 Umgebungsvariablen eines Recipes

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

- bitbake -e <recipes>

Es kan geziehlt nach einzelnen Variablen gefiltert werden:

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

9.8.1 Häufig benötigte Umgebungsvariablen

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

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

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

9.9 section name

9.10 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 Phython-Shell (Die beiden Skriptsprachen welche innerhalb bitbake Anwendung finden. Beispielsweise wahlweise Python als auch Bash-Shell commandos (kombiniert) innerhalb eines Recipe verwenden.

10 Yocto devtool

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

10.1 devtool Kommandos

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

10.2 Arbeiten mit *devtool*

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

11 Bekannte Fehler - Pittfalls

11.1 Never do

Nachfolgende actionen füren 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 Recipies oder beispielsweise Ordner müssen klein geschreiben werden. CamelCase oder Großschreibung ist nicht zulässig.

Umbenennung von Dateien oder Pfaden Benenne niemals einen Ordner, einen Pfad oder ähnliches um. Anderfalls ist ein komletter 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.

11.2 Must do

- Es **muss** der *sstate chache* 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 Software Module 23
  - Eine **Maschinenkonfiguration** *machine/mymachine.conf* verändert wurde.
  - Eine **Distributionsconfiguration** *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 herzu auch Unterabschnitt 12.1.1 Lösungschritte 41

11.3 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 Reposetorie Versionieren. In einem Fehlerfall ist so ein Rollback 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-dos Abschnitt 11.2 Must do 37 beachten.
  - 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]

12 Lösung für bekannte Fehler

12.1 ERROR: Fehler beim Bauen eines recipes>

12.1.1 Lösungschritte

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.

12.2 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*
- *warnings* und *(fatal-) errors*
- *debug* informationen

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

12.3 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 12.1: Dependency Graph

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

13 Ausblick

13.1 Nächste Schritte

13.1.1 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

13.1.2 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.

13.2 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

13.3 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. . 19, 45

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

F

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

M

Metadaten Metadaten oder Metainformationen sind Informationen über andere Daten. 1

Metadaten Metadaten oder Metainformationen sind Informationen über andere Daten. 45

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, 35

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

[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 resorces

[Phy19a] PhyTec. *PhyTec Mira BSP Manual*. Device Booting & Flashing, Pheripherals Mira6. 2019. URL: https://www.phytec.de/documents/?title=l-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=l-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).

[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).

[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).