Erste Schritte mit Linux und Buildroot

1 Vorwort

Endlich geschafft. Nachdem Sie das Seminar hinter sich gelassen und angefangen haben, Ihre Projektidee zu dokumentieren, kann es endlich losgehen. Das ganze sechste Semester steht Ihnen jetzt zur Verfügung, um Ihre Idee umzusetzen. Zwar haben Sie während Ihres Studiums schon das ein oder andere Programmierprojekt bearbeitet, dieses mal gehen wir aber einen Schritt weiter und betrachten den kompletten Entwicklungsprozess – vom Konzept, über die Programmierung bis zur Integration der Software zu einer eigenen Embedded Firmware. Dabei spielt Linux gleich in zweierlei Hinsicht eine zentrale Rolle. Sowohl unser Entwicklungssystem als auch die Firmware selbst basieren auf GNU/Linux. Freilich handelt es sich bei eingebetteten Systemen um ein sehr spezielles Anwendungsgebiet, das hier gelernte Wissen können Sie aber ohne Probleme auf andere Bereiche übertragen. Egal ob Sie später Webserver unter Linux administrieren, eine Cloud-Umgebung aufsetzen oder eigene Docker-Container erstellen, die grundlegenden Zusammenhänge sind immer dieselben.

Zur Erstellung der Firmware nutzen wir Buildroot[1], da es einen guten Kompromiss zwischen flexibler Konfiguration und einfacher Bedienung bietet. Hierfür steht Ihnen eine bereits vorkonfigurierte, virtuelle Maschine auf Basis von Debian[2] zur Verfügung. Dieses Dokument soll Ihnen die wichtigsten Schritte im Umgang mit der VM, Linux und Buildroot aufzeigen. Es empfiehlt sich daher, alle Kapitel vor Projektstart durchzulesen und auszuprobieren. Sollten Sie dabei Fragen haben oder Ihnen Verbesserungsvorschläge einfallen, freue ich mich schon darauf, von Ihnen zu hören. Bis dahin wünsche ich Ihnen viel Spaß und gutes Gelingen.

Dennis Schulmeister-Zimolong

1 http://www.buildroot.org
2 http://www.debian.org

2 Inbetriebnahme der Linux VM

2.1 Einrichten von VirtualBox

Alle Programme, die Sie zum Erstellen der Firmware benötigen, bekommen Sie in Form einer virtuellen Maschine auf Basis von Debian-Linux zur Verfügung gestellt. Um diese nutzen zu können, müssen Sie also zunächst VirtualBox[3] auf Ihrem Computer installieren. Anstelle von VirtualBox können Sie natürlich auch andere Hypervisor wie z.B. VMware nutzen, allerdings müssen Sie dann das Port Forwarding und die mit dem Host geteilten Ordner manuell einrichten.

Nachdem Sie VirtualBox installiert haben, starten Sie die Admin-Oberfläche und wählen Sie den Menüpunkt *File → Import Applicance…* aus. Im drauf folgenden Fenster wählen Sie die Datei `IoT-Embedded-Buildroot.ova` aus und klicken auf *Import*. Der Import dauert mehrere Minuten.

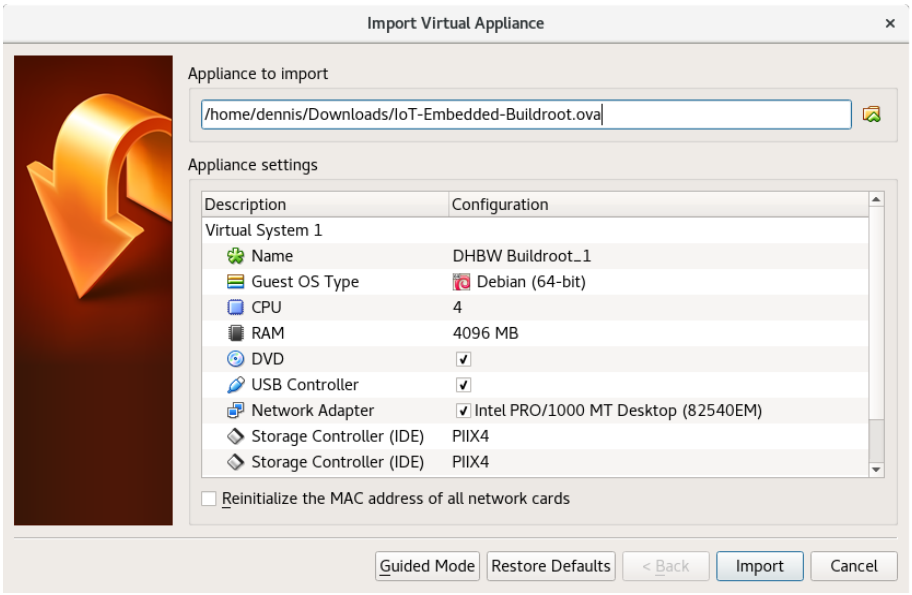

Abb. 1: Import der VM in VirtualBox

Die neue VM sollte nun in der Übersicht auftauchen und kann per Doppelklick oder über das Kontextmenü gestartet werden. Doch bevor Sie die VM das erste mal starten, müssen Sie erst die folgenden Einstellungen überprüfen:

- Guest OS Type: Debian (64-bit)

Wenn Ihnen bei *Guest OS Type* nur 32-bit-Systeme angeboten werden, liegt dies an den BIOS/UEFI-Einstellungen Ihres Computers. Dort muss der „Intel VTx“-Befehlssatz eingeschaltet sein. Je nach Hersteller kann es sein, dass die Option „Virtualization Support“, „VTx-Extensions“ oder ähnlich heißt.

- Portweiterleitung des lokalen Ports 2222 an Port 22 der VM
- Teilen des Verzeichnisses `/home/buildroot/custom` mit dem Host
- Teilen des Verzeichnisses `/home/buildroot/shared` mit dem Host

Öffnen Sie hierfür das Kontextmenü der VM und wählen Sie den Eintrag *Settings…* aus. Innerhalb des Fensters kontrollieren Sie folgende Seiten und passen sie an Ihre Umgebung an. Insbesondere die Pfade der geteilten Verzeichnisse sollten Sie anpassen. Entpacken Sie hierfür die Datei `IoT-Embedded-Buildroot.zip` irgendwo auf Ihrem Rechner und passen Sie die Pfade so an, dass sie auf die beiden Verzeichnisse `custom` und `shared` darin verweisen.

3 https://www.virtualbox.org

- *Network → Adapter 1 → Advanced → Port Forwarding*
- *Shared Folders → custom*
- *Shared Folders → shared*

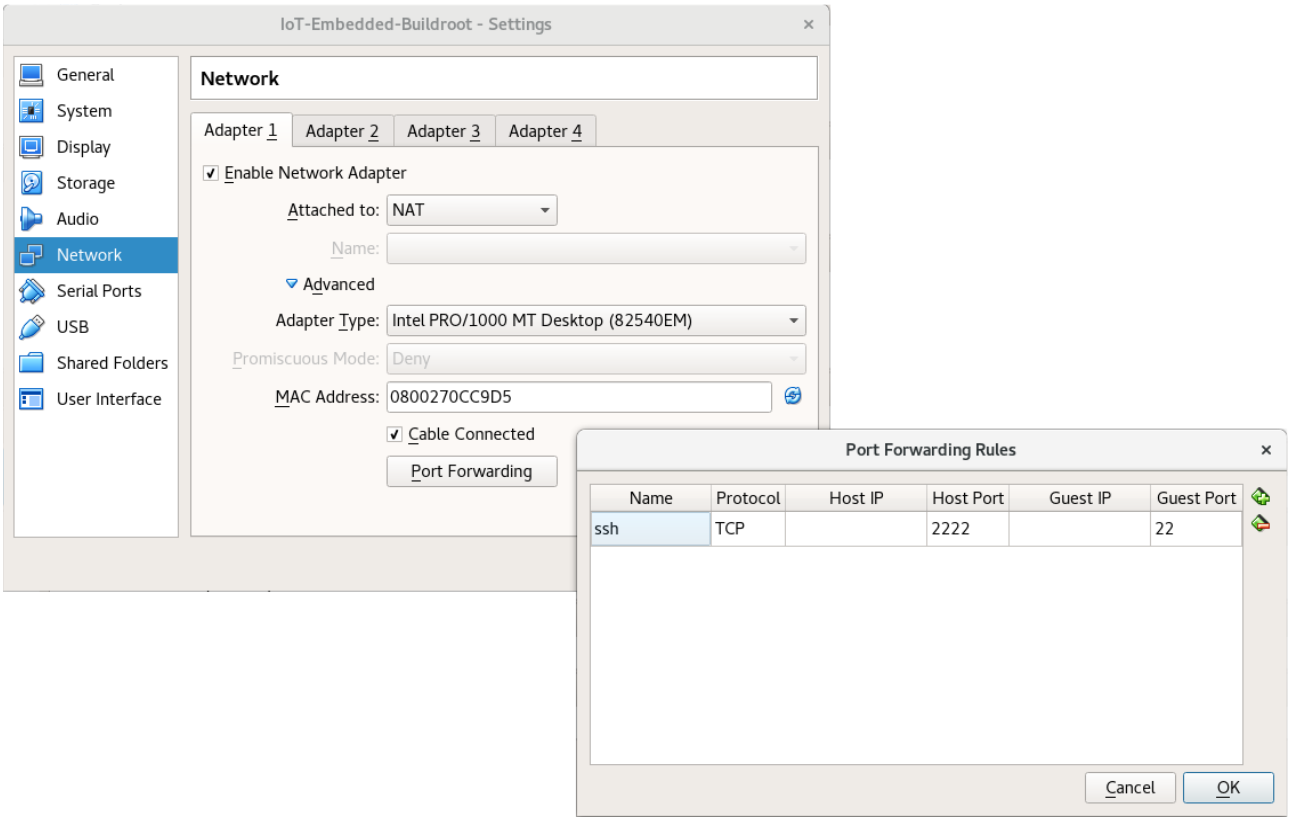

Abb. 2: Portweiterleitung des lokalen Ports 2222 an Port 22 der VM

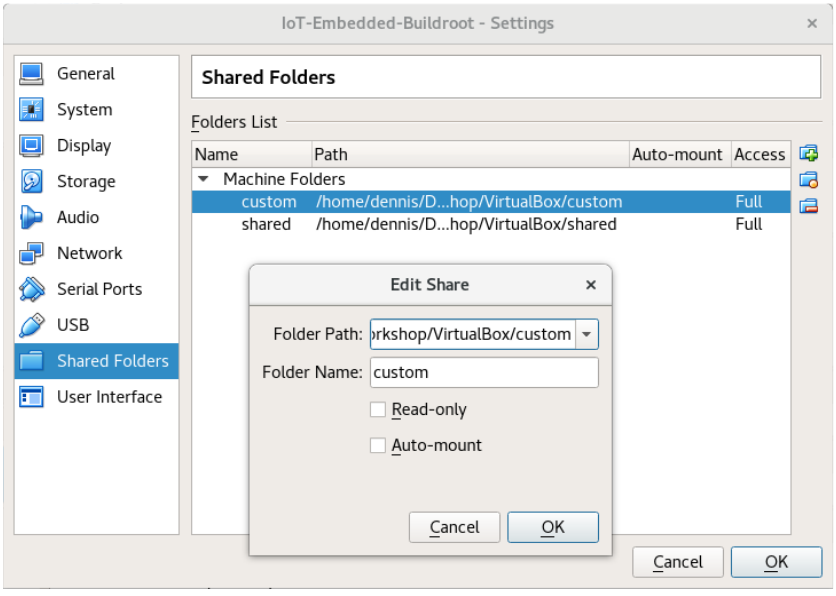

Abb. 3: Konfiguration der geteilten Verzeichnisse zwischen Host und VM

Das Fenster zum Bearbeiten der geteilten Verzeichnisse erreichen Sie, indem Sie einen der beiden Einträge doppelklicken. Sollten die Einträge fehlen, können sie mit den Icons am rechten Rand angelegt werden. Achten Sie in jedem Fall darauf, den *Folder Name* nicht zu verändern. Er muss exakt *custom* und *shared* heißen. Für *Folder Path* wählen Sie das jeweilige Verzeichnis des entpackten ZIP-Archivs auf Ihrem Computer.

Anschließend können Sie die virtuelle Maschine starten. Beim ersten Start sollten Sie dabei die Option *Normal Start* auswählen, bzw. die VM einfach doppelklicken. Sobald der SSH-Zugriff auf die VM klappt, kann sie auch im *Headless Mode* ohne das Vorschaufenster gestartet werden.

2.2 SSH-Verbindung zur virtuellen Maschine herstellen

Obwohl die virtuelle Maschine keine grafische Desktopumgebung beinhaltet, könnten Sie die VM alleine über das Vorschaufenster von VirtualBox bedienen. Die Bedienung ist dann aber nicht sonderlich komfortabel, da das Linuxsystem nur eine ganz kleine Bildschirmauflösung nutzt und es auf der Konsole auch keine geteilte Zwischenablage gibt. Beide Nachteile lassen sich mit SSH (Secure Shell) umgehen.

Falls Sie Windows nutzen, empfiehlt sich die Installation von PuTTY[4] als SSH-Client. Unter Linux und macOS sollten Sie stattdessen OpenSSH vorinstalliert haben. Beide Programmpakete bieten dieselben Funktionen, PuTTY kommt jedoch zusätzlich zur Kommandozeilenversion in einer grafischen Version mit einer einfachen Fensteroberfläche.

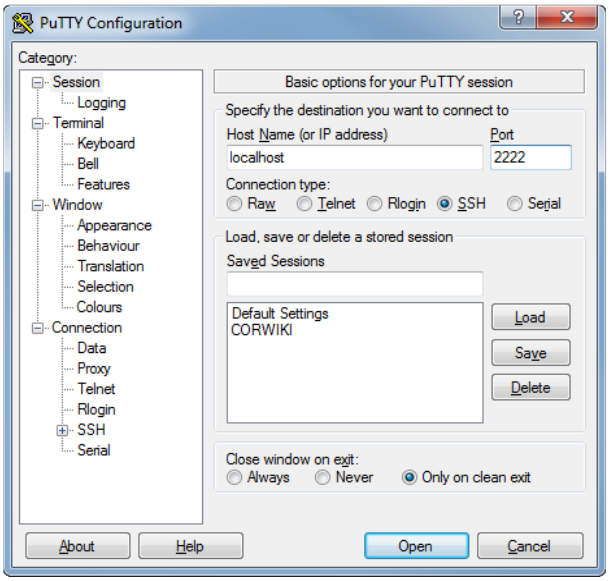

Abb. 4: Konfigurationsfenster von PuTTY unter Windows

Um sich mit der VM zu verbinden geben Sie als *Host Name* `localhost` und als Port `2222` ein. Wenn Sie wollen, können Sie die Werte mit *Save* unten in der Liste speichern. Unter Linux und Mac öffnen Sie stattdessen eine Konsole und geben folgenden Befehl ein[5]:

```
$ ssh buildroot@localhost -p 2222
```

Um sich die Tipparbeit zu sparen, können Sie auch das beigelegte Skript starten, das nur aus diesem einen Befehl besteht:

```
$ ./ssh.sh
```

Folgende Logindaten werden Sie brauchen:

- **Benutzername:** buildroot
- **Passwort:** debian
- **Root-Passwort:** debian

Achtung: Den Benutzer *root* sollten Sie nur verwenden, wenn Sie Änderung am System selbst vornehmen wollen. Aus Sicherheitsgründen wurde der direkte Login für den Root User daher gesperrt. Benutzen Sie

4 http://www.putty.org

5 Dasselbe funktioniert auch mit PuTTY unter Windows. Das Kommando heißt dort aber `pssh` anstatt `ssh`.

stattdessen das Kommando `sudo`, wenn Sie Super User-Rechte benötigen. Mit `sudo bash` können Sie auch eine Konsole mit Root-Rechten starten.

Wenn alles funktioniert, sollten Sie nun als Benutzer *buildroot* angemeldet sein und eine freundliche Eingabeaufforderung vor sich sehen.

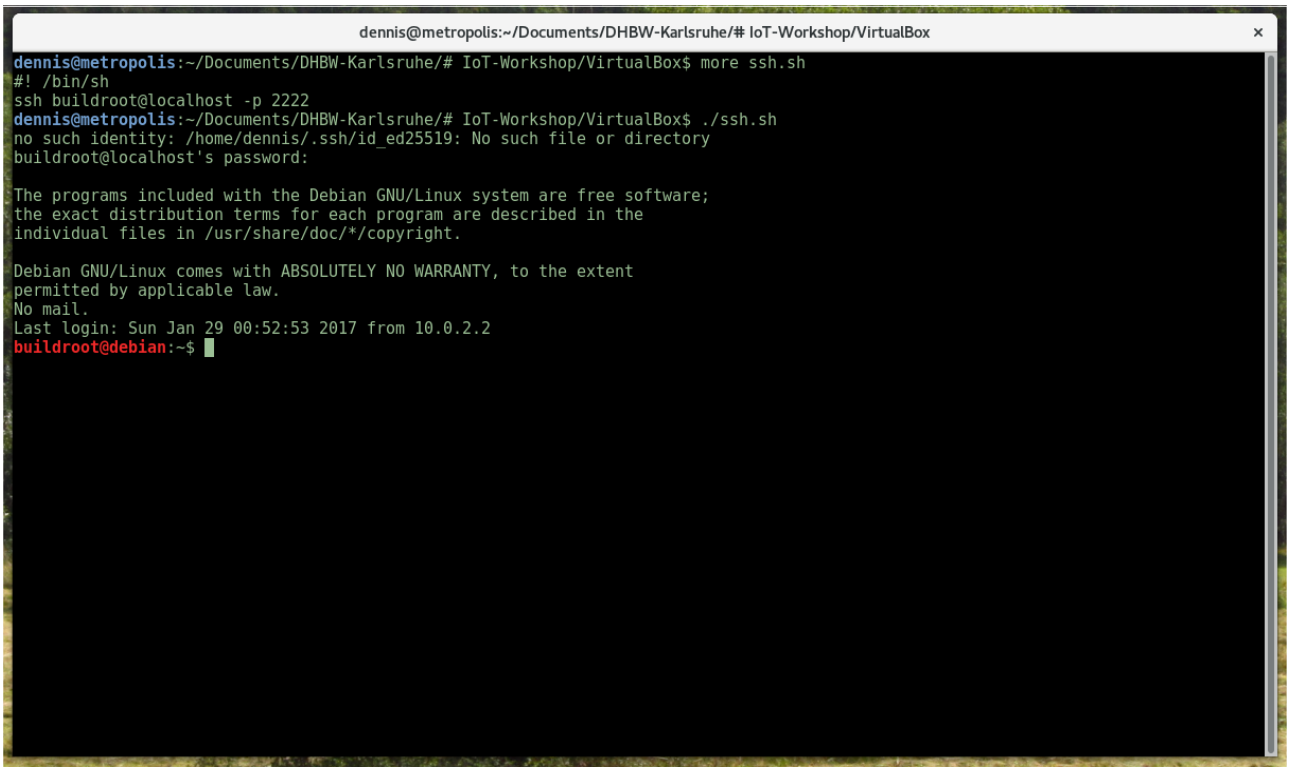

Abb. 5: SSH-Login auf der virtuellen Maschine, willkommen zuhause!

2.3 Erste Schritte mit Linux und der Konsole

Da sind Sie nun. Sie haben VirtualBox und SSH installiert und melden sich zum ersten mal in der VM an. Doch wie geht es nun weiter? Wenn Sie schon Erfahrung mit Linux haben oder öfters das Terminal von macOS nutzen, erwartet Sie hier nichts neues und Sie müssen nicht weiterlesen. Ist Ihnen der Umgang mit der Konsole jedoch neu, werden Sie sich erst noch an diese „neue“ Art zu Arbeiten gewöhnen müssen[6]. Deshalb erst mal ganz langsam, was sehen wir hier überhaupt?

```
buildroot@debian:~$
```

Das blinkende Kästchen ist der Cursor, nicht zu verwechseln mit dem Mauscursor. Er funktioniert völlig gleich, wie der Eingabecursor eines Texteditors oder einer Textverarbeitung. Im Unterschied dazu kann man aber immer nur eine Zeile bearbeiten, die nach Drücken der `[ENTER]`-Taste sofort ausgeführt wird. Auch die anderen bekannten Tasten funktionieren wie gewohnt:

`[Links] / [Rechts]`	Verschieben des Cursors
`[Pos1]`	Sprung an den Anfang der Zeile
`[Ende]`	Sprung ans Ende der Zeile
`[Backspace]`	Das Zeichen **links** vom Cursor löschen
`[Entf]`	Das Zeichen **rechts** vom Cursor löschen

6 Tatsächlich handelt es sich um die ältere Art zu arbeiten. Vgl. https://www.youtube.com/watch?v=AyuuqsGoXys

Die einzigen Tasten, die nicht wie gewohnt funktionieren, sind `[Hoch]`, `[Runter]` und `[Tab]`. Mit ihnen kann man nicht, wenn ein Befehl länger als das Fenster ist und sich daher auf mehrere Zeilen erstreckt, zwischen den Zeilen wechseln. Stattdessen blättert man mit `[Hoch]` und `[Runter]` in der Kommandohistorie, um einen älteren Befehl ihn in ggf. abgewandelter Form erneut auszuführen. Mit `[Tab]` hingegen kann der aktuelle Befehl vervollständigt werden, ohne in komplett ausschreiben zu müssen. Gerade diese Taste sollten Sie sich daher angewöhnen, häufig zu drücken.

Links des Cursors steht der sog. Prompt. Er kann viele verschiedene Informationen anzeigen und tatsächlich ist es auch Geschmackssache, wie man ihn konfiguriert[7]. Häufig anzutreffen sind aber die folgenden Informationen, die auch unser Prompt zeigt:

- Wir sind angemeldet als Benutzer *buildroot*
- An einem Rechner mit dem Namen *debian*[8]
- Und wir befinden uns in unserem Home-Verzeichnis.

Damit der Prompt nicht zu lang wird, wird das Home-Verzeichnis üblicherweise durch ~ abgekürzt. Es handelt sich dabei um das persönliche Verzeichnis des aktuellen Benutzers, in dem in der Regel niemand sonst Schreibrechte besitzt. Voll ausgeschrieben würde es in unserem Fall `/home/buildroot` heißen. Die Abkürzung mit der Tilde funktioniert dabei auch, wenn man einen Befehl eingibt. Will man einen Pfad eingeben, der sich relativ zum eigenen Home-Verzeichnis befindet, kann man den ersten Teil einfach mit ~ abkürzen. Zum Beispiel so:

```
$ cd ~/custom/board/rootfs_overlay
```

Anstelle von

```
$ cd /home/buildroot/custom/board/rootfs_overlay
```

Nach diesem Befehl sieht der Prompt wie folgt aus, da Sie das Arbeitsverzeichnis gewechselt haben:

```
buildroot@debian:~/custom/board/rootfs_overlay$
```

Mit folgendem Befehl kehren Sie wieder zurück:

```
$ cd ~
```

Stattdessen hätten Sie auch mehrmals `cd ..` oder einmal `cd ../../..` eingeben können, um sich schrittweise wieder rückwärts zu bewegen. Fehlt Ihnen übrigens einmal die Anzeige des aktuellen Verzeichnisses, zum Beispiel innerhalb der Embedded Firmware, wenn der Prompt nicht konfiguriert wurde, können Sie dieses mit dem Befehl `pwd` anzeigen lassen[9].

Exkurs: Dateisystempfade in Windows, Linux und macOS

Von Windows her kennen Sie die Laufwerksbuchstaben und Pfade wie `C:\users\elwood\Desktop`. Jedes Laufwerk oder besser gesagt jede Partition auf einem Laufwerk besitzt dabei einen eigenen Buchstaben, mit dem ein Pfad beginnt. Dieses Konzept und auch die Tatsache, dass die erste Festplattenpartition immer den Buchstaben C trägt, hat Windows von seinem Vorgängerbetriebssystem MS-DOS und dieses wiederum von seinem Vorgängerbetriebssystem CP/M aus den 1970er-Jahren geerbt. Seinerzeit war es üblich, ein oder zwei Diskettenlaufwerke im Computer zu haben, die jeweils als Laufwerk A und B bezeichnet wurden. Umgekehrt hatte aber nicht jeder Computer eine Festplatte …

Linux hat seine Konvention hingegen von Unix ebenfalls aus den 1970er-Jahren geerbt. Im Gegensatz zu

7 Die Konfiguration können Sie mit `echo $PS1` anzeigen.
8 Der Name des Rechners wird in der Datei `/etc/hostname` festgelegt.
9 pwd = Personal Working Directory

Windows gibt es keine Laufwerksbuchstaben, sondern alle Partitionen befinden sich in einem großen Dateisystem, dessen oberstes Verzeichnis immer `/` heißt. Dasselbe Zeichen dient auch als Trennstrich zwischen den Verzeichnissen. Zum Beispiel: `/var/spool/mail`.

Laufwerke und Partitionen können an jeder beliebigen Stelle im Dateisystem eingehängt werden und tauchen dann als weiteres Verzeichnis darin auf. USB-Sticks und andere externe Medien werden zum Beispiel häufig als Unterverzeichnis von `/run/media` zum Beispiel als `/run/media/dennis/meinstick` eingehängt. Man kann einem Pfad daher anders als in Windows nicht ansehen, auf welchem Laufwerk er liegt.

Seit 2001 gilt dieselbe Konvention auch für Macintosh, da das damals neue macOS X auf einem Unix-Kernel namens Darwin basiert.

Auf den ersten Blick mag das alles ziemlich anachronistisch wirken. Doch tatsächlich ist die Konsole eine der größten Stärken der unixoiden Betriebssysteme. Hierfür muss man wissen, dass das ursprüngliche Unix, von dem auch Linux im weitesten Sinne abstammt, aber auch Linux selbst von Programmierern für Programmierer entworfen wurde. Nicht nur, dass wirklich jeder Bestandteil des Systems ausgetauscht werden kann (Sie möchten eine andere Konsole?[10] Sie möchten Systemmeldungen anders protokollieren? Sie möchten eine andere Desktopumgebung? Einen anderen Display Server? … Kein Problem, bitte sehr!), das System selbst ist auch hochgradig programmierbar. Tatsächlich bietet die Konsole daher viele Konstrukte wie Variablen, Unterprogramme, if-Abfragen, while-Schlifen usw., die man sonst eher in einer Programmiersprache erwartet. Sie können die Befehle daher auch als kleines Programm in einer Datei speichern[11] und diese Datei wie jedes andere Programm ausführen.

Ziel dieses Dokuments ist es nicht, alle Feinheiten der Linux-Kommandozeile zu vermitteln. Stattdessen empfehle ich Ihnen, das Tutorial *Learning The Shell* auf www.linuxcommand.org durchzuarbeiten. Es bietet einen leicht verständlichen Einstieg und kann an einem Nachmittag durchgearbeitet werden. Danach kennen Sie praktisch alle für den Alltag relevanten Befehle. Mit Blick auf das Projekt kann es zusätzlich noch sinnvoll sein, das Tutorial *Writing Shell Scripts* zu bearbeiten, da Sie unter Umständen eigene kleine Scripts schreiben müssen, um bestimmte Dinge in Ihrer Firmware zu automatisieren.

☞ Nehmen Sie sich die Zeit für die beiden Tutorials ☜

Die nachfolgende Liste gibt einen kurzen Überblick über die wichtigsten Befehle[12].

Erste Hilfe

`ls --help`	Zeigt die Hilfeseite des `ls`-Befehls an
`man ls`	Zeigt das Handbuch zum Befehl `ls` an
`apropos echo`	Sucht nach allen Handbuchseiten zum Begriff `echo`

Mit Verzeichnissen arbeiten

`cd ~`	Wechsel in das persönliche Home-Verzeichnis
`cd buildroot/docs`	Wechsel in das Unterverzeichnis `buildroot/docs`
`cd ..`	Wechsel in das übergeordnete Elternverzeichnis
`cd ../..`	Wechsel in das Elternverzeichnis des Elternverzeichnisses
`pwd`	Zeigt den Pfad des aktuellen Arbeitsverzeichnisses an
`mkdir sourcecode`	Legt das Unterverzeichnisses `sourcecode` an
`mkdir -p docs/example/config`	Legt eine gesamte Verzeichniskette an
`rm -r backup`	Löscht das Verzeichnis `backup` mit allen Unterverzeichnissen (**Achtung:** Es gibt auf der Konsole keinen Papierkorb. Weg ist

10 Wir nutzen die BASH, die u.a. die von POSIX vorgeschriebenen Befehle und Syntax versteht
11 Ein sog. Shell Script
12 Zum schnellen Nachschlagen ist auch die O`Reilly Linux Quick Reference nicht schlecht: http://www.linuxdevcenter.com/excerpt/LinuxPG_quickref/linux.pdf

	weg!)
Mit Dateien arbeiten	
`ls`	Zeigt den Inhalt des aktuellen Verzeichnisses im Kurzformat an
`ls -lh`	Zeigt den Inhalt des aktuellen Verzeichnisses im Langformat an
`ls -alh`	Zeigt auch versteckte Datei (beginnend mit einem Punkt)
`touch Makefile`	Legt die Datei `Makefile` an oder aktualisiert Datum und Uhrzeit der letzten Änderung, falls sie schon existiert
`cp README README.ALT`	Kopiert die Datei `README` auf `README.ALT`
`cp README doc/`	Kopiert die Datei `README` in das `doc`-Verzeichnis
`mv README LIESMICH`	Benennt die Datei `README` in `LIESMICH` um
`rm todo.txt`	Löscht die Datei `todo.txt`
`file a.out`	Ermittelt den Dateityp von `a.out`. (Dateiendungen haben unter Linux keine Bedeutung und sind nur für den Anwender da)
`more COPYING`	Zeigt den Inhalt der Textdatei `COPYING` an
`less COPYING`	Wie `more` nur wesentlich komfortabler
`hexdump -C sdcard.img`	Zeigt den Inhalt der Binärdatei `sdcard.img` an
`echo "Hallo Welt" > beispiel.txt`	Schreibt die Zeile „Hallo Welt“ in die Datei `beispiel.txt`. Falls die Datei schon existiert, wird ihr Inhalt ersetzt.
`echo "Wie geht es dir?" >> beispiel.txt`	Hängt „Wie geht es dir?“ an die Datei `beispiel.txt` an.
Programme starten und beenden	
`program`	Führt das Programm `program` aus. Die ausführbare Datei muss systemweit in einem der `PATH`-Verzeichnisse installiert werden
`chmod +x run_server.sh`	Macht das Script `run_server.sh` ausfürhbar (nur für seinen Besitzer, `w+x` macht es für alle ausführbar)
`chmod -x run_server.sh`	Macht das Script `run_server.sh` nicht mehr ausführbar (hier analog `w-x`, um `w+x` rückgängig zu machen)
`./build_firmware`	Führt das nicht systemweit installierte Skript oder Programm `build_firmware` aus dem aktuellen Verzeichnis aus
`program > output.txt`	Leitet die Ausgabe des Programms in die Datei `output.txt` um.
`program > /dev/null`	Verwirft alle Ausgaben das Programms ohne sie anzuzeigen.
`program &`	Führt das Programm im Hintergrund aus, so dass im Vordergrund mit der Konsole weitergearbeitet werden kann
`fg`	Holt das letzte Hintergrundprogramm in den Vordergrund
`ps`	Zeigt alle vom aktuellen Benutzer gestarteten Prozesse
`ps ax`	Zeigt eine Liste aller aktiven Prozesse inklusive ihrer Prozess-IDs
`ps ax | grep nano`	Sucht nach dem
`pstree`	Zeigt die Prozessliste in Baumform, so dass man erkennt, welcher Prozess welchen anderen Prozess gestartet hat
`kill 2409`	Sendet das `SIGTERM`-Signal an den Prozess mit der ID 2409 und fordert ihn damit auf, sich zu beenden
`kill -9 2409`	Schießt den Prozess 2409 hart ab (`SIGKILL`-Signal)
`killall java`	Beendet alle Prozesse mit dem Namen `java`
Umgebungsvariablen setzen und lesen	
`VARIABLE=Wert`	Speichert das Wort `Wert` in einer Umgebungsvariable (**Achtung:** Keine Leerzeichen vor oder nach dem `=` erlaubt). Alle Variablen sind stets Strings, nummerische Datentypen gibt es nicht.
`VARIABLE="Ein langer String"`	Speichert einen String mit Leerzeichen in einer Variable
`VARIABLE="Hallo, $USER"`	Beispiel für die Expansion von Variablen. Die Variable `$USER` wird aufgelöst und das Ergebnis „Hallo, buildroot“ gespeichert.
`VARIABLE=$(ls)`	Führt den Befehl `ls` aus und speichert seine Ausgabe als Variable

`VARIABLE=`ls``	Traditionelle Syntax für das Beispiel zuvor (sog. Backticks)
`echo $HOME`	Zeigt den Inhalt der Variable `$HOME` an
`echo "Du bist $USER und wohnst in $HOME"`	In Wirklichkeit zeigt `echo` nur den Wert an, den man ihm übergibt
`export`	Zeigt eine Liste aller Umgebungsvariablen und ihrer Werte
`export VARIBALE=Wert`	Legt eine neue Variable an und exportiert sie. Dadurch wird sie an Kindprozesse weitergereicht. Startet man also ein Programm oder Skript, kann dieses die Variable nun sehen.
unexport VARIABLE	Macht den Export einer Variable wieder rückgängig
`VAR1=Wert VAR2=Wert program`	Setzt zwei Umgebungsvariablen und exportiert sie für das in der selben Zeile ausgeführte Programm. Die Variablen sind nur für das Programm sichtbar und verschwinden nach dessen Ende wieder.

2.4 Textdateien bearbeiten auf der Konsole

Manchmal möchten man nur schnell eine kleine Datei auf der Konsole ändern, um zum Beispiel eine Konfiguration anzupassen, oder einen Quellcodefehler zu korrigieren. Hierfür ist in der VM *nano* installiert, dessen Bedienung sich recht schnell lernen lässt. Der Aufruf erfolgt wie folgt:

`nano`	Anlegen einer neuen Datei ohne Namen
`nano` *`dateiname`*	Öffnen einer vorhandenen Datei oder Anlegen einer neuen

Wollten Sie zum Beispiel den Inhalt des Installationsskripts modifizieren, könnten Sie den Editor wie folgt aufrufen und würden folgendes Bild sehen:

```
$ nano install.sh
```

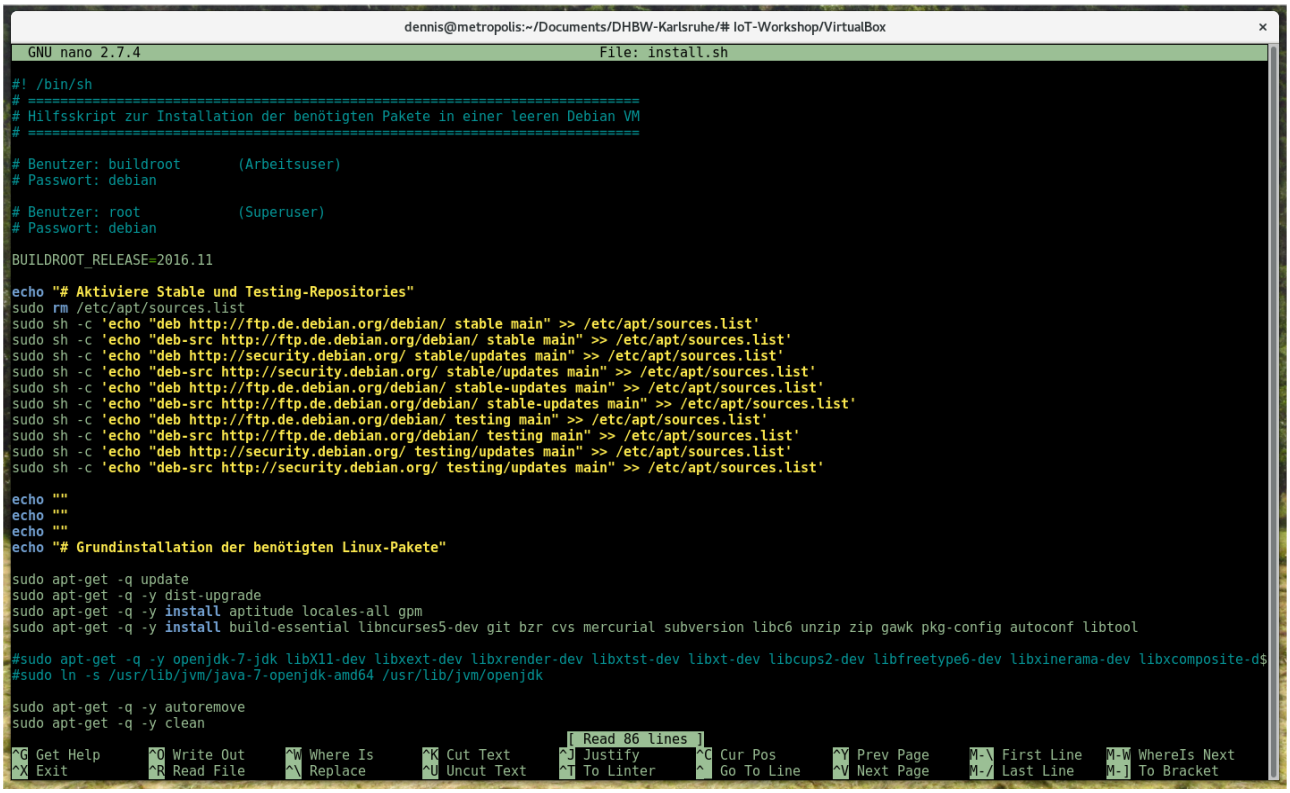

Abb. 6: Ein Shell Script in nano

Die unteren beiden Zeilen zeigen die wichtigsten Shortcuts. `^` steht dabei gemäß Emacs-Tradition für die `[Strg]`-Taste und `M` für `[Meta]`, die auf modernen Rechnern `[Alt]` heißt. Die meiste Zeit kommen Sie aber schon mit folgenden einfachen Tastenkürzeln aus:

`[Strg]+[X]`	Programm beenden, bei ungesicherten Änderungen erscheint am unteren Bildschirmrand eine Fragen, ob Sie sichern wollen
`[Strg]+[O]`	Datei speichern (Write Out)
`[Strg]+[R]`	Datei öffnen (Read In)
`[Strg]+[W]`	Suche nach einem bestimmten String
`[Strg]+[K]`	Ausschneiden der aktuellen Zeile, um sie in der internen Zwischenablage zu sichern. Mehrfach hintereinander gedrückt, wird jede Zeile der Zwischenablage hinzugefügt.
`[Strg]+[U]`	Einfügen aller Zeilen aus der internen Zwischenablage

Auch kleine Programme lassen sich problemlos mit *nano* schreiben, da es für alle wesentlichen Sprachen Syntax Highlighting unterstützt. Für größere Sachen empfiehlt es sich hingegen, VIM oder Emacs zu installieren. Die Lernkurve ist aber bei beiden deutlich steiler[13].

2.5 Eigene Skripte schreiben

Die Linux-Konsole hat eine vollwertige Programmiersprache eingebaut, durch die sich wiederkehrende Aufgaben (mehr oder weniger komfortabel[14]) automatisieren lassen. Hierfür können alle Befehle, die man sonst von Hand eintippen würde, in eine Textdatei (das sog. Skript) geschrieben und diese anschließend ausgeführt werden. Hierfür muss die Datei allerdings zwei wichtige Voraussetzungen erfüllen:

1. Die erste Zeile der Datei lautet: `#! /bin/sh`
2. Die Datei muss mit `chmod +x` *`dateiname`* ausführbar gemacht werden.

Bei der ersten Zeile handelt es sich um den sog. *Shebang*. Linux erkennt die Zeichenfolge `#!` am Anfang der Datei und führt die Datei mit dem dort aufgeführten Programm aus. In unserem Fall also `/bin/sh`. Wurde die Datei mit `chmod +x` ausführbar gemacht[15], kann sie dann mit folgendem Befehl gestartet werden:

```
$ ./dateiname
```

Das ganze ist dabei lediglich eine Abkürzung für folgenden Befehl, der auch dann funktioniert, wenn die Datei das *Executable Flag* nicht besitzt.

```
$ /bin/sh dateiname
```

Der Präfix `./` bezieht sich dabei auf das aktuelle Arbeitsverzeichnis. Liegt das Skript stattdessen in einem der offiziellen Verzeichnisse für ausführbare Dateien, kann es ohne den vorangestellten Verzeichnispfad ausgeführt werden. Welche Verzeichnisse das sind, können Sie dabei durch Abfragen der `$PATH`-Umgebungsvariable herausfinden:

```
buildroot@debian:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

Ein einfaches Skript, das den Aufruf von Buildroot automatisiert, könnte daher zum Beispiel so aussehen. Hier sehen Sie auch, dass Kommentare immer mit `#` beginnen:

```
#! /bin/sh
set -e                                  # Beim ersten Fehler das Skript beenden
cd ~/make                               # In das ~/make-Verzeichnis wechseln
make BR2_EXTERNAL=../custom all         # Neues Firmware Image bauen
cp images/sdcard.img ../shared          # Das Image in das ../shared-Verzeichnis kopieren
```

Für komplexere Abläufe gibt es auch Bedingungen, Schleifen, Variablen und Funktionen, ähnlich wie man es von einer richtigen Programmiersprache erwarten würde. Diese sind jedoch manchmal etwas schwierig zu verwenden und verhalten sich nicht immer so, wie von neueren Sprachen gewohnt. An dieser Stelle daher nur ein kurzer Auszug häufig benötigter Code-Schnippsel:

13 Es könnte allerdings passieren, dass Sie hinterher nie mehr mit einem anderen Editor arbeiten wollen. :-)
14 Die Sytnax ist manchmal etwas eigenwillig.
15 Durch chmod +x wird im Dateisystem das *Executable Flag* gesetzt.

```
# Definition und Verwendung von Umgebungsvariablen
INFILE=MeinProgramm.java
OUTFILE="backup/$INFILE"
PATH="$PATH:~/bin"

# Prüfen, ob eine Datei existiert
if [ -e dateiname ]; then
    echo "Die Datei ist vorhanden"
else
    echo "Die Datei existiert nicht"
fi

# Prüfen, ob eine Datei nicht existiert
if [ ! -e "$OUTFILE" ]; then
    echo "Die Datei ist noch nicht vorhanden"
fi

# Art einer Datei abfragen
if [ -f dateiname ]; then
    echo "Es ist eine normale Datei"
elif [ -d dateiname ]; then
    echo "Es ist ein Verzeichnis"
elif [ -L dateiname ]; then
    echo "Es ist ein symbolischer Link (Verweis)"
fi

# Eigenschaften einer Datei prüfen
if [ -r dateiname ]; then
    echo "Ich darf die Datei lesen"
fi

if [ -w dateiname ]; then
    echo "Ich darf in die Datei schreiben"
fi

if [ -x dateiname ]; then
    echo "Ich darf die Datei ausführen"
fi

# Schleife über eine Liste von Dateien
for filename in *.txt; do
    echo "Textdatei: $filename"
done

# Ein einfaches Auswahlmenü bauen
auswahl=

until [ "$auswahl" = "E" ]; do
    echo <<EOF
    HAUPTMENÜ

    [L] Liste aller Textdateien anzeigen
    [E] Programm beenden

EOF

    echo -n "Auswahl: "
    read auswahl
    echo

    case "$auswahl" in
        L|l)
            ls -lh *.txt
        E|e)
            echo "Auf wiedersehen …"
            echo
        *)
            echo "Unbekannte Auswahl!"
            echo
    esac
done
```

```
# Definition und Aufruf einer Funktion
ReadmeDateiAnzeigen () {
    if [ -e README ]; then
        less README
    fi
}

ReadmeDateiAnzeigen
```

2.6 Herunterfahren der virtuellen Maschine

Um die virtuelle Maschine zu beenden, kann ihr VirtualBox ein *ACPI Shutdown* ausgelöst werden. Dies entspricht dem Drücken des Stromschalters, wodurch die Maschine herunterfährt. Alternativ können Sie die VM mit folgendem Befehl beenden (innerhalb der Konsole der VM):

```
$ sudo poweroff
```

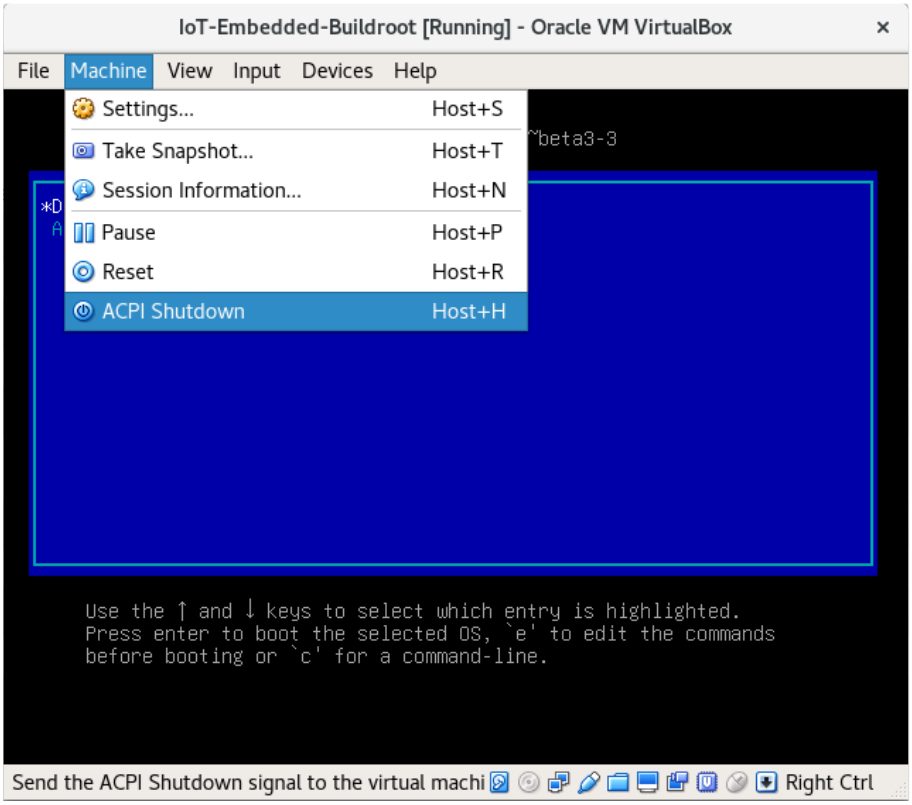

Abb. 7: Virtuelles Ausschalten der Maschine

2.7 Installation weiterer Pakete und Upgrade des Systems

Innerhalb der virtuellen Maschine sind bereits alle Programme installiert, die Sie für die Erstellung einer Firmware mit Buildroot benötigen. Die Installation wurde dabei mit dem im Home-Verzeichnis liegenden Skript `install.sh` durchgeführt. Sollten Sie noch weitere Pakete installieren wollen, können Sie folgende Befehle hierfür nutzen, wobei sie jeweils noch `sudo` voranstellen müssen, falls Sie sich nicht in einer Root-Konsole befinden.

`apt-get update`	Aktualisieren der Paketinformationen
`apt-get dist-upgrade`	Voller Upgrade aller installierter Pakete
`apt-get install vim`	Installation des Pakets „vim“ inklusive aller Abhängigkeiten und vorgeschlagenen Zusatzpakete

`apt-get --no-install-recommends install vim`	Installation des Pakets „vim“ inklusive alle Abhängigkeiten aber ohne die vorgeschlagenen Zusatzpakete
`apt-get remove vim`	Deinstallation des Pakets „vim“
`apt-get autoremove`	Deinstallation aller nicht mehr benötigter, automatisch installierter Pakete
`aptitude search vim`	Suche nach der Zeichenkette „vim“ im Namen und der Kurzbeschreibung aller verfügbaren Pakete
`aptitude show vim`	Anzeige aller Informationen zum Paket „vim“

Mit Aptitude steht auch eine „grafische“ Version mit Menüoberfläche zur Verfügung. Wenn Sie die VM auf einem Linux Host laufen lassen, können Sie die Oberfläche sogar mit der Maus bedienen. Mit der Tastatur geht es aber wesentlich schneller.

`[Strg]+[T]`	Menü öffnen
`[/]`	Suche nach einem Paket (Regex)
`[+]`	Markiertes Paket zur Installation vormerken
`[-]`	Markiertes Paket zur Deinstallation vormerken oder Installationsvormerkung aufheben
`[g]`	Installation starten (2x drücken)
`[q]`	Zurück zum vorherigen Bildschirm oder Programm beenden

Falls Sie Hilfe zu einem der Programme benötigten, können Sie diese mit `man` anzeigen[16]. Der Folgende Befehl zeigt zum Beispiel die Man Page zu *apt-get*.

```
$ man apt-get
```

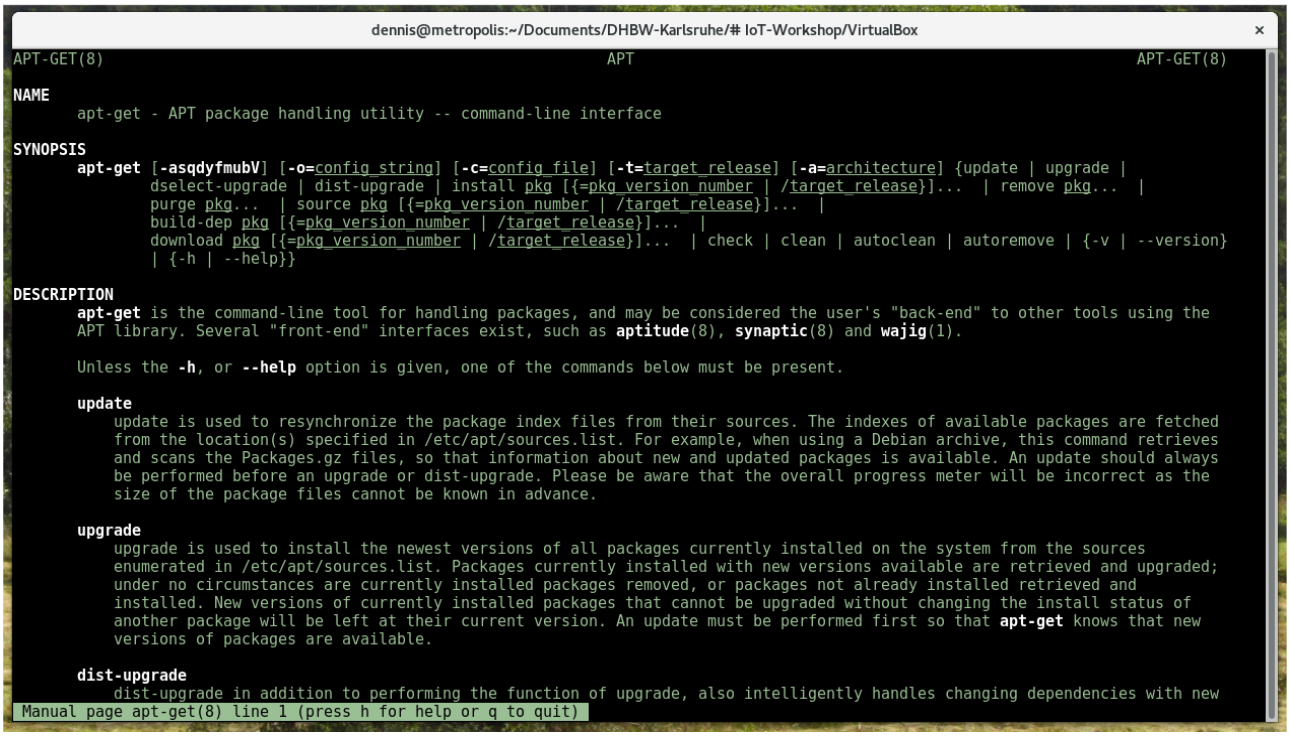

Abb. 8: Man Page zu apt-get

2.8 Geteilte Verzeichnisse wieder zum Laufen bringen

Nach einem Upgrade der virtuellen Maschine kann es vorkommen, dass die geteilten Verzeichnisse nicht mehr funktionieren. Das kann entweder passieren, wenn ein neuerer Linux-Kernel installiert oder sich die

16 man = Manual, Speicherplatz war früher extrem teuer …

VirtualBox-Version verändert hat. In beiden Fällen müssen die aktuellen *VirtualBox Guest Additions* in der VM installiert und eine neue Version des dazugehörigen Kernel-Moduls kompiliert werden. Zur Vereinfachung dieser Aufgabe liegt in der VM das Skript `reinstall-vbox-additions.sh` bereit.

Falls sich weder der Linux-Kernel noch VirtualBox verändert haben, reicht es oft schon aus, einfach das Skript `/etc/rc.local` nochmal auszuführen:

```
$ sudo /etc/rc.local
```

Sollte dies nicht helfen, müssen Sie doch die VirtualBox Guest Additions neuinstallieren. Starten Sie hierfür das Skript `reinstall-vbox-addtions.sh` mit folgendem Befehl. Es führt Sie durch die Installation:

```
$ ~/reinstall-vbox-additions.sh
```

```
# Bereite Installation der VirtualBox Guest Additions vor
Reading package lists...
Building dependency tree...
Reading state information...
module-assistant is already the newest version (0.11.9).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
Getting source for kernel version: 4.9.0-1-amd64
Kernel headers available in /lib/modules/4.9.0-1-amd64/build
apt-get install build-essential
Reading package lists... Done
Building dependency tree
Reading state information... Done
build-essential is already the newest version (12.2).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.

Done!


Nun in VirtualBox auf "Devices" --> "Insert Guest Additions" klicken.
Anschließend ENTER um fortzufahren
```

Öffnen Sie die VM in VirtualBox, sobald Sie das Skript dazu auffordert und wählen Sie den Menüeintrag *Devices → Insert Guest Additions CD image…* aus. Anschließend wechseln Sie wieder auf die Konsole und drücken `[ENTER]`, um das Skript fortzufahren.

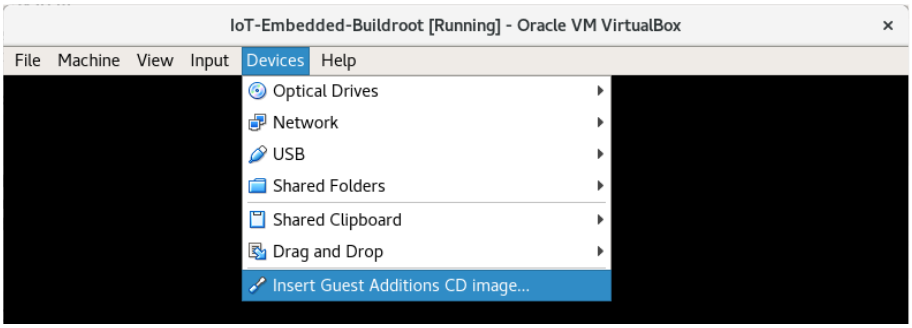

Abb. 9: Einlegen der virtuellen CD mit den VirtualBox Guest Additions

Anschließend beginnt die Installation. Dabei sollten folgende Zeilen ausgegeben werden. Ist die Ausgabe kürzer, ist etwas schief gelaufen. In diesem Fall brechen Sie das Skript mit [Strg]+[C] ab und starten die Installation erneut.

```
mount: /dev/sr0 is write-protected, mounting read-only
Verifying archive integrity... All good.
Uncompressing VirtualBox 5.1.12 Guest Additions for Linux..........
VirtualBox Guest Additions installer
Removing installed version 5.1.12 of VirtualBox Guest Additions...
Copying additional installer modules ...
Installing additional modules ...
vboxadd.sh: Building Guest Additions kernel modules.
```

```
vboxadd.sh: You should restart your guest to make sure the new modules are actually used.
vboxadd.sh: Starting the VirtualBox Guest Additions.

Could not find the X.Org or XFree86 Window System, skipping.


Die CD mit den Guest Additions kann nun in VirtualBox ausgeworfen werden.
Anschließend ENTER um fortzufahren
```

Sobald Sie das Skript auffordert, können Sie die virtuelle CD wieder entfernen.

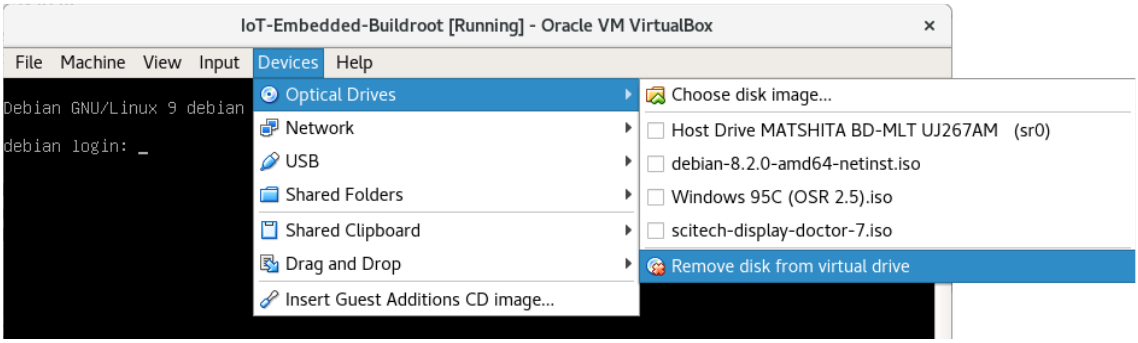

Abb. 10: Auswerfen der virtuellen CD

Nachdem das Skript zu Ende gelaufen ist, sollten in den Verzeichnissen `custom` und `shared` wieder die geteilten Inhalte sichtbar sein. Dies können Sie mit einem einfachen `ls custom` überprüfen.

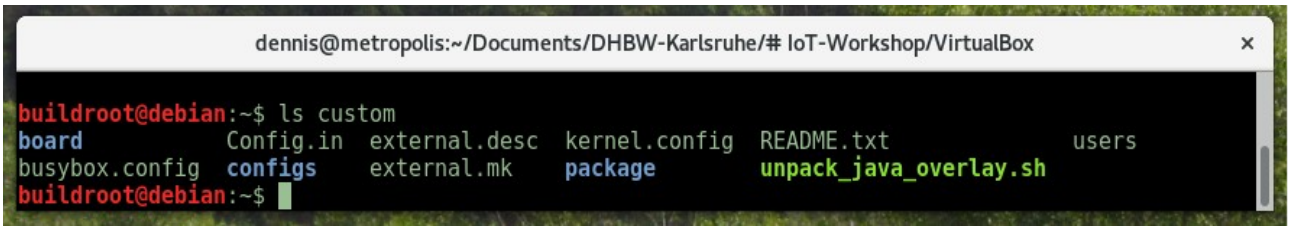

Abb. 11: Juhuu, die geteilten Verzeichnisse sind wieder da

3 Firmware bauen mit Buildroot

3.1 Wir erkunden Buildroot

Wenn Sie sich bereits ein wenig im Home-Verzeichnis umgeschaut haben, sind Ihnen sicher die folgenden Verzeichnisse aufgefallen:

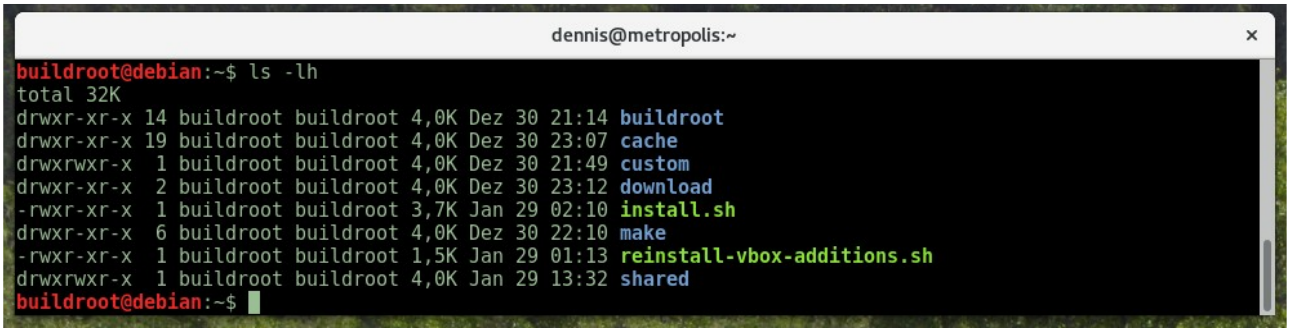

Abb. 12: Inhalt des Home-Verzeichnisses. Die blauen Einträge sind Verzeichnisse, die grünen ausführbare Dateien.

Buildroot-spezifische Verzeichnisse

`buildroot`	Quellcodes von Buildroot selbst. Hierin sollten Sie nichts ändern. Sie können aber mit `git` eine neuere Version herunterladen.
`custom`	Aus dem Buildroot-Verzeichnis ausgelagertes Customizing der zu erstellenden Firmware. Ordner wird mit dem Host-Rechner geteilt.
`make`	Aus dem Buildroot-Verzeichnis ausgelagertes Build-Verzeichnis. Von hier aus müssen alle Buildroot-Befehle ausgeführt werden und hier werden auch alle beim Build erzeugen Dateien abgelegt.
`shared`	Mit dem Host geteiltes Verzeichnis, in dem Sie die fertigen Firmware-Images ablegen können.

Temporäre Verzeichnisse

`download`	Hier legt Buildroot alle heruntergeladenen Quellcodes ab.
`cache`	Compiler Cache zum Beschleunigen nachfolgender Builds.

Die beiden Verzeichnisse `download` und `cache` haben dabei keine tiefere Bedeutung. Sie dienen nur der Ablage temporärer Dateien, um nachfolgende Builds zu beschleunigen. Dennoch sollten Sie die Verzeichnisse nicht anrühren, da der nächste Build sonst unter Umständen mehrere Stunden dauert. Auch das `buildroot`-Verzeichnis werden Sie nur selten benötigen, da Buildroot entsprechend den Empfehlungen des Handbuchs in der VM bereits für einen Out-Of-Tree Build konfiguriert wurde (vgl. nachfolgendes Kapitel). Sowohl die Konfiguration der Firmware als auch die dabei anfallenden Dateien werden daher in den beiden Verzeichnissen `custom` und `make` verwaltet.

Das Besondere an Buildroot ist, dass es eigentlich nur umfangreiche Sammlung von Makefiles ist, mit denen das Cross Compiling und die Zusammenstellung der Firmware automatisiert wird. Sämtliche Aktionen werden daher über den `make`-Befehl gesteuert. Und da dieser in dem Verzeichnis aufgerufen werden muss, in dem die Firmware erstellt wird, wurde dieses Verzeichnis entsprechend ebenfalls `make` genannt. Führen Sie daher folgende Anweisungen aus, um in das make-Verzeichnis zu wechseln und eine initiale Standardkonfiguration für den Raspberry Pi zu laden:

```
$ cd ~/make
```

```
$ make BR2_EXTERNAL=../custom list-defconfigs
```

```
$ make dhbw_minimal_defconfig
```

Eine Übersicht aller möglichen Aktionen können Sie sich übrigens mit `make help` anzeigen. Darüber hinaus sollten Sie das Buildroot-Handbuch[17] stets griffbereit haben. Auf den ersten Blick mag es einen zwar sicher etwas erschlagen, tatsächlich ist aber jede Option ausführlich dokumentiert.

Nach Ausführung der obigen Befehle sollten Sie in etwa folgende Ausgabe sehen. Die mittlere Anweisung dient dabei nur der Aufzählung aller vorhandenen Standardkonfigurationen sowie dazu, Buildroot mit Hilfe des Zusatzes `BR2_EXTERNAL=../custom` für alle nachfolgenden Aktivitäten auf die externe Konfiguration im `custom`-Verzeichnis hinzuweisen. Erst dadurch werden die beiden DHBW-Standardkonfigurationen am Ende der Liste überhaupt sichtbar.

```
buildroot@debian:~$ cd make
buildroot@debian:~/make$ make BR2_EXTERNAL=../custom list-defconfigs
umask 0022 && make -C /home/buildroot/buildroot O=/home/buildroot/make/. list-defconfigs
Built-in configs:
  …

External configs in "DHBW-Spezifische Anpassungen":
  dhbw_java_defconfig                - Build for dhbw_java
  dhbw_minimal_defconfig             - Build for dhbw_minimal

buildroot@debian:~/make$ make dhbw_minimal_defconfig
umask 0022 && make -C /home/buildroot/buildroot O=/home/buildroot/make/. dhbw_minimal_defconfig
  GEN     /home/buildroot/make/Makefile
#
# configuration written to /home/buildroot/make/.config
#
buildroot@debian:~/make$
```

Zum Bearbeiten der Konfiguration steht ein grafisches Konfigurationswerkzeug zur Verfügung, das mit folgender Anweisung aufgerufen werden kann:

```
$ make menuconfig
```

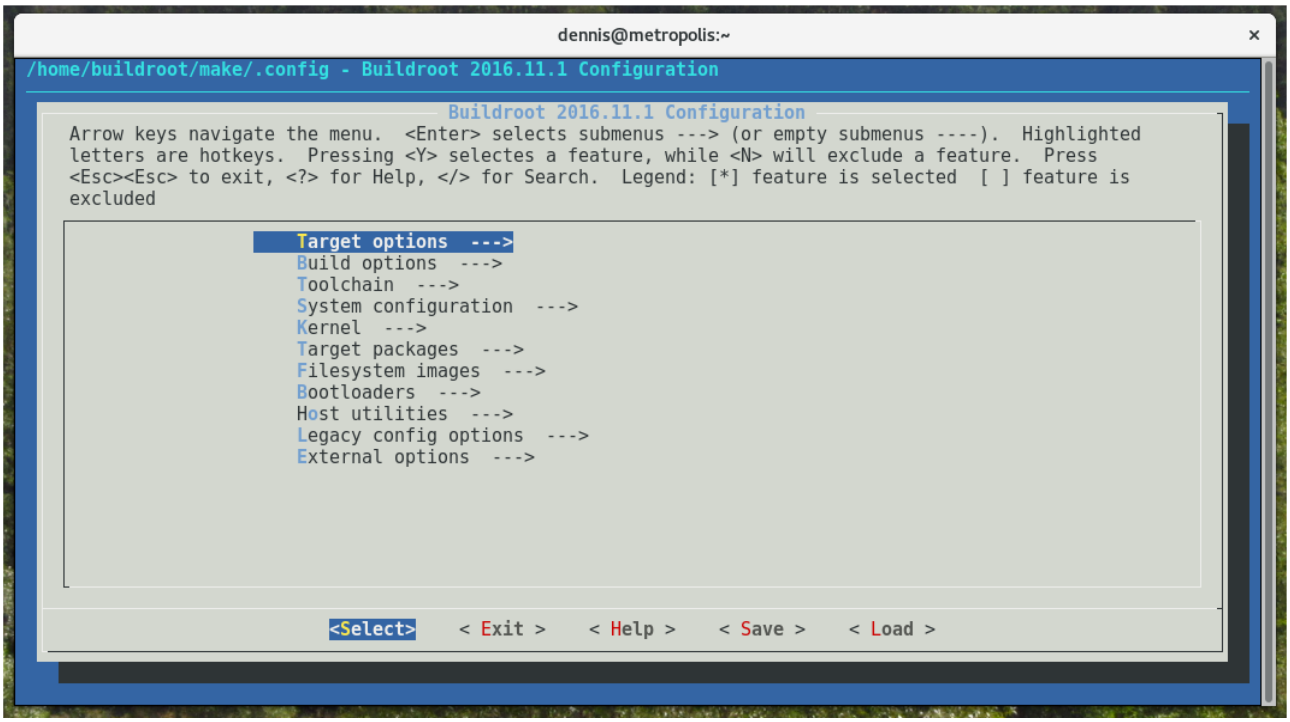

Abb. 13: Konfigurationswerkzeug von Buildroot

17 https://buildroot.org/downloads/manual/manual.html

Schauen Sie sich ruhig ein wenig in den Menüs um, um ein Gefühl für die Bedienung zu bekommen. Die einzelnen Ordner haben dabei folgende Bedeutung. Für Sie überwiegend relevant sind die unten rot markierten Einträge. Die anderen sind bereits für den Raspberry Pi passend vorkonfiguriert.

Target Options	Auswahl der CPU-Familie und Eigenschaften des Zielboards
Build Options	Compiler-Optionen für den Bau der Firmware. Hier müssen Sie eigentlich nur den Pfad ändern, wo die Buildroot-Konfiguration gespeichert wird, wenn Sie `make savedefconfig` ausführen (*Location to save buildroot config*). Dieser zeigt Anfangs auf die zuvor geladene Standardkonfiguration, die Sie jedoch nicht überschreiben sollten.
Toolchain	Spezielle Einstellungen der verwendeten Cross Compiling Toolchain. Zum Beispiel, ob C++-Anwendungen unterstützt werden (empfohlen) und welche C-Library eingesetzt wird[18].
System configuration	Allgemeine Einstellungen der zu erstellenden Firmware. Hier legen Sie zum Beispiel den Systemnamen oder die anzulegenden Systembenutzer fest. Ebenso können Sie sog. Overlays eintragen, um eigene Dateien in das Firmware-Image aufzunehmen (vgl. Kapitel 3.9).
Kernel	Versionsauswahl des Linux-Kernels mit ggf. abweichender Download-URL. Hier wird auch die Konfigurationsdatei eingetragen, die beim Bauen des Kernels verwendet wird.
Target packages	Auswahl der in der Firmware enthaltenen Bibliotheken und Programme. In diesem Ordner werden Sie viel Zeit verbringen. :-)
Filesystem images	Hier legen Sie fest, welches Dateisystem die Firmware nutzt und ob dieses getrennt vom Kernel oder als im Kernel enthaltene *Initial RAM Disk* abgelegt wird. Die erste Variante hat den Vorteil, dass das Dateisystem später in einer eigenen Partition auf der SD-Karte enthalten ist. Die zweite Variante hat den Vorteil, dass die gesamte Firmware nur aus einer Datei besteht, was den späteren Austausch der Firmware vereinfacht. Wir nutzen die erste Variante.
Bootloaders	Konfiguration der für das Zielboard zu erstellenden Bootloader. Da der Raspberry Pi einen fest eingebauten Bootloader besitzt, ist hier nichts ausgewählt.
Host utilities	Für das Cross Compiling benötigte Werkzeuge, die auf dem Entwicklungssystem benötigt werden. Diese werden von Buildroot automatisch heruntergeladen und aus den Quellen gebaut. In der Regel müssen Sie hier nichts einstellen, da sich benötigten Optionen automatisch aktiviert werden.
Legacy config options	Hierhin werden alle alten Optionen verschoben, die in der aktuellen Buildroot-Version nicht mehr unterstützt werden. Dadurch wird sichergestellt, dass die Optionen beim Sichern der Konfiguration nicht verloren gehen. Nach einem Upgrade sollte man daher hier reinschauen und eventuell vorhandene Einträge entsprechend der neuen Buildroot-Version anpassen.
External options	Externe Konfigurationsoptionen, die nicht Teil von Buildroot selbst sind. Die hier sichtbaren Einträge kann man selbst definieren, wir nutzen dies der Einfachheit halber allerdings nicht.

Nachdem Sie die Konfiguration gesichert und das Menü verlassen haben, können Sie die Firmware mit einem schlichten `make` erstellen lassen. Dabei sollten Sie in der ersten Version keine weiteren *Target packages* auswählen, damit sich der Build-Vorgang nicht unnötig zieht. Da die oben erwähnten Cache-Verzeichnisse bereits mit Inhalten gefüllt sind, sollte der erste Build dann nur ein paar Minuten dauern. Ohne die Caches oder mit zu vielen neuen Paketen kann es aber auch eine Stunde oder länger dauern.

```
buildroot@debian:~/make$ make
...
>>>   Executing post-image script board/raspberrypi3/post-image.sh
Version: Linux version 4.4.36-v7 (buildroot@debian) (gcc version 5.4.0 (Buildroot 2016.11.1) )
...
hdimage(sdcard.img): adding partition 'boot' (in MBR) from 'boot.vfat' ...
hdimage(sdcard.img): adding partition 'rootfs' (in MBR) from 'rootfs.ext4' ...
hdimage(sdcard.img): writing MBR
buildroot@debian:~/make$
```

18 Die großen Linux-Systeme basieren i.d.R. auf der glibc, für eingebettete Systeme gibt es aber schlankere Alternativen wie µlibc. Standardmäßig nutzt Buildroot die µlibc, jedoch funktioniert damit das Oracle JDK nicht. In der DHBW-Standardkonfiguration ist deshalb glibc ausgewählt.

Wenn alles fehlerfrei geklappt hat, finden Sie im Unterverzeichnis `images` das fertige Image. Dieses können Sie (immernoch ausgehend vom `make`-Verzeichnis) wie folgt in das `shared`-Verzeichnis kopieren, um es später auf eine SD-Karte zu schreiben.

```
$ cp images/sdcard.img ../shared/
```

Sollte der Build eine andere Ausgabe als oben erzeugen und auf einen Fehler laufen, kann es sein, dass Sie zunächst alle zuvor erstellten Dateien verwerfen müssen. Dies ist insbesondere immer dann wichtig, wenn Sie eine der grundlegenden Einstellungen ändern und die Programme zum Beispiel gegen neuere Systembibliotheken gelinkt oder für eine andere CPU-Architektur gebaut werden müssen. Natürlich dauert der Build dann dementsprechend länger, da die gesamte Firmware neu kompiliert wird.

```
$ make clean
```

```
$ make
```

Um das Ganze zu beschleunigen kann es daher ganz sinnvoll sein, zunächst nur das fehlerhafte Paket zu verwerfen und alle anderen Dateien beizubehalten. Läuft Buildroot beispielsweise beim Kompilieren des VLC Media Players auf einen Fehler, kann VLC wie folgt verworfen und der Build-Vorgang anschließend fortgesetzt werden:

```
$ make vlc-dirclean
```

```
$ make
```

3.2 Wie der Out-Of-Tree Build funktioniert

Wenn Sie Buildroot das erste mal ausführen, werden alle Konfigurationen und die beim Build entstehenden Dateien standardmäßig innerhalb des Buildroot-Verzeichnnises abgelegt. Auf den erstem Blick mag dies ganz sinnvoll erscheinen, da sich dadurch die Übersichtlichkeit erhöht. Auf den zweiten Blick ist das jedoch keine gute Idee, da man so nur sehr umständlich unterschiedliche Firmwares bauen kann und alle Dateien bei einer Neuinstallation von Buildroot verloren gehen. Teilweise bietet `git` hierfür zwar Abhilfe (wenn man Buildroot aus dem git-Repository klont), wesentlich einfacher ist es aber, Buildroot gleich von Anfang an von den selbsterstellten Dateien sauber zu trennen. Mit ein paar einfachen Handgriffen ist das auch gar nicht mal schwer.

Hinweis: In der VM ist Buildroot bereits für Out-Of-Tree Builds konfiguriert. Dieses Kapitel ist nur relevant, wenn Sie Buildroot neuinstallieren oder auf einem anderen Rechner einrichten wollen.

Angenommen, Sie nutzen die in der VM vorgegebene Verzeichnisstruktur und haben deshalb Buildroot nach `buildroot` entpackt, das Bauen soll aber im daneben liegenden `make`-Verzeichnis geschehen.

```
buildroot@debian:~$ ls -1
buildroot
make
custom
...
buildroot@debian:~$
```

Dann müssen Sie einmalig in das `buildroot`-Verzeichnis wechseln und `make` mit dem Parameter `O=../make` ausführen. Welches *Make Target* Sie nach `O=../make` angeben ist dabei im Prinzip egal, `help` hat allerdings den Vorteil, dass es noch keinen Kompiliervorgang anstößt.

```
$ cd ~/buildroot
```

```
$ make O=../make help
```

Von nun an werden alle von Buildroot erzeugten Dateien im `make`-Verzeichnis abgelegt und auch Buildroot selbst muss ab jetzt von dort aufgerufen werden. Dadurch kann Buildroot zu jeder Zeit ausgetauscht oder aktualisiert werden. Ggf. müssen Sie dann nur die obigen zwei Befehle nochmal ausgeführen.

Eine noch sauberere Trennung erhalten Sie, wenn Sie das Build-Verzeichnis (hier `make` genannt) vom Konfigurationsverzeichnis (hier `custom` genannt) trennen. Dadurch wird es möglich, die projektspezifischen Anpassungen ohne die unnötigen Zwischendateien des Build-Vorgangs weiterzugeben und das Build-Verzeichnis kann zu jeder Zeit verworfen und durch die obigen Befehle neu definiert werden. Führen Sie hierfür folgende Anweisungen aus:

```
$ cd ~/make
```

```
$ make BR2_EXTERNAL=../custom dhbw_minimal_defconfig
```

Der entscheidende Zusatz ist hier `BR2_EXTERNAL=../custom`, das *Make Target* spielt wieder keine Rolle. Allerdings ist die Standardkonfiguration `dhbw_minimal_defconfig` nur in `~/custom/configs` vorhanden, so dass Sie gleich sehen, ob das neue Verzeichnis wirkt. Den `BR2_EXTERNAL`-Zusatz brauchen Sie ab jetzt nicht mehr angeben. Analog zu `O=…` merkt sich Buildroot den Parameter für alle weiteren Aufrufe.

Dasselbe Endergebnis hätte man auch in einem Schritt erzielen können, weshalb dieses Vorgehen auch am Ende des Installationsskripts vorgeschlagen wird:

```
$ cd ~/buildroot
```

```
$ make O=../make BR2_EXTERNAL=../custom dhbw_minimal_config
```

An dieser Stelle noch ein Hinweis zum Konfigurationsverzeichnis: Bis Buildroot 2016.08 musste dieses Verzeichnis keine besonderen Anforderungen erfüllen. Seit der Version 2016.11 jedoch müssen darin mindestens die folgenden Dateien mit den hier gezeigten Inhalten enthalten sein:

```
buildroot@debian:~/custom$ ls Config.in external.*
Config.in  external.desc  external.mk

buildroot@debian:~/custom$ more external.desc
name: DHBW
desc: DHBW-Spezifische Anpassungen

buildroot@debian:~/custom$ more external.mk
include $(sort $(wildcard $(BR2_EXTERNAL_DHBW_PATH)/package/*/*/*.mk))

buildroot@debian:~/custom$ more Config.in
#Inhalt durch die Raute auskommentiert
#source "$BR2_EXTERNAL_DHBW_PATH/package/jdk/Config.in"

buildroot@debian:~$
```

Die Datei `Config.in` darf leer sein. Will man aber seine eigenen *Target Packages* für Buildroot erstellen,[19] müssen sie hier allerdings eingetragen werden. Der Name `$BR2_EXTERNAL_DHBW_PATH` ergibt sich dabei aus der Tatsache, dass in der Datei `external.desc` der Name auf `DHBW` festgelegt wurde. Weitere Informationen hierzu liefert das Handbuch ab Kapitel 9.[20]

3.3 Buildroot auf eine neuere Version aktualisieren

Da die Erstellung der VM mit einem gewissen Vorlauf verbunden ist, ist die darin installierte Buildroot-Version mit hoher Wahrscheinlichkeit bereits am Anfang des Projekts nicht mehr aktuell. Dank `git` kann die Version allerdings realtiv einfach angehoben werden. Alternativ können Sie damit auch auf eine ältere Version zurückgehen, wenn Sie Probleme mit der aktuellen Buildroot-Version haben.

Hinweis: Für den Anfang können Sie dieses Kapitel überspringen.

19 Der Einfachheit halber verzichten wir darauf und arbeiten nur mit Overlays.

20 https://buildroot.org/downloads/manual/manual.html#outside-br-custom sowie https://buildroot.org/downloads/manual/manual.html#customize-packages

Informieren Sie sich zunächst auf der Download-Seite[21], im Release-Archiv[22] oder im git Webfrontend[23] über die Version, die Sie gerne nutzen möchten. Angenommen Sie haben dabei die fiktive Version 2017.02 gefunden. Dann können Sie diese mit folgenden Befehlen herunterladen:

```
$ cd ~/buildroot
$ git remote set-branches --add origin 2017.02.x
$ git fetch --depth=1
$ git checkout 2017.02.x
```

Achten Sie hier unbedingt auf die exakte Schreibweise der Kommandos und des neuen Branches, da die Fehlermeldungen von `git` sonst verwirrend sein können. Mit welchem Zweig und damit mit welcher Buildroot-Version Sie gerade arbeiten, können Sie mit folgendem Kommando feststellen:

```
buildroot@debian:~/buildroot$ git branch -a
* 2017.02.x
  2016.11.x
  remotes/origin/2017.02.x
  remotes/origin/2016.11.x
```

Mit `git checkout` *branch* können Sie dabei jederzeit zwischen den Zweigen wechseln. Zur Sicherheit sollten Sie dann aber nochmal die Kommandos aus Kapitel 3.2 ausführen, um den Out-Of-Tree Build erneut zu konfigurieren:

```
$ cd ~/buildroot
$ make O=../make BR2_EXTERNAL=../custom help
```

3.4 Hallo Welt: Unsere erste Firmware auf dem Raspberry Pi

Für den Fall, dass Sie sich nicht mehr sicher sind, ob Sie nach dem Ausprobieren noch eine sinnvolle Konfiguration vorliegen haben, hier nochmal die Befehle, um eine sicheren Ausgangslage zurückzukehren:

```
$ cd ~/make
$ make dhbw_minimal_defconfig
```

Oder falls das nicht klappt:

```
$ cd ~/buildroot
$ make BR2_EXTERNAL=../custom  O=../make help
$ cd ~/make
$ make dhbw_minimal_defconfig
```

Diese Konfiguration enthält nur die allernötigsten Komponenten für eine einfache Firmware ohne irgendwelche Zusatzprogramme. Die Firmware besteht lediglich aus dem Linux-Kernel, den Busybox-Systemwerkzeugen und einem SSH-Server, den wir später für das Remote Debugging benötigen werden. Bis auf ein paar ausgetauschte Systemdateien beinhaltet die Firmware auch noch keinerlei spezifische Anpassungen. Daher erscheint nach dem Hochfahren eine einfache Login-Zeile, mit der man sich am Raspberry Pi anmelden kann. Das generierte Image ist dabei 120 MB groß, was in der heutigen Zeit nicht besonders viel aber auch nicht wirklich klein ist. Für unsere Zwecke soll das aber genügen, da wir auf den heute üblichen SD-Karten mehr als genug Speicherplatz zur Verfügung haben.[24]

21 https://buildroot.org/download.html
22 https://buildroot.org/downloads/
23 https://git.busybox.net/buildroot, suchen Sie hier nach den sog. *Branches*, da jede Buildroot-Versionen in einem eigenen Zweig entwickelt wird.

Mit einem einfachen `make`-Aufruf wird die Firmware erstellt. Das fertige Image liegt danach im Verzeichnis `~/make/images` und muss nach `~/shared` kopiert werden, um es auf dem Host-Rechner nutzen zu können.

```
$ make
```

```
$ cp images/sdcard.img ../shared/
```

Wie es jetzt weitergeht hängt von Ihrem eingesetzten Betriebssystem ab. Unter Linux und Mac können Sie das Image mit dem `dd`-Kommando einfach auf eine Mini SD-Karte schreiben. Windows besitzt ein solches Werkzeug leider nicht, mit dem Win32 Disk Imager[25] gibt es aber ein brauchbares Open Source-Programm.

Bei der Bedienung dieser Programme sollten Sie extreme Vorsicht walten lassen. Denn das Firmware Image wird unter Umgehung aller höheren Betriebssystemfunktionen direkt auf die SD-Karte geschrieben[26]. Ein falscher Parameter oder ein kleiner Programmfehler und Sie überschreiben womöglich ihre gesamte Festplatte. Vergewissern Sie sich also lieber doppelt und dreifach, das richtige Laufwerk ausgewählt zu haben, bevor Sie den Schreibvorgang starten und nutzen Sie nach Möglichkeit nicht Ihren Firmenlaptop, um unangenehme Nachfragen Ihres IT-Helpdesks zu vermeiden.

☞ Vorsicht! Geht etwas schief, sind alle Ihre Daten weg … ☜

Einige moderne Linux-Desktopumgebungen beinhalten auch grafische Werkzeuge, mit denen das Firmware Image auf die SD-Karte geschrieben werden kann. Falls dem so ist, brauchen Sie nur die SD-Karte in den entsprechenden Card Reader-Slot Ihres Rechners zu stecken und das Image per Doppelklick öffnen. Unter GNOME erscheint dann zum Beispiel folgender Dialog:

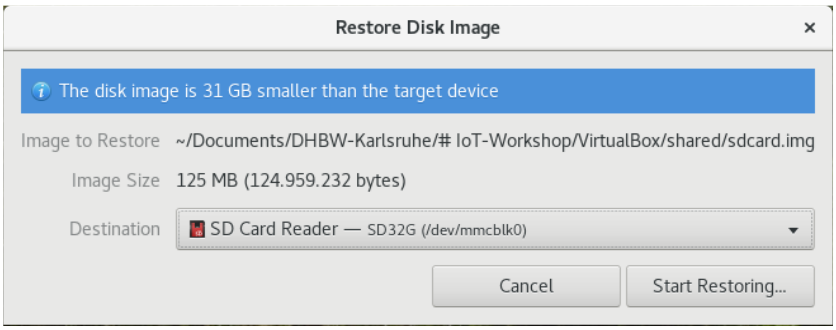

Abb. 14: Übertrag des Firmware Images auf eine SD-Karte unter GNOME

Wenn Ihr Betriebssystem kein solches Werkzeug bereithält, können Sie den Übertrag auch aus der Kommandozeile heraus durchführen. Sowohl unter Linux als auch Mac verwenden Sie folgende Befehle hierfür, wobei Sie zunächst herausfinden müssen, unter welchem technischen Namen der SD Card Reader angesprochen wird. Häufig handelt es sich dabei um einen Namen wie `/dev/mmcblk0`, was für *Multimedia Card Block Device Number 0* steht.

```
$ sudo dd if=sdcard.img of=/dev/mmcblk0
```

```
$ sync
```

Unter Windows können Sie wie gesagt das Programm Win32 Disk Imager nutzen.

Legen Sie nun die SD-Karte in den Raspberry Pi ein und schließen Sie eine Tastatur und einen Bildschirm an. Zuletzt stellen Sie eine Stromverbindung her, um den Raspberry Pi einzuschalten. Wenn alles klappt,

24 Die ersten Fritz!Box-Router hatten zum Beispiel nur 4 MB internen Speicher. Dennoch war es mit dem auf Buildroot basierten Freetz problemlos möglich, eine vollwertige, alternative Firmware zu bauen.

25 https://sourceforge.net/projects/win32diskimager/

26 Es werden also nicht einfach Dateien kopiert, da dies ein bereits vorhandenes Dateisystem auf der SD-Karte voraussetzt. Stattdessen werden auf einer viel niedrigeren Ebene die einzelnen Speicherzellen der SD-Karte überschrieben, wodurch die im Firmware Image enthaltenen Dateisysteme auf die SD-Karte übertragen werden.

sollten Sie den Raspberry Pi nun starten sehen und können sich binnen weniger Sekunden mit folgenden Benutzern anmelden:

Benutzer: mulder	**Benutzer:** scully
Passwort: xfiles	**Passwort:** xfiles

Abb. 15: Unsere neue Firmware beim Hochfahren

Allerdings können Sie noch nicht viel mit dem Raspberry Pi anfangen. Im Prinzip können Sie sich nur auf der Konsole anmelden und sich ein wenig umsehen. Anschließend fahren Sie ihn mit folgendem Befehl wieder herunter:[27]

```
$ sudo poweroff
```

3.5 SSH-Login am Raspberry Pi

In der Vorlagekonfiguration ist bereits ein SSH-Server integriert, so dass Sie sich auch ohne Bildschirm und Tastatur am Rasbperry Pi anmelden können. Einerseits sparen Sie sich so die sperrige Zusatzhardware, andererseits wird der SSH-Server später ohnehin für das Remote Debugging benötigt. Allerdings können Sie mit SSH nur sehr begrenzt die grafische Ausgabe des Raspberry Pi übertragen und das auch nur, wenn Sie einen X-Server in der Firmware aufnehmen, dessen Netzwerkprotokoll Sie über SSH tunneln. SSH eignet sich daher eigentlich nur für die Anmeldung per Konsole und zum Dateiaustausch.

Sicherheitshinweis: Der Einfachheit halber wird ein fest vorgegebener RSA Host Key in die Firmware eingebaut. Da somit alle Geräte denselben Host Key bekommen, ist dies eigentlich eine Sicherheitslücke. Wenn Sie das ändern wollen, tauschen Sie den Host Key aus oder aktivieren Sie die automatische Neugenerierung bei jedem Systemstart.

- Den vorgegebenen Host Key finden Sie unter `~/custom/board/rootfs_overlay/etc/ssh`.

27 Ja, es ist derselbe Befehl wie ganz am Anfang zum Herunterfahren der VM.

- Für die automatische Neugenerierung löschen Sie den vorgegebenen Key und kommentieren Sie markierte Zeile in der Datei `~/custom/board/rootfs_overlay/etc/inittab` wieder ein.

Damit Sie sich mit dem Raspberry Pi verbinden können müssen Sie dessen IP-Adresse kennen. Hierfür gibt es zwei Möglichkeiten: Einerseits besorgt sich die Firmware bei jedem Neustart eine dynamische IP-Adresse via DHCP. Diese finden Sie daher heraus, indem Sie sich in der Weboberfläche Ihres Routers umschauen. Weil dadurch aber nicht garantiert ist, dass der Raspberry Pi immer dieselbe IP-Adresse erhält, vergibt die Firmware zusätzlich noch die statische IP-Adresse `192.168.99.99`.

Der Vorteil der statischen IP-Adresse ist, dass sie sich niemals ändert. Der Nachteil ist aber, dass Sie die Netzwerkkonfiguration Ihres Entwicklungsrechners anpassen müssen, um sich mit dem Raspberry Pi zu verbinden. Die Verbindung klappt nämlich nur, wenn sich beide Computer im selben Subnetz befinden, das mit `198.168.99.0/24` definiert ist. Für den Zugriff per statischer IP-Adresse müssen Sie Ihrer Netzwerkkarte daher eine IP-Adresse aus demselben Nummernkreis geben.

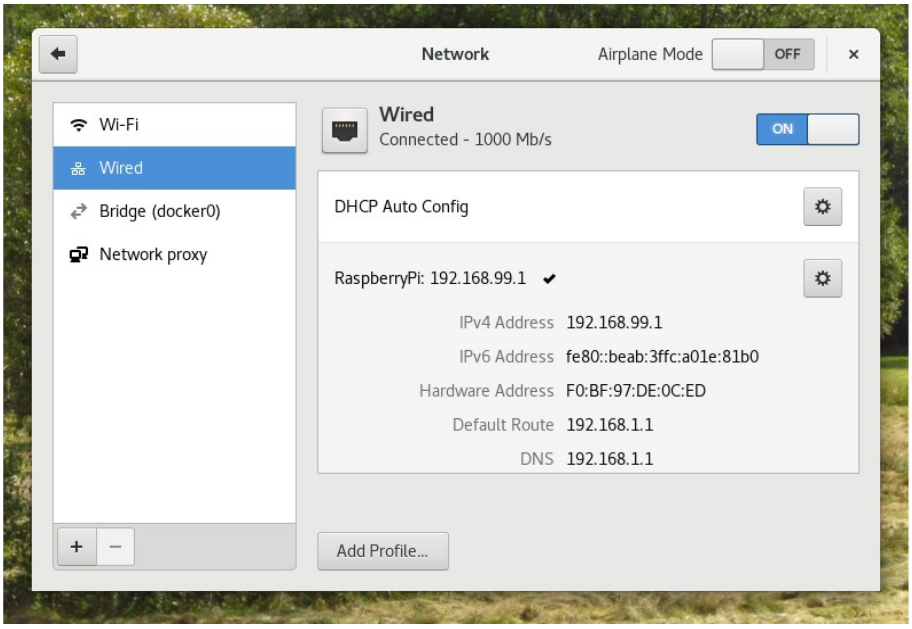

Abb. 16: Konfiguration einer statischen IP-Adresse im GNOME Network Manager

Eine andere Möglichkeit besteht darin, Ihren Router so zu konfigurieren, dass dem Raspberry Pi immer dieselbe IP-Adresse zugewiesen wird. Hierzu müssen Sie sich im Webinterface Ihres Routers anmelden, und die DHCP-Einstellungen anpassen. Die Zuweisung der festen IP-Adresse erfolgt dabei anhand der MAC-Adresse des Raspberry Pi, die Ihnen dort ebenfalls angezeigt werden sollte.

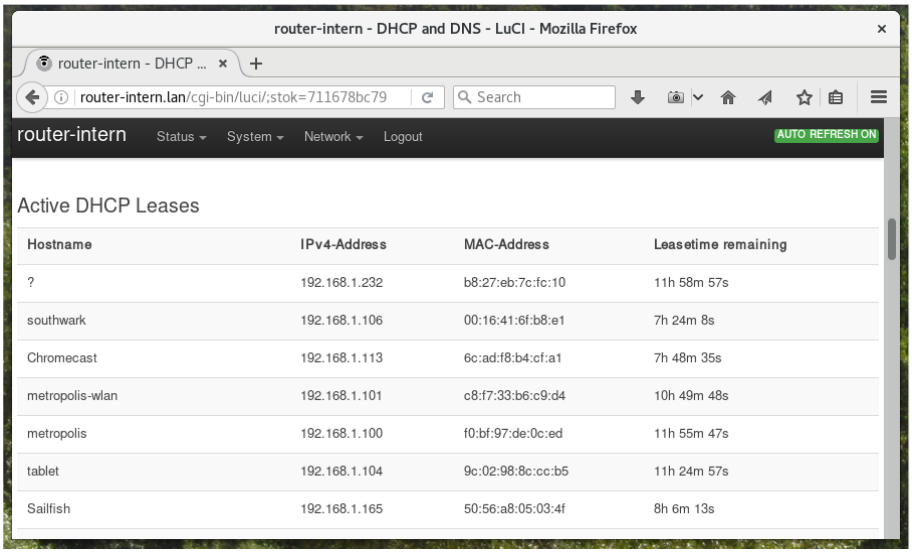

Abb. 17: Anzeige der aktiven DHCP-Leases in OpenWRT LuCI, der Raspberry Pi steht ganz oben

Sobald die IP-Adresse des Raspberry Pi bekannt ist, können Sie sich mit folgendem Befehl verbinden:

```
$ ssh mulder@192.168.99.99
```

Unter Windows mit PuTTY schreiben Sie stattdessen:

```
$ pssh mulder@192.168.99.99
```

Mit dem Befehl `exit` können Sie die Verbindung wieder trennen.

3.6 Dauerhafte Sicherung der Buildroot-Konfiguration

Alle Änderungen, die Sie im Konfigurationswerkzeug von Buildroot vornehmen, werden standardmäßig in der versteckten Datei `~/make/.config` gesichert.[28] Diese Datei wird daher auch herangezogen, wenn Sie den Build-Vorgang starten. Da die Datei aber versteckt ist, kann sie leicht übersehen werden und wenn man nicht aufpasst, verliert man dadurch alle seine Einstellungen. Aus diesem Grund sollten Sie in regelmäßigen Abständen eine Sicherung anlegen. Im einfachsten Fall kopieren Sie die Datei `~/make/.config` mit `cp` irgend wo hin, zum Beispiel so:

```
$ mkdir ~/backup
```

```
$ cp ~/make/.config ~/backup/buildroot-config
```

Um die gesicherte Konfiguration wiederherzustellen, können Sie dann folgenden Befehl ausführen:

```
$ cp ~/backup/buildroot-config ~/make/.config
```

Dies hat den Vorteil, dass Sie sich auf diese Weise beliebig viele Sicherungskopien anlegen können, ohne gleich ein eigenes git-Repository hierfür pflegen zu müssen. Beispielsweise könnten Sie die Dateien nach dem Muster *`jahr-monat-tag`*`_buildroot` benennen, um eine zeitlich aufsteigende Sortierung aller Änderungen zu bekommen. Und mit Werkzeugen wie `diff`, könnten Sie auch die Unterschiede in den Dateien sichtbar machen.

```
buildroot@debian:~/backup$ ls -1
2017-01-10_buildroot
2017-01-23-buildroot
...
```

28 Dateien, deren Namen mit einem Punkt beginnen sind gemäß Unix-Tradition versteckte Dateien. Mit dem Befehl `ls -a` oder `ls -alh` können Sie sich alle Dateien inklusive der sonst nicht sichtbaren anzeigen lassen.

Nachteilhaft daran ist aber, dass die so gesicherten Konfigurationen mehr Informationen als nötig enthalten, da auch die von Ihnen nicht angepassten Standardeinstellungen darin gespeichert sind. Außerdem passen die Konfigurationen nur zu Ihrer aktuellen Buildroot-Version so richtig. Mit hoher Wahrscheinlichkeit funktionieren sie zwar auch mit neueren Buildroot-Versionen, um ganz sicher zu gehen, sollten Sie Ihre Einstellungen jedoch lieber in Form einer neuen Standardkonfiguration abspeichern. Hierfür müssen Sie im Konfigurationsmenü zunächst den Namen der neuen Standardkonfiguration definieren:

```
$ cd ~/make
```

```
$ make menuconfig
```

Dort wählen Sie dann den Eintrag *Build options → Location to save buildroot config* aus und geben folgenden Wert ein: `../custom/configs/`*`eigener_name`*`_defconfig`. Der Name muss dabei zwingend mit der Zeichenkette `_defconfig` enden. Zum Beispiel so:

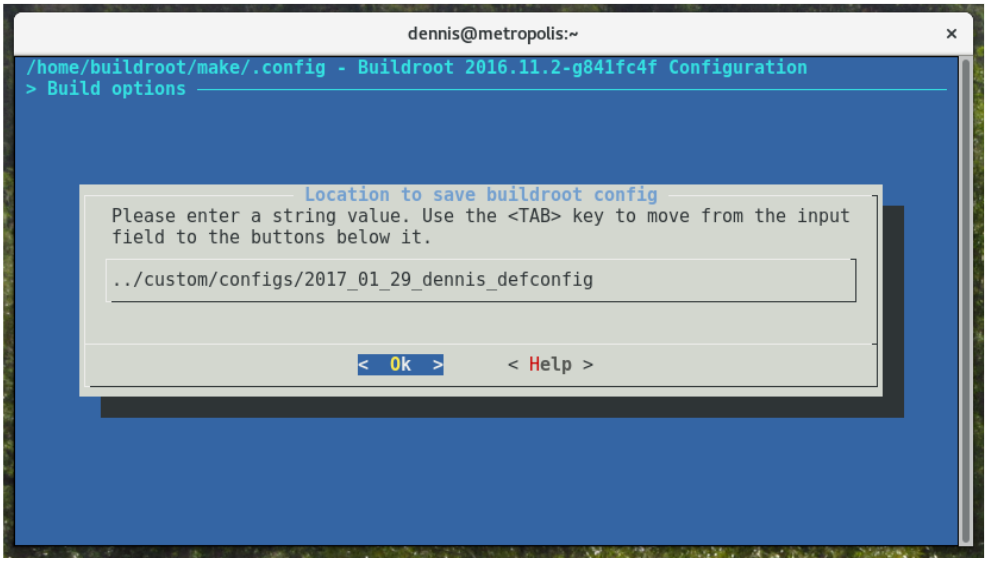

Abb. 18: Festlegen des Pfads der neuen Standardkonfiguration

Anschließend sichern Sie die Änderung und verlassen das Menü wieder. Sie müssen im Menü also sowohl die Option `<Save>` als auch `<Exit>` ausführen. Zurück auf der Konsole geben Sie dann folgenden Befehl ein, um die neue Standardkonfiguration zu speichern:

```
$ make savedefconfig
```

Mit folgendem Befehl können Sie prüfen, ob die neue Sicherung erkannt wird. Sie muss ganz am Ende der Ergebnisliste aufgeführt werden.

```
$ make list-defconfigs
```

Um die Sicherung wieder zu aktivieren, rufen Sie einfach `make` wie folgt auf. Dadurch wird die Datei `~/make/.config` mit dem Inhalt der übergebenen Sicherung überschrieben:

```
$ make 2017_01_29_dennis_defconfig
```

3.7 Wie der Raspberry Pi startet: Vom Einschalten bis zum Login

Wie bei allen Computern ist der Bootvorgang des Raspberry Pi komplizierter, als man es auf den ersten Blick vermuten würde. Das liegt daran, dass der Raspberry Pi zunächst drei verschiedene Boot Loader durchläuft, bevor das eigentliche Betriebssystem geladen wird. Das Betriebssystem bzw. in unserem Fall der Linux Kernel durchläuft anschließend ebenfalls mehrere Phasen, bevor das System gestartet und der Raspberry Pi einsatzbereit ist. Diese Komplexität ist für eingebettete Systeme durchaus üblich, erlaubt es aber auch, die Hard-

ware optimal auszunutzen und die Startzeit an verschiedenen Stellen zu optimieren. Die nachfolgende Grafik zeigt den groben Ablauf:

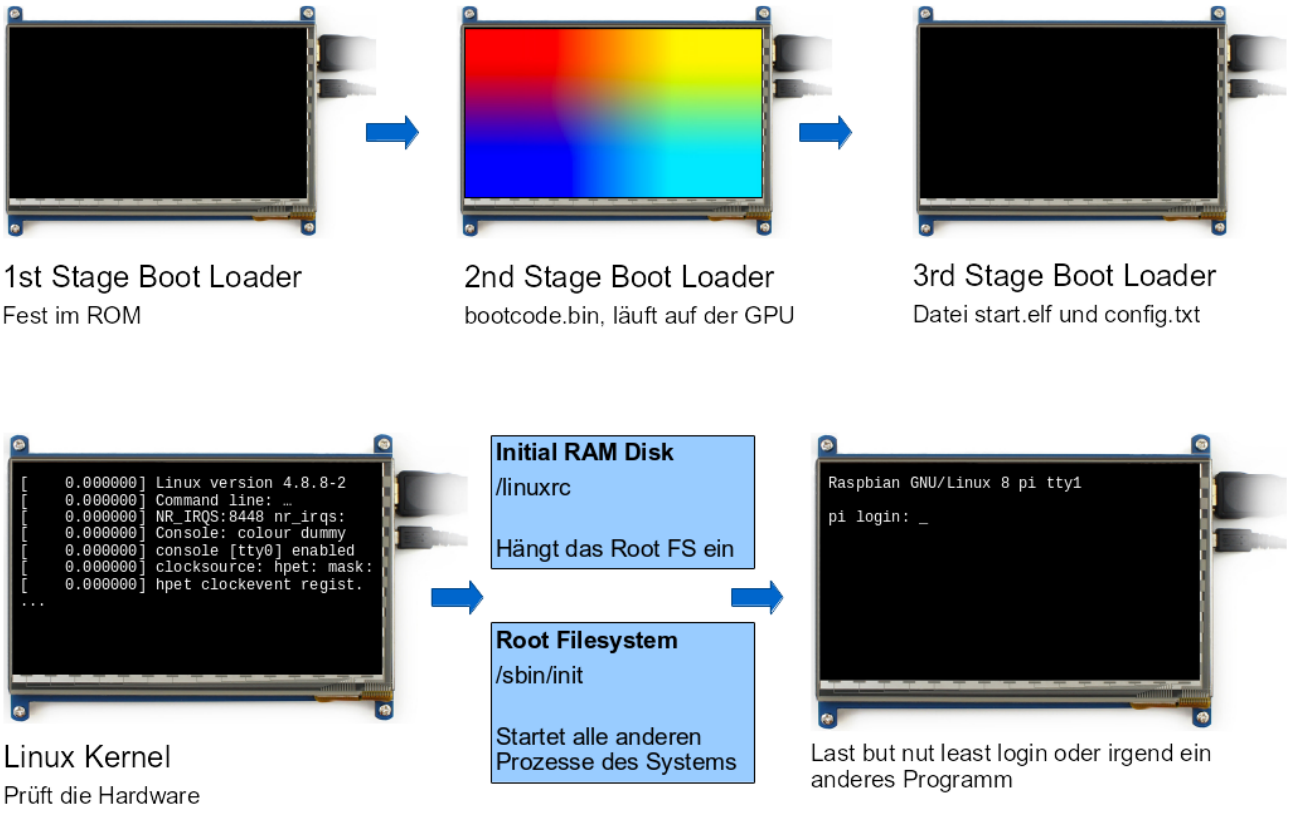

Abb. 19: Der Bootvorgang des Raspberry Pi

1. Zunächst wird der fest in der Hardware verbaute *1st Stage Boot Loader* ausgeführt. Dieser sucht am Anfang der SD-Karte eine VFAT-Partition[29] mit den restlichen Startdateien. Von dort wird die Datei `bootcode.bin` den Speicher geladen und auf der GPU[30] ausgeführt.

2. Bei der Datei `bootcode.bin` handelt es sich um den *2nd Stage Boot Loader*. Er initialisiert den Grafikprozessor und übergibt die Kontrolle dann an den *3rd Stage Boot Loader*. Beim ersten Raspberry Pi wurde an dieser Stelle ein großflächiges Regenbogenmuster zum Test der Grafikhardware angezeigt. Blieb dieses Muster nach ein paar Sekunden immer noch stehen, konnte man daran erkennen, dass das Betriebssystem nicht geladen werden konnte. Die neueren Raspberry Pi-Modelle verzichten allerdings auf die Anzeige, um die Startzeit zu verkürzen.

3. Anschließend wird die Kontrolle wieder an die CPU übergeben, indem mit der Datei `start.elf` der *3rd Stage Boot Loader* ausgeführt wird. Dieser wertet die beiden Dateien `config.txt` und `cmdline.txt` aus, um das Betriebssystem zu starten.

4. Nachdem der Linux-Kernel in den Speicher geladen wurde, wird dieser ausgeführt und die Kontrolle damit an das Betriebssystem übergeben. Der Kernel beginnt dabei, sich im Hauptspeicher zu entpacken (der Kernel wird aus Platzgründen meistens komprimiert) und die restliche Hardware zu initialisieren.

5. Innerhalb der Kernel-Datei befindet sich ein kleines, *Initial RAM Disk* genannten Dateisystem, das während der ersten Bootphase des Kernels in den Speicher geladen wird. Es beinhaltet im einfachs-

29 VFAT ist das alte Dateisystemformat von Windows. Es reicht zurück bis in die frühen 1980er-Jahre und ist daher besonders einfach aufgebaut. Mit Windows XP wurde es standardmäßig durch NTFS abgelöst.

30 Graphical Processing Unit, also der Grafikprozessor

ten Fall nur ein paar Programme, um das richtige Dateisystem einzuhängen und den Bootvorgang dann fortzusetzen. Je nach Linux-System können darin aber auch verschiedene Notfallreparaturprogramme oder die gesamte restliche Firmware enthalten sein.[31] Auf jeden Fall befindet sich darin die Datei `/linuxrc`, die der Kernel zum Starten des Systems ausführt.

6. Sobald das eigentliche *Root File System* von der SD-Karte eingehängt wurde, wird darin die Datei `/sbin/init` ausgeführt. Hierbei handelt es sich um das sogenannte Init System, das dafür verantwortlich ist, die Dienste und Prozesse des Betriebssystems zu starten. Historisch gesehen gibt es dabei viele verschiedene Init Systeme, Buildroot verwendet jedoch ein ganz einfaches, das seine Anweisungen aus der Datei `/etc/inittab` bezieht (vgl. Teilkapitel 3.12).

An dieser Stelle denken Sie vielleicht, dass der Bootvorgang eine gefühlte Ewigkeit dauern müsste. Dies ist aber mitnichten der Fall. Eine einfache mit Buildroot erstellte Firmware benötigt ohne Optimierungen zwischen vier und sechs Sekunden vom Einschalten bis zum Login. Wobei hierzu natürlich noch die Zeit kommt, die die später in die Firmware integrierten Programme zum Starten benötigen. Für unsere einfachen Anwendungsfälle ist das dann aber immer noch schnell genug.

Wollten Sie die Startzeit optimieren, wären jedoch folgende Ansätze denkbar:

1. Ersetzen der Datei `/sbin/init` durch ein eigenes Programm, das somit ohne Init System direkt beim Hochfahren gestartet wird.
2. Verzicht auf das zweite Dateisystem auf SD-Karte, so dass die komplette Firmware in der Initial RAM Disk enthalten ist.
3. Austausch der Datei `/linuxrc` in der Initial RAM Disk durch ein eigenes Programm. Ihr Programm wird dann zum frühest möglichen Zeitpunkt gestartet.

Unabhängig davon kann es sich auch lohnen, die Datei `cmdline.txt` auf der SD-Karte zu bearbeiten. Sie enthält die Startparameter, die dem Kernel mitgegeben werden und beinhaltet beim Raspberry Pi 3 standardmäßig folgende Zeile:

```
root=/dev/mmcblk0p2 rootwait console=tty1 console=ttyAMA0,115200
```

Um die Kernelmeldungen auszublenden und damit auch die Startzeit zu verkürzen, bietet es sich an, noch die `quiet`-Option hinzuzufügen. In Kapitel 3.14 ist auch beschrieben, wie die Änderung dauerhaft vorgenommen werden kann, so dass sie nicht bei jeder Erstellung der Firmware verloren geht.

```
root=/dev/mmcblk0p2 rootwait console=tty1 console=ttyAMA0,115200 quiet
```

3.8 Anatomie einer Linux-Firmware: Wichtige Verzeichnisse

Wie jedes eingebettete Systemboard hat auch der Raspberry Pi seine ganz speziellen Anforderungen, was die Speicheraufteilung angeht. Im Gegensatz zu vielen anderen Boards hat der Raspberry Pi ja keinen fest eingebauten Flash-Speicher, sondern muss mit einer Mini SD-Karte ausgestattet werden. Diese muss jedoch ganz am Anfang eine spezielle Boot-Partition mit fest vorgegebenen Dateien besitzen, damit der Raspberry Pi davon starten kann. Den restlichen Platz der SD-Karte kann man nach Belieben verwenden, solange man aus seiner Firmware heraus darauf zugreifen kann. Bei anderen Boards ist das in der Regel genauso. Deshalb findet man in der Praxis oft folgende Aufteilung, die gleichzeitig auch sichere Firmware-Upgrades ermöglicht:

`boot`	Boot-Partition mit den grundlegenden zum Starten benötigten Dateien. Beim Raspberry Pi muss hier auch der Linux-Kernel enthalten sein.
`recovery`	Recovery-Partition mit einem kleinen Notfallsystem zum Wiederherstellen des Werkzustands.
`system 1`	Systempartition mit der eigentlichen Firmware.

31 Insbesondere sog. Deeply Embedded Devices wie z.B. digitale Autotachos, die möglichst schnell starten sollen, verzichten auf ein weiteres Dateisystem und packen stattdessen alle benötigten Programme in die Initial RAM Disk.

`system 2`	Spiegelung der Systempartition. Bei einem Upgrade wird nur diese Partition überschrieben und anschließend der Bootloader so konfiguriert, dass sie anstelle der anderen verwendet wird. Beim nächsten Upgrade wird die andere Partition verändert und der Eintrag im Bootloader wieder getauscht. Somit hat man immer eine funktionsfähige Kopie der Firmware, selbst wenn das Upgrade abbricht.
`user`	Benutzerspezifische Daten, die bei einem Firmware-Upgrade nicht überschrieben werden sollen.

Diese Aufteilung findet man häufig bei Android-Smartphones und anderen Geräten, die ein Uprade der Firmware durch den Benutzer erlauben. Für unseren Anwendungsfall ist das jedoch zu viel, weshalb wir uns mit der Voreinstellung von Buildroot zufrieden geben:

`boot`	FAT32[32] Boot-Partition mit den Raspberry Pi-spezifischen Boot-Dateien und dem Linux-Kernel.
`system`	ext4[33] Root Filesystem unserer Firmware. Enthält das komplette Linux-System ohne den Kernel.

Die beiden Partitionen werden dabei von Buildroot automatisch erzeugt. Wir müssen nichts weiter tun, als die fertige Datei `sdcard.img` byteweise auf eine SD-Karte zu schreiben. Innerhalb der Boot-Partition befinden sich unter anderem übrigens folgende Dateien:

`bootcode.bin`	1st Stage Bootloader: Dieser wird auf der GPU ausgeführt und lädt den 2nd Stage Bootloader.
`start.elf`	2nd Stage Bootloader. Dieser wird auf der CPU ausgeführt und wertet die Datei `config.txt` aus, um den Kernel zu starten.
`config.txt`	Konfigurationsdatei, die der Raspberry Pi auswertet, bevor der Kernel gestartet wird. Hier können teilweise die Hardwareeigenschaften des Raspberry Pi verändert werden, um ihn zum Beispiel zu übertakten. Zusätzlich wird hier festgelegt, welches Kernel Image gestartet werden soll.
`zImage`	Der Linux-Kernel.
`cmdline.txt`	Startparameter des Linux-Kernels, diese bestimmten zum Beispiel wo das Root Filesystem liegt.

Für unser Projekt wesentlich interessanter ist das Root Filesystem, da hier die eigentliche Musik spielt. Dabei handelt es sich im Wesentlichen um gewöhnliches Linux-Dateisystem, das zumindest bei Buildroot dem *Linux Filesystem Hierarchy Standard*[34] entspricht. Andere Firmware-Familien neigen hingegen dazu, den Standard nicht allzu genau zu befolgen, so dass es hier mehr oder weniger große Abweichungen geben kann. Für den Linux-Kernel spielt die Verzeichnisstruktur ohnehin keine Rolle. Der Standard ist vielmehr dazu gedacht, einen gewissen Grad an Kompatibilität zwischen den Linux-Systemen herzustellen und dem Anwender zu helfen, sich zurechtzufinden.

Technisch gesehen befindet sich in jedem Linux-Kernel ein kleines *Initial RAM Disk* genanntes Dateisystem, von dem der Kernel zunächst startet. Im einfachsten Fall befindet sich darin nur ein Programm, das dafür sorgt, dass das eigentliche Dateisystem eingehängt und der Startvorgang fortgeführt wird. Auf Desktopsystemen liegen häufig aber auch Reparaturwerkzeuge in der Initial RAM Disk, mit denen ein sonst nicht bootfähiges Linuxsystem mit etwas Glück wieder zum Laufen gebracht werden kann. Einige eingebettete Geräte gehen sogar soweit, das gesamte Root Filesystem hierhin zu verlagern um die Startzeit zu verkürzen und das Firmware Image zu vereinfachen. Mit der Initial RAM Disk werden wir uns im Projekt allerdings nicht weiter beschäftigen und auch die Voreinstellung, ein separates Root Filesystem außerhalb der Initial RAM Disk anzulegen, müssen Sie nicht verändern.

Schauen wir uns an dieser Stelle noch den Inhalt des Root Filesystems an, damit Sie wissen, wo Sie später Ihre eigenen Dateien ablegen können oder wo Sie suchen müssen, wenn Sie eine Konfigurationsdatei durch ein Overlay überschreiben müssen (vgl. nächstes Kapitel). Die wichtigsten Einträge sind rot hervorgehoben.

Ausführbare Programme

`linuxrc -> bin/busybox`	Der sog. PID 1. Das erste Programm, dass beim Hochfahren ausgeführt wird und alle anderen Programme startet. Diese Aufgabe übernimmt für uns (wie so vieles anderes) Busybox, weshalb

32 FAT ist das ursprüngliche Windows-Dateisystemformat, das es ebenfalls von MS-DOS geerbt hat. Seit Windows NT wurde es weitgehend durch NTFS abgelöst, kommt bei eingebetteten Systemen aufgrund seiner Einfachheit aber immer noch oft zum Einsatz. Dem stehen in jüngerer Zeit allerdings Patentklagen von Microsoft entgegen.

33 ext4 ist das aktuelle Standard-Dateisystemformat von Linux. Neben ext4 beherrscht Linux noch viele weitere.

34 http://www.pathname.com/fhs/

	`/linuxrc` nur ein symbolischer Verweis auf `/bin/busybox` ist.
`bin -> usr/bin`	Ausführbare Programme, die jeder Benutzer ausführen darf. `/bin` verweist hier auf `/usr/bin`, um eine Doppeldeutigkeit bei der Ablage der Dateien zu vermeiden.[35]
`sbin -> usr/sbin`	Ausführbare Programme, die nur der Super User ausführen darf. Meist handelt es sich dabei um Adminwerkzeuge zur Systemwartung. Analog zu `/bin` verweist `/sbin` hier auf `/usr/sbin`.
`opt`	Leeres Verzeichnis, in das man eigene Programme packen kann. Hier können Sie beliebige Unterverzeichnisse anlegen und ihre selbstgeschriebenen Programme hineinkopieren.

Systembibliotheken

`lib -> usr/lib`	Systembibliotheken, die zur Ausführung der Programme benötigt werden. Auch hier handelt es such um einen SymLink nach `/usr/lib`.
`lib32 -> lib`	Alternatives `lib`-Verzeichnis mit 32-bit Bibliotheken. Dieses Verzeichnis findet man gelegentlich auf 64-bit Systemen. Sie haben es erraten, Buildroot legt einen SymLink hierfür an.
`usr`	Zweite Dateisystemhierarchie. Teilweise sind hier dieselben Verzeichnisse wie in `/` enthalten, weshalb Buildroot viele SymLinks hierauf einrichten. Im Gegensatz zu `/` muss `/usr` während dem Bootvorgang nicht vorhanden sein, sondern kann nachträglich über ein Netzlaufwerk eingebunden werden. Dies hat heute aber selbst bei Servern kaum noch eine Bedeutung.

Konfiguration

`etc`	Konfigurationsdateien des Systems und aller Programme. Im Gegensatz zu Windows und macOS werden in Linux traditionell alle Einstellungen in Textdateien vorgenommen. In diesem Verzeichnis werden Sie daher wohl etwas Zeit verbringen.

Variable Daten

`var`	Variable Daten, die durch die Ausführung der Programme entstehen und dauerhaft gespeichert werden sollen. Spielt für eingebettete Systeme eher nur eine untergeordnete Rolle. Beispielsweise befindet sich hier traditionell die Maildatei jedes Systembenutzers oder der Drucker-Spool.
`run`	Laufzeitdaten, die nur für die Dauer einer Programmausführung benötigt werden.
`media`	Leeres Verzeichnis, in das automatisch erkannte externe Datenträger eingehängt werden. Dabei handelt es sich sozusagen um den geistigen Nachfolger von `/mnt`, der in vielen Systemen seinerseits aber von `/run/media` abgelöst wurde.
`mnt`	Leeres Hilfsverzeichnis, das man nutzen kann, um vorübergehend ein externes Laufwerk einzuhängen. Dieses Verzeichnis wird vom System selbst nicht genutzt sondern ist nur dafür da, damit man beim manuellen „mounten“ einer Partition nicht immer ein Verzeichnis anlegen muss. Zum Beispiel so: `$ sudo mount -t autofs /dev/sdb1 /mnt`

Benutzerverzeichnisse

`home`	Enthält die Home-Verzeichnisse der einzelnen Benutzer.
`root`	Home-Verzeichnis des root-Benutzers. Im Gegensatz zu `/home` muss `/root` immer vorhanden sein und sollte nicht über ein Netzlaufwerk eingehängt werden.

Kernel und Hardware

`dev`	Enthält virtuelle Dateien, über die auf alle Hardwarekomponenten zugegriffen werden kann entsprechend *Everything is a file*-Mentalität von UNIX.
`proc`	Virtuelles Dateisystem mit Laufzeitinformationen des Kernels.
`sys`	Geistiger Nachfolger von `/proc` als man erkannte, dass `/proc` ziemlich chaotisch aufgebaut ist.

35 Für eingebettete Systeme ergibt die Unterscheidung von `/bin` und `/usr/bin` ohnehin keinen Sinn. Sie kommt aus der Frühzeit der UNIX-Cluster, wo `/bin` auf dem lokalen Rechner und `/usr` auf einem Netzlaufwerk liegen konnte.

Sonstige Verzeichnisse

`tmp`	Temporäre Dateien, die beim Ausschalten verloren gehen.

3.9 Anpassungen am Dateisystem mit Overlays

Jetzt da Sie wissen, wie das Linux Root Filesystem ungefähr aufgebaut ist, muss ich Ihnen natürlich auch zeigen, wie Sie eigene Dateien darin aufnehmen können. Hierfür sieht Buildroot verschiedene Mechanismen vor, die alle ihre besonderen Vor- und Nachteile besitzen:[36]

- *Overlays* sind der einfachste Mechanismus. Es handelt sich lediglich um einfache Verzeichnisse, deren Inhalte eins zu eins in das Root Filesystem hinein kopiert werden. Dadurch wird es möglich, neue Dateien hinzuzufügen oder bereits vorhandene Dateien zu überschreiben.
- *Post Build Scripts* sind Skripte, die nach dem Erstellen der Dateien aber vor dem Bauen des Firmware Images ausgeführt werden. Mit ihnen kann man überflüssige Dateien löschen oder die Inhalte der Dateien manipulieren, wodurch die Skripte aber schnell unübersichtlich werden.
- *Project-specific Packages*[37] sind eigene Programmpakete, die wie die anderen *Target Packages* von Buildroot im Konfigurationsmenü ausgewählt werden können. Sie werden während dem Bauen der Firmware als Quellcode heruntergeladen und für das Zielsystem kompiliert. Allerdings muss man sicher hierfür mit der Syntax von Makefiles und der Funktionsweise von Buildroot auseinandersetzen.
- *Custom Target Skeleton* bedeutet, dass man seine ganz eigene Vorlage für das Root Filesystem verwendet. Dadurch kann man viele der oben aufgezählten, eigentlich überflüssigen Verzeichnisse und symbolischen Verweise einsparen, gleichzeitig macht das aber auch den meisten Aufwand. Das Buildroot-Handbuch rät deshalb von dieser Option in den meisten Fällen ab.

An dieser Stelle schauen wir uns nur die Overlays weiter an, das sie sehr einfach zu verwenden sind und es uns auf ein oder zwei MB mehr oder weniger nicht ankommen soll. Sollten Sie jedoch erwägen, Ihre Programme zum Beispiel in C oder C++ statt in Java zu schreiben, werden Sie um Project-specific Packages nicht umhin kommen. Mit Hilfe des Handbuchs sollte dies für Sie dann aber kein Problem sein.

Die bereitgestellten DHBW-Vorlagekonfigurationen beinhalten bereits mehrere Overlays, die sich um Verzeichnis `~/custom/board` befinden.

```
buildroot@debian:~custom/board$ ls -1
rootfs_overlay
rootfs_overlay_custom
rootfs_overlay_java
buildroot@debian:~$
```

Diese Verzeichnisse sind wie folgt vorgesehen:

- **`rootfs_overlay`:** Hier liegen einfache Standardkonfigurationen, die das eingebettete System ein wenig benutzbar machen. Beispielsweise befindet sich hier die Netzwerkkonfiguration oder der SSH Host Key. Ebenso finden Sie hier eine Vorlage für die Datei `/etc/inittab`, die weiter unten noch beschrieben wird.
- **`rootfs_overlay_java`:** Dieses Overlay ist nur in der `dhbw_java_defconfig` aktiv und dient dazu, das Oracle JDK in die Firmware aufzunehmen. Anfänglich ist das Verzeichnis leer, da Sie das JDK erst noch herunterladen und mit `~/custom/unpack_java_overlay.sh` entpacken müssen.

36 https://buildroot.org/downloads/manual/manual.html#rootfs-custom
37 https://buildroot.org/downloads/manual/manual.html#customize-packages

Falls Sie die vorgegebene Struktur nicht beibehalten wollen, können Sie natürlich noch weitere Verzeichnisse anlegen. Diese müssen Sie in der Buildroot-Konfiguration dann bei *System configuration → Root filesystem overlay directories* (der fünfte Menüpunkt von unten) eintragen. Haben Sie zum Beispiel noch das Overlay-Verzeichnis `rootfs_overlay_python` angelegt, müssen Sie die Option wie folgt ergänzen:

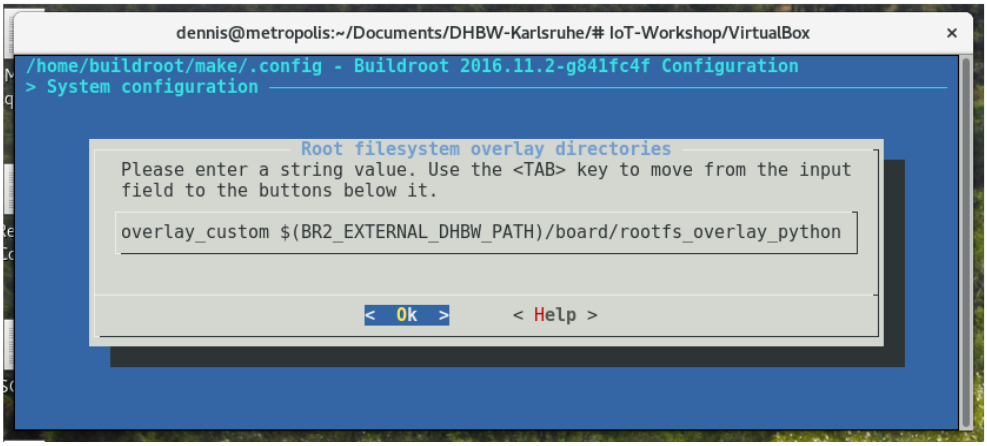

Abb. 20: Erweiterung der Overlay-Verzeichnisse um rootfs_overlay_python

3.10 Benutzerverwaltung und Rechtevergabe

In einer frischen Buildroot-Konfiguration ohne jedwede Anpassung sind keine Benutzerkonten definiert, die im Linuxsystem der Firmware angelegt werden sollen. Dies führt dazu, dass man sich am System nicht einloggen kann, da der standardmäßig angelegte Benutzer `root` sowie die Systemkonten kein gültiges Login-Passwort besitzen. Im Prinzip braucht man auch keine weiteren Benutzerkonten, man müsste dann aber alle Programme unter dem `root`-Benutzer mit Super User-Rechten laufen lassen, was allerdings keine sehr gute Idee ist.[38] In der DHBW-Standardkonfiguration wurde dem durch die beiden Benutzer `mulder` und `scully` entgegengewirkt. Doch woher weiß Buildroot, welche Benutzer es anlegen soll?

Das Geheimnis liegt in der Buildroot-Option *System configuration → Path to the users tables*. Dort ist in der Vorlage der Wert `$(BR2_EXTERNAL_DHBW_PATH)/users` eingetragen, womit nach Auflösung der Variable die Datei `~/custom/users` gemeint ist. Diese Datei hat folgenden Inhalt:

```
buildroot@debian:~/custom$ more users
mulder   -1   xfiles   -1   =xfiles   /home/mulder   /bin/sh   wheel   Fox Mulder
scully   -1   xfiles   -1   =xfiles   /home/scully   /bin/sh   wheel   Dana Scully
```

Es handelt sich um eine einfache Textdatei, die je Zeile einen Benutzer definiert. Dabei besteht jede Zeile aus mehreren Feldern, die durch Whitespace voneinander getrennt werden. Bis auf das letzte Feld muss jede Zeile jedes Feld beinhalten und in keinem Feld darf ein Doppelpunkt vorkommen. Die Felder haben dabei folgende Bedeutung:[39]

Benutzername	Der Login-Name des Benutzers.
Benutzer-ID	Die nummerische Benutzer-ID oder `-1`, wenn die ID automatisch vergeben werden soll.
Gruppenname	Die hauptsächliche Benutzergruppe des Benutzers. Typischerweise gehört jeder Login-Benutzer zu einer gleichnamigen Benutzergruppe, in der keine anderen Benutzer enthalten sind. Man kann die Grup-

38 Traditionell besitzt jedes UNIX-System einen Benutzer `root` mit der Benutzer-ID 0 und vollen Super User-Rechten. Dieser Benutzer war ursprünglich zur Administration des Systems gedacht. Inzwischen ist man aber dazu übergegangen, keinen direkten `root`-Login mehr zu erlauben. Stattdessen muss sich ein normaler Systembenutzer mit Befehlen wie `su` oder `sudo` (Super User Do) vorübergehend Adminrechte beschaffen. `sudo` hat dabei den Vorteil, dass die Authentifizierung nicht mit dem Kennwort von `root` sondern des jeweiligen Benutzers erfolgt.

39 https://buildroot.org/downloads/manual/manual.html#customize-users sowie https://buildroot.org/downloads/manual/manual.html#makeuser-syntax

	pe aber auch dazu nutzen, die Zugriffsrechte bestimmter Pfade für mehrere Benutzer festzulegen.
Gruppen-ID	Die nummerische Gruppen-IP oder `-1`, wenn die ID automatisch vergeben werden soll.
Passwort	Das Passwort des Benutzers. Ist dem Passwort ein = vorangestellt, ist es im Klartext abgelegt, andernfalls ist es bereits verschlüsselt. Stellt man dem Passwort zusätzlich noch ein `!` Vorweg, besitzt der User zwar ein Kennwort, kann sich damit aber nicht anmelden. Alternativ kann man auch nur ein `*` ohne Kennwort eintragen, um den Login zu sperren. Trägt man nur ein `-` ein, besitzt der Benutzer einfach kein Passwort.
Home-Verzeichnis	Pfad des Home-Verzeichnisses des Benutzers oder `-`, wenn der Benutzer kein eigenes Home-Verzeichnis besitzen soll. Das Verzeichnis wird mit den entsprechenden Berechtigungen automatisch angelegt.
Login-Shell	Programm, das ausgeführt wird, wenn sich der Benutzer anmeldet. Üblicherweise wird hier `/bin/sh` eingetragen, um eine Shell zu starten. Häufig sieht man auf `/bin/false`, wodurch sich der Benutzer zwar anmelden kann, aber sofort wieder ausgeloggt wird. Hierfür kann auch einfach `–` eingetragen werden.
Weitere Gruppen	Komma getrennte Liste weiterer Gruppen, denen der Benutzer angehört oder `-`. Zwischen den Gruppennamen darf kein Leerzeichen stehen. Fehlende Gruppen werden automatisch angelegt. Im obigen Beispiel werden die beiden Benutzer der Gruppe `wheel` zugeordnet, um Ihnen die Benutzung des Hilfsprogramms `sudo` zu ermöglichen. Auf diese Weise können einem Benutzer auf sichere Art Super User-Rechte gegeben werden.
Kommentar	Freitextfeld, in dem üblicherweise der vollständige Name des Benutzers oder eine Beschreibung seiner Funktion eingetragen wird. Dieses Feld ist als einziges optional.

Ein nicht privilegierter Benutzer, der sich nicht interaktiv anmelden kann und nur dazu da ist, beim Hochfahren des Systems ein Programm ohne Super User-Rechte auszuführen, könnt zum Beispiel so aussehen:

```
buildroot@debian:~/custom$ more users
walter_skinner   -1   walter_skinner   -1   *   -   -   -
```

3.11 Zugriffsrechte einzelner Dateien ändern

In manchen Fällen kann es vorkommen, dass Sie die Zugriffsrechte einzelner Dateien innerhalb der Firmwäre ändern möchten. Beispielsweise wenn Sie wollen, dass bestimmte Dateien nur von `root` oder einem speziellen anderen Benutzer lesbar und/oder beschreibbar sind. Buildroot sieht hierfür einen ähnlichen Mechanismus wie zur Anlage von Benutzer vor, nämlich dass die zu verändernden Zugriffsrechte in einer Konfigurationsdatei festgelegt werden müssen. Diese Datei sollten Sie `permissions` nennen und um `~/custom`-Verzeichnis ablegen.

Die Syntax der Datei ist an folgender Stelle in Kapitel 23 im Buildroot-Handbuch beschrieben.[40] Jede Zeile beschreibt eine vorzunehmende Änderung und beinhaltet hierfür mehrere Felder, die durch Whitespace getrennt werden. Wenn die hier eingetragenen Dateien schon in der Firmware enthalten sind, können somit ihre Zugriffsrechte definiert werden. Andernfalls legt Buildroot die Dateien automatisch an.

Dateiname	Der vollständige Pfad der Datei, die verändert werden soll. z.B.: `/etc/sshd/ssh_host_rsa_key`
Dateityp	Ein Buchstabe, der den Typ der Datei kennzeichnet. f: Datei d: Verzeichnis r: Verzeichnis inklusive aller Oberverzeichnissen c: Character Device b: Block Device p: Named Pipe
Zugriffsrechte	Nummerischer Code mit den Zugriffsrechten. Diesen können Sie mit der Webseite http://chmod-calculator.com/ berechnen lassen.
Benutzername	Benutzername des Eigentümers der Datei. Dies kann entweder der ausgeschriebene Benutzername

40 https://buildroot.org/downloads/manual/manual.html#makedev-syntax

	(z.B. `root` oder `scully`) oder die nummerische Benutzer-ID aus der Datei `/etc/passwd` sein.
Gruppenname	Benutzergruppe der Datei. Auch hier kann entweder der ausgeschriebene Name der Gruppe oder die numerische Kennung aus der Datei `/etc/group` verwendet werden.
Major	Siehe Buildroot-Handbuch. Sollte in unserem Fall immer `-` sein.
Minor	Siehe Buildroot-Handbuch. Sollte in unserem Fall immer `-` sein.
Start	Siehe Buildroot-Handbuch. Sollte in unserem Fall immer `-` sein.
Inc	Siehe Buildroot-Handbuch. Sollte in unserem Fall immer `-` sein.
Count	Siehe Buildroot-Handbuch. Sollte in unserem Fall immer `-` sein.

Angenommen Sie wollen nun sicherstellen, dass der Private Key des SSH-Servers nur für den Root-Benutzer lesbar ist[41]. Dann legen Sie innerhalb des `~/custom`-Verzeichnisses eine Datei namens `permissions` mit folgendem Inhalt an:

```
buildroot@debian:~/custom$ more permissions
/etc/ssh/ssh_host_rsa_key    f    600    root    root    -    -    -    -    -
```

Anschließend starten Sie das Buildroot-Menü und wählen folgenden Menüeintrag aus:

System configuration → Path to the permission tables

Dort befindet sich bereits der Eintrag `system/device_table.txt`. Ergänzen Sie ihn wie folgt um ihre eigene Konfigurationsdatei: `system/device_table.txt $(BR2_EXTERNAL_DHBW_PATH)/permissions`

3.12 Automatischer Start von Programmen beim Hochfahren

Bei der Beschreibung der Linux Dateisystemstruktur haben Sie bereits gelernt, dass PID 1 das einzige direkt vom Linux-Kernel ausgeführte Programm ist und es daher die Aufgabe hat, alle anderen Dienste und Programme zu starten. Dieses Programm, das übrigens ständig laufen muss und niemals beendet werden darf, wird deshalb auch Init-System genannt.[42] Lange Zeit gab es eigentlich nur ein wirklich weit verbreitetes Init-System für Linux: `sysvinit`, benannt nach *UNIX System V*[43], wo es 1983 eingeführt wurde. Nach über zwanzig Jahren aktivem Einsatz gilt es inzwischen aber als veraltet und wurde nahezu vollständig durch modernere Alternativen wie `systemd` abgelöst.

Im Bereich der eingebetteten Systeme gibt es heute im Wesentlichen zwei Lager: Die ganz einfachen, in sich geschlossenen Systeme, die ein möglichst einfaches Init-System wie das von Busybox nutzen und die ganz großen Systeme, die überwiegend auf `systemd` zurückgreifen. Für unsere Fälle reicht Busybox völlig aus, so dass wir uns an dieser Stelle nicht weiter mit `systemd` beschäftigen werden. Stattdessen müssen wir uns nur anschauen, wie die Einträge in der Datei `/etc/inittab` aussehen, da diese Datei die zu startenden Programme definiert.

Das nachfolgende Beispiel zeigt eine einfache `/etc/inittab`. Die im `rootfs_overlay` vorgegebene Datei enthält allerdings noch weitere Einträge und viele erklärende Kommentare.

```
# /etc/inittab
::sysinit:/bin/mount -o remount,rw /
::sysinit:/bin/mount -a

::shutdown:/sbin/swapoff -a
```

41 Neuere OpenSSH-Versionen setzen dies zwingend voraus.

42 Unter normalen Umständen verhindert Linux, dass PID 1 beendet wird bzw. fast dies als Aufforderung zum Herunterfahren auf. Sollte das Programm dennoch mal abstürzen, reißt es den Kernel mit und das gesamte System verabschiedet sich mit einer *Kernel Panic*.

43 Ausgesprochen *System Five*

```
::shutdown:/bin/umount -a -r

tty1::respawn:/sbin/getty -L  tty1 0 vt100 # HDMI console
::ctrlaltdel:/sbin/reboot
```

Jede Zeile definiert genau ein auszuführendes Programm, wobei jede Zeile mehrere durch Doppelpunkt getrennte Felder enthält. Das erste Feld, die ID, befindet sich vor dem ersten Doppelpunkt, wird aber in den meisten Zeilen nicht benötigt. Die Felder haben folgende Bedeutung:

id	Entweder `tty1`, `tty2`, … bzw. `console`, um die virtuelle Konsole festzulegen, auf das Programm läuft. Zwischen den virtuellen Konsolen kann mit `[Strg]+[Alt]+[F1]`, `[Strg]+[Alt]+[F2]` usw. umgeschaltet werden. Meistens ist das Feld leer und wird nur in den Zeilen genutzt, in denen eine interaktive Login-Konsole gestartet wird.
runlevel	Von Busybox ignoriert und daher immer leer.
action	Bestimmt, zu welchem Zeitpunkt das Programm wie ausgeführt wird. Busybox kennt folgende Werte: `sysinit`, `shutdown`, `respawn`, `askfirst`, `wait` oder `once`.
process	Das auszuführende Kommando

Wirklich interessant sind eigentlich nur die letzten beiden Felder, so dass die anderen beiden meistens leer bleiben. Im Beispiel oben erkennt man das an den führenden Doppelpunkten. Die Werte in der Spalte `action` wirken sich dabei so aus:[44]

- **`sysinit:`** Diese Einträge werden beim Hochfahren des Systems ausgeführt. Busybox wartet dabei, bis das aufgerufene Programm sich beendet, bevor es mit dem nächsten Eintrag fortfährt.
- **`wait:`** Im Prinzip dasselbe wie `sysinit`, jedoch werden die `wait`-Zeilen erst ausgeführt, wenn alle `sysinit` abgearbeitet wurden.
- **`once:`** Hier wartet Busybox nicht, bis sich das Programm beendet, sondern startet das Programm im Hintergrund. Diese Option muss daher verwendet werden, um langlaufende Server zu starten.
- **`respawn:`** Wie `wait`, jedoch wird das Programm sofort neugestartet, sobald es sich beendet. Dies wird zum Beispiel für die Login-Konsole genutzt, da diese nach dem Ausloggen beendet wird.
- **`askfirst:`** Wie `respawn`, jedoch muss der Anwender den Start erst durch Drücken einer Taste bestätigen. Der angezeigte Fragetext lautet *Please press Enter to activate this console*.
- **`shutdown:`** Diese Einträge werden beim Herunterfahren des Systems ausgeführt.
- **`ctrlaltdel:`** Das angegebene Programm wird ausgeführt, wenn der Anwender [Strg]+[Alt]+[Entf] drückt. Meistens wird hier `/sbin/reboot` eingetragen, um den Computer neuzustarten.

Möchten Sie beim Hochfahren zum Beispiel das Programm `psplash` starten, um die Bootmeldungen zu verstecken, kann dies durch folgenden Eintrag geschehen:

```
::sysinit:/bin/psplash
```

Der Nachteil daran ist jedoch, dass das Programm mit Root-Rechten ausgeführt wird und daher unter anderem alle Dateien der Firmware überschreiben kann. Viel sicherer ist es deshalb, wie im vorherigen Kapitel gezeigt, einen eigenen Benutzer anzulegen und diesen das Programm starten zu lassen. Hierfür muss das eigentliche Kommando mit `sudo` ausgeführt werden, wie das folgende Beispiel anhand des ebenfalls aus Akte X entlehnten Benutzers `trustno1` zeigt:

```
::sysinit:/bin/sudo -u trustno1 /bin/psplash
```

44 http://spblinux.de/2.0/doc/init.html

Und schon ist das Sicherheitsleck gestopft.

3.13 Detailkonfiguration von Kernel und Busybox

Für das Projekt werden Sie das höchstwahrscheinlich nicht benötigen. Der Vollständigkeit halber sie an dieser Stelle jedoch darauf hingewiesen, dass es neben dem bekannten Buildroot-Konfigurationsmenü noch zwei weitere ähnliche Konfigurationswerkzeuge gibt. Eines zum Konfigurieren des Kernels[45] und eines zum Konfigurieren der Busybox-Werkzeugsammlung.[46]

Das Linux-Konfigurationsmenü rufen Sie wie folgt auf:

```
$ make linux-menuconfig
```

Und das für Busybox so:

```
$ make busybox-menuconfig
```

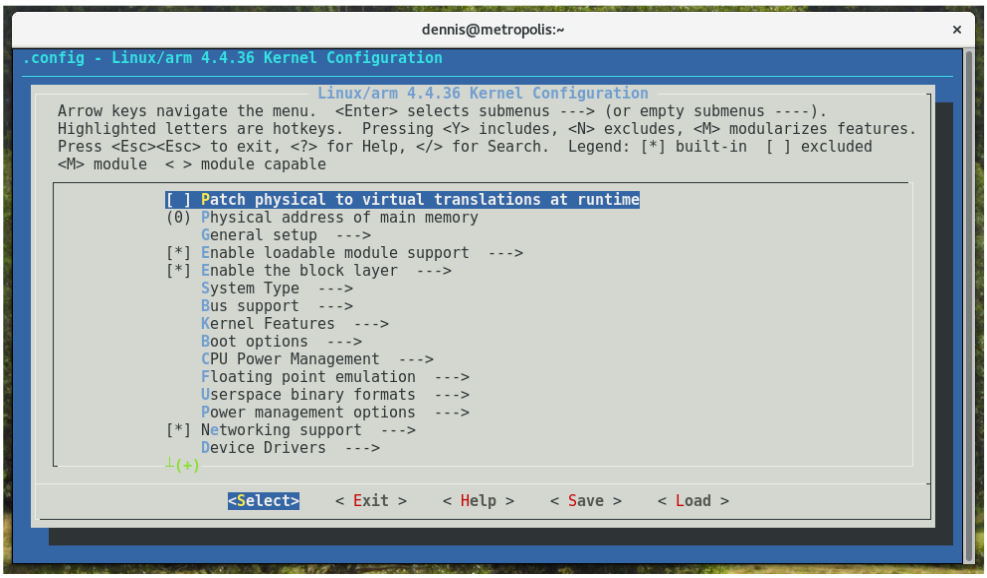

Abb. 21: Detailkonfiguration des Linux-Kernels

Im Buildroot-Handbuch sind insbesondere die Kapitel 7 und 9.4 interessant.[47] Leider ist das Handling etwas umständlich, wenn man die Konfigurationen dauerhaft sichern will. Am Beispiel des Kernels sei das Vorgehen beispielhaft erläutert.

Zunächst ruft man das Kernel-Konfigurationsmenü auf und nimmt dort eine Einstellungen vor:

```
$ make linux-menuconfig
```

Leider sieht man beim Speichern der Konfiguration nicht, in welchem Verzeichnis die Datei landet. Im entsprechenden Popupfenster wird nur `.config` angezeigt. Tatsächlich liegt die Datei im Build-Verzeichnis des Kernels, das `~/make/build/linux-c6d86f7aa554854b04614ebb4d394766081fb41f` heißen kann. Die lange zufällige Buchstabenkette entspricht dabei einer GUID, welche die Kernelversion eindeutig identifiziert. Das

45 Tatsächlich das Konfigurationswerkzeug des Linux-Kernels die Vorlage für die anderen beiden.

46 Busybox stellt in unserem eingebetteten System die Shell sowie nahezu alle grundlegenden Linux-Kommandos bereit. Busybox ist dabei insbesondere auf eine kleine Größe und Geschwindigkeit optimiert, nicht jedoch auf einen möglichst breiten Funktionsumfang. Buildroot ist aus dem Busybox-Projekt hervorgegangen, da die Entwickler eine Möglichkeit brauchten, Busybox auf echter Hardware zu testen.

47 https://buildroot.org/downloads/manual/manual.html#_configuration_of_other_components sowie https://buildroot.org/downloads/manual/manual.html#customize-store-package-config

Verzeichnis kann sich daher jederzeit ändern und umso wichtiger ist daher, dass Sie die Kernel-Konfiguration dauerhaft sichern. Hierfür rufen Sie folgende Befehle auf:

```
$ make linux-savedefconfig
```

```
$ cp build/linux-c6d86…/Kconfig ../custom/linux.config
```

Diesen Tanz wiederholen Sie bei jeder Änderung der Kernel-Konfiguration. Den Pfad beim Kopieren müssen Sie natürlich ausschreiben. Anschließend starten Sie die reguläre Buildroot-Konfiguration und setzen darin folgende Parameter:

- *Kernel → Kernel configuration:* Using a custom (def)config file
- *Kernel → Configuration file path:* `$(BR2_EXTERNAL_DHBW_PATH)/linux.config`

Sollten Sie später wieder auf die Standardkonfiguration zurückkehren wollen, tragen Sie stattdessen folgende Werte ein:

- *Kernel → Kernel configuration:* Using an in-tree defconfig file
- *Kernel → Defconfig name:* `bcm2709`

Für Busybox ist das Vorgehen prinzipiell identisch. Im letzten Schritt befindet sich die relevante Buildroot-Option jedoch an folgender Stelle. Da die DHBW-Vorlagen jedoch bereits eine angepasste Busybox-Konfiguration beinhalten, müssen Sie diese nicht anpassen.

Target packages → BusyBox configuration file to use? `$(BR2_EXTERNAL_DHBW_PATH)/busybox.config`

3.14 Anpassen der Linux-Bootparameter

Dem Linux-Kernel können beim Starten verschiedene Parameter mitgegeben werden, um das Verhalten des Betriebssystems zu beeinflussen. Beispielsweise bekommt der Kernel oftmals den Namen der Root-Partition als Parameter übergeben, damit er diese während dem Startvorgang mounten und somit nutzbar machen zu können. Wo die Parameter einzutragen sind hängt dabei von der jeweiligen Hardware-Plattform sowie dem verwendeten Boot-Loader ab. Im Falle des Rasperry Pi ist es aber ganz einfach. Dort befindet sich auf der ersten Partition der SD-Karte eine Datei namens `cmdline.txt`. Sie enthält genau eine Zeile mit den an den Kernel zu übergebenen Bootparametern.

```
dennis@metropolis:/run/media/dennis/0FBE-CE26$ more cmdline.txt
root=/dev/mmcblk0p2 rootwait console=tty1 console=ttyAMA0,115200
```

Wie Sie sehen bekommt der Kernel hier folgende Parameter übergeben:

root=/dev/mmcblk0p2	Teilt dem Kernel mit, dass die zweite Partition (`p2`) der SD-Karte (`/dev/mmcblk`) das Dateisystem mit dem Root-Verzeichnis beinhaltet.
rootwait	Weist den Kernel an, beim Einhängen der Root-Partition ohne Timeout auf die SD-Karte zu warten.
console=tty1	Sorgt dafür, dass alle Kernel-Meldungen auf der ersten virtuellen Konsole ausgegeben werden. Deshalb sehen Sie so viele Meldungen vorbeiziehen, während das System startet.
console=ttyAMA0,115200	Weist den Kernel an, seine Meldungen mit einer Baudrate von 115200 Bits / Sekunde auf dem Serial Port auszugeben. Diese Möglichkeit wird insbesondere von eingebetteten Systemen genutzt, um die Kernelmeldungen verfügbar zu machen, ohne Sie auf dem Display des Geräts anzuzeigen.

Eine gute Übersicht der möglichen Kernel-Parameter finden Sie zum Beispiel auf redsymbol.net[48]. Ein weiterer nützlicher Parameter könnte in diesem Zusammenhang auch `quiet` sein. Durch ihn unterdrücken Sie sämtliche Kernel-Meldungen bis auf wirklich schwerwiegende Fehler, wodurch das System auch ein wenig

48 http://redsymbol.net/linux-kernel-boot-parameters/

schneller startet. Um ihn zu ergänzen, hängen Sie ihn einfach durch ein Leerzeichen getrennt an die Zeile in der `cmdline.txt`-Datei an:

```
dennis@metropolis:/run/media/dennis/0FBE-CE26$ more cmdline.txt
root=/dev/mmcblk0p2 rootwait console=tty1 console=ttyAMA0,115200 quiet
```

Allerdings hat dieses Vorgehen den Nachteil, dass die Änderung bei jeder Neuerstellung der Firmware verloren geht. Sie müssten die Änderung daher jedes mal wiederholen, wenn Sie ein neues Firmeware-Image bauen und auf die SD-Karte schreiben. Nachfolgend soll daher gezeigt werden, wie Sie die Änderung in den Build-Prozess von Buildroot integrieren können. Leider ist dies etwas aufwändig, da hierfür überraschenderweise keine einfache Möglichkeit in Buildroot vorgesehen ist. Sehr schwer ist es aber nicht.

Zunächst müssen Sie verstehen, wie nach dem Zusammenstellen des Wurzeldateisystems das finale Abbild der SD-Karte erzeugt wird. Hierfür ist die folgende Konfigurationsoption zuständig:

System configuration → Custom scripts to run after creating filesystem images

Wie Sie sehen, ist dort der Wert `board/raspberrypi3/post-image.sh` voreingestellt, wobei sich der Pfad auf das Buildroot-Verzeichnis bezieht. Der vollständige Pfad lautet daher `~/buildroot/board/raspberrypi3/post-image.sh`. Wenn Sie sich das Skript dabei anschauen, fällt Ihnen am Ende des Skripts sicher folgender Befehl auf:

```
genimage                           \
        --rootpath "${TARGET_DIR}"     \
        --tmppath "${GENIMAGE_TMP}"    \
        --inputpath "${BINARIES_DIR}"  \
        --outputpath "${BINARIES_DIR}" \
        --config "${GENIMAGE_CFG}"
```

Dieser Befehl ist es, der die fertige `sdcard.img`-Datei erzeugt. `${TARGET_DIR}` verweist dabei auf das Verzeichnis mit dem assemblierten Wurzeldateisystem (Partition 2 auf der SD-Karte). `${BINARIES_DIR}` hingegen ist das Verzeichnis, in dem die Dateien für die Boot-Partition abgelegt werden, indem sich auch die Datei `cmdline.txt` befindet. Die Idee, um den Inhalt dieser Datei anzupassen, ist daher, ein eigenes Skript zu schreiben, das vor dem bereits hinterlegten Skript läuft und die `cmdline.txt` für uns anpasst.

Legen Sie hierfür im `custom`-Verzeichnis eine neue Datei mit folgendem Inhalt an und sorgen Sie dafür, dass die Datei ausführbar ist. Den genauen Inhalt müssen Sie natürlich anpassen, je nachdem welche Bootparameter Sie an den Kernel übergeben möchten. Achten Sie beim Abschreiben auch auf die genaue Schreibweise und dabei insbesondere auf falsch platzierte Leerzeichen, da das Skript sonst nicht funktioniert.

```
buildroot@debian:~/custom$ nano cmdline.sh
#! /bin/sh
echo ">> Füge weitere Bootparameter in der cmdline.txt hinzu"

# Einlesen der Bootparameter aus cmdline.txt
CMDLINE_TXT="$BINARIES_DIR/rpi-firmware/cmdline.txt"
BOOT_PARAMS=""

while read line; do
    if [ -z "$BOOT_PARAMS" ]; then
        BOOT_PARAMS="$line"
    else
        BOOT_PARAMS="$BOOT_PARAMS $line"
    fi
done < "$CMDLINE_TXT"

# Fügen Sie hier nach Bedarf weitere Parameter hinzu
BOOT_PARAMS="$BOOT_PARAMS quiet"

# Veränderten Inhalt in cmdline.txt schreiben
echo "$BOOT_PARAMS" > "$CMDLINE_TXT"

buildroot@debian:~/custom$ chmod +x cmdline.sh
```

Anschließend starten Sie das Buildroot-Konfigurationsmenü und ändern Sie folgenden Wert, damit das neue Skript aufgerufen wird:

System configuration → Custom scripts to run after creating filesystem images:
`$(BR2_EXTERNAL_DHBW_PATH)/cmdline.sh board/raspberrypi3/post-image.sh`

Anschließend bauen Sie ein neues Firmware-Image. Wenn alles geklappt hat, beinhaltet die `cmdline.txt` auf der ersten Partition nun die veränderten Bootparameter.

3.15 Zusätzlichen Platz im Dateisystem reservieren

Buildroot bemisst das Root-Dateisystem der Firmware in der Regel sehr knapp, so dass kaum Platz für nachträglich hinzugefügte oder während der Laufzeit des Endgeräts erzeugte Dateien übrig bleibt. Um dem abzuhelfen, gibt es folgende Einstellung:

Filesystem images → Extra size in blocks

Hier können Sie eine Anzahl in Blöcken eintragen, die zusätzlich zum mindestens benötigten Platz im Dateisystem freigehalten werden. Ein Block entspricht dabei in der Regel 1 kB, so dass 1024 Blöcke 1 MB entsprechen. Wollen Sie daher 100 MB zusätzlichen Platz beanspruchen, tragen Sie hier einfach `102400` ein.

4 Der Linux-Werkzeugkasten

4.1 Konfiguration des Kabelnetzwerks (in Arbeit)

Das mit Buildroot erstelle Linuxsystem beinhaltet bereits eine minimale Netzwerkkonfiguration, so dass das kabelgebundene Netzwerk beim Hochfahren aktiviert und via DHCP konfiguriert wird. Sofern es in Ihrem Netzwerk keine Zugangsbeschränkungen gibt, sollte der Raspberry Pi daher eine automatische IP-Adresse zugewiesen bekommen und auf das Internet zugreifen können. Zuhause in den heimischen vier Wänden ist das so gut wie immer genau was man will, da die meisten Heimrouter auch einen DHCP-Server beinhalten und sich das gesamte Netzwerk damit quasi von alleine konfiguriert. Und auch im geschäftlichen Umfeld erfolgt die Netzwerkkonfiguration oftmals durch einen DHCP-Server. Nachteilhaft an der Sache ist jedoch, dass sich die IP-Adresse bei jedem Neustart ändern kann und Sie daher immer erst herausfinden müssen, welche IP-Adresse der Raspberry Pi gerade besitzt, bevor Sie über das Netzwerk auf ihn zugreifen können.

Eine Möglichkeit, dieses Problem zu lösen, besteht darin, die Konfiguration des Routers anzupassen, so dass der Raspberry Pi immer dieselbe IP-Adresse zugewiesen bekommt. Falls das aber aus irgend einem Grund nicht möglich ist, oder Sie den Raspberry Pi über ein LAN-Kabel direkt mit Ihrem Entwicklungsrechner verbinden wollen, können Sie stattdessen auch auf die Autokonfiguration via DHCP verzichten und eine statische IP-Adresse in der Linux-Firmware einstellen. Auch eine Mischung aus beidem ist möglich, so dass das Raspberry Pi sowohl eine statische als auch eine dynamische IP-Adresse besitzt.

Wie funktioniert die Standardkonfiguration? (etc/inittab → ifup/ifdown → /etc/network/interfaces)

Auszüge aus den beiden Daten /etc/inittab und /etc/network/interfaces

Wie man eine statische IP konfiguriert (IP, Netmask, Gateway)

Was bedeuten die drei Werte?

Verweis auf Start/Stop-Skripte und ip/ifconfig-Kommandos

4.2 (Konfiguration des WLANs)

4.3 (Der Raspberry Pi als WLAN Access Point)

4.4 Datum und Uhrzeit aus dem Internet beziehen

Im Gegensatz zu anderen Computern besitzt der Raspberry Pi keine Real Time Clock, weshalb er sich das aktuelle Datum und die Uhrzeit nicht merken kann, wenn er vom Strom getrennt ist. Als Folge daraus nimmt Linux bei jedem Systemstart den 01.01.1970, 0:00 Uhr als aktuelle Systemzeit an.[49] Dies hat natürlich zur Folge, das Zeitstempel im Dateisystem oder in anderen Protokollen völlig falsche Werte erhalten. Aber neben diesem Ärgerniss hat es noch viel schwerwiegendere Folgen: Der Raspberry Pi kann keine Netzwerkverbindung zu mit SSL gesicherten Endpunkten aufbauen und daher zum Beispiel auch keine HTTPS-URLs aufrufen. Denn diese nutzen in der Gültigkeit begrenzte SSL-Zertifikate, um die Authentizität der Gegenstelle zu bestätigen und auch die für die Verschlüsselung benötigten Keys auszutauschen. Und da Linux eben von einem völlig falschen Datum ausgeht, lehnt es jeden Verbindungsaufbau mit den Hinweis ab, dass das vom Server empfangene Zertifikat noch nicht gültig sei.

49 Nicht, dass es zu dieser Zeit schon Linux gab, jedoch handelt es sich dabei um eine Konvention, die Linux von den älteren UNIX-Systemen übernommen hat. Denn diese speichern Datum und Uhrzeit nicht etwas als eine Folge von Ziffern entsprechend Tag, Monat Jahr, Stunden, Minuten und Sekunden sondern als die Anzahl Sekunden seit dem 01.01.1970, 0:00 Uhr. Dieses Datum wird daher auch der Beginn der UNIX-Epoche genannt.

Die Lösung des Problems ist allerdings ganz einfach und nennt sich NTP (Network Time Protocol) und ist so eine Art Funkuhr für Internetteilnehmer. Alles was Sie tun müssen, ist einen NTP-Client in die Firmware zu integrieren und beim Hochfahren des Systems automatisch zu starten. Dadurch kann der Raspberry Pi dann das aktuelle Datum und die aktuelle Uhrzeit beim Hochfahren ermitteln und alle Probleme sind beseitig. Zumindest so lange, wie der Raspberry Pi über eine funktionierende Internetverbindung verfügt.

Die von Buildroot verwendete BusyBox-Werkzeugsammlung beinhaltet bereits einen NTP-Client, dieser wird in der Standardkonfiguration allerdings nicht kompiliert und daher auch nicht in die Firmware integriert. Um ihn zu aktiveren müssen Sie daher die BusyBox-Konfiguration ändern. Doch zunächst überprüfen Sie in Buildroot folgende Einstellung, um sicherzugehen, dass die veränderte BusyBox-Konfiguration auch tatsächlich übernommen wird:

Target packages → BusyBox configuration file to use?

Hier muss `$(BR2_EXTERNAL_DHBW_PATH)/busybox.config` eingetragen sein.

Anschließend speichern Sie die Buildroot-Konfiguration und geben zurück auf der Konsole folgenden Befehl ein, woraufin sich das BusyBox-Konfigurationsmenü öffnet:

```
$ make busybox-menuconfig
```

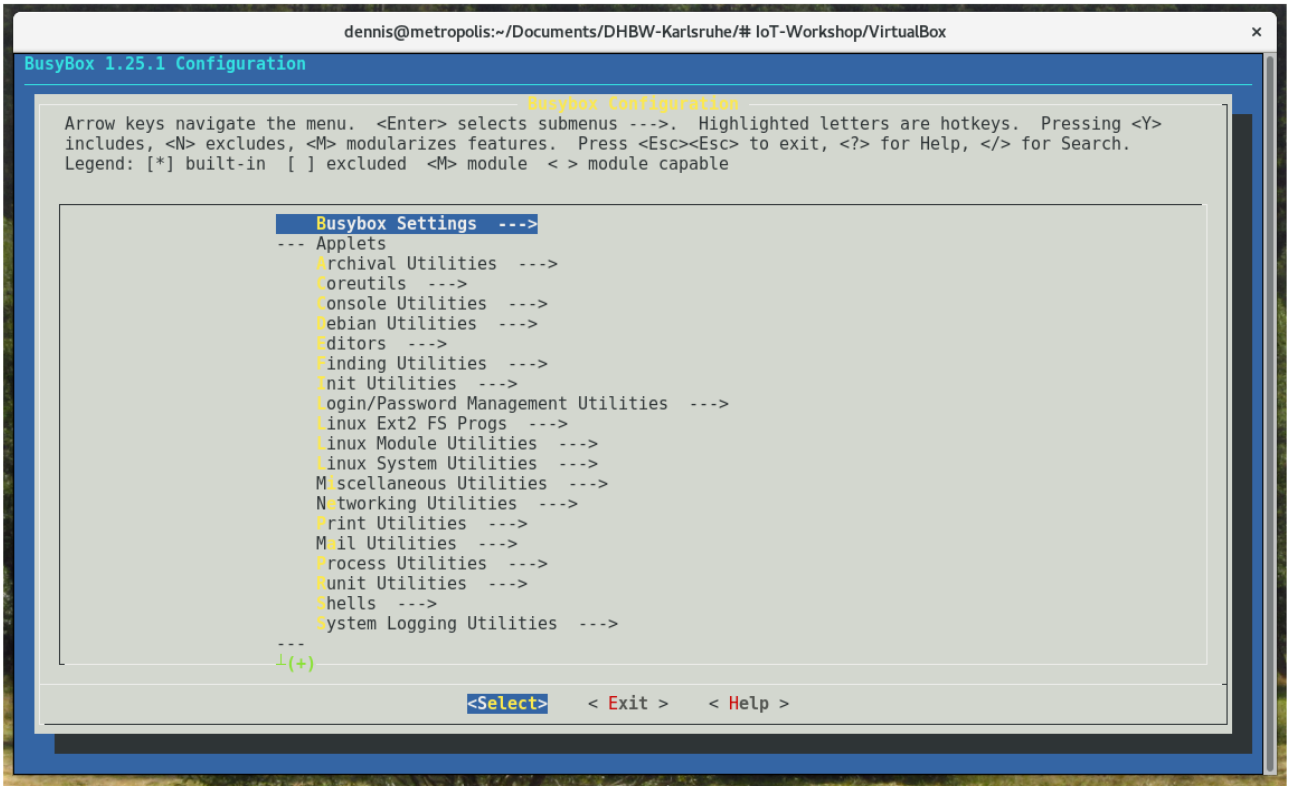

Abb. 22: Sieht dem Buildroot-Menü sehr ähnlich: Das BusyBox-Konfigurationsmenü

In der BusyBox-Konfiguration aktiveren Sie dann folgende Einstellung:

Networking Utilities → ntpd

Die daraufhin erscheinenden Optionen *Make ntpd usable as a NTP server* und *Make ntpd understand /etc/ntpd.conf* können Sie hingegen deaktivieren. Anschließend kehren Sie zur obersten Menüebene zurück und wählen dort den Eintrag `Save Configuration to an Alternate File` aus. Als Wert tragen Sie den Pfad `/home/buildroot/custom/busybox.config` ein. Danach können Sie das Menü beenden.

Anschließend bearbeiten Sie die Datei `~/custom/board/rootfs_overlay/etc/network/interfaces`, um sicherzustellen, dass der NTP-Client automatisch gestartet wird, sobald das Kabelnetzwerk verfügbar ist. Dieselbe Datei können Sie auch bearbeiten, wenn Sie den NTP-Client über WLAN starten wollen.

```
buildroot@debian:~/custom/board/rootfs_overlay$ nano etc/network/interfaces
…
iface eth0 inet dhcp
    post-up sleep 5; /sbin/ntpd -p pool.ntp.org
    post-down killall ntpd
…
```

Danach müssen Sie nur noch die Firmware neu bauen. Durch einen simplen Aufruf des `date`-Befehls können Sie dann Datum und Uhrzeit auf dem Raspberry Pi überprüfen. Dieses sollte im Vergleich zu vorher nicht mehr ganz so drastisch abweichen.

```
$ date
```

4.5 (Ein einfacher HTTP-Server für statische Dateien)

4.6 (Darf es ein bisschen mehr sein? Der Apache-Webserver)

4.7 (Remote Login und sicherer Dateitransfer mit SSH)

4.8 (Musik abspielen mit VLC)

4.9 (Ein einfacher Splash-Screen)

4.10 (Nutzung der GPIO-Pins in der Konsole)

5 Grafische Benutzeroberflächen

5.1 Wie die Grafikausgabe unter Linux funktioniert

Wie jedes moderne Betriebssystem besitzt auch Linux eine grafische Benutzeroberfläche, die eine komfortable Bedienung des Systems ermöglicht. Im Gegensatz zu vielen anderen Betriebssystemen wie Windows oder macOS ist die Benutzeroberfläche aber nicht untrennbar mit dem Betriebssystem verbunden, weshalb es eigentlich komplett falsch ist, von *der* Linux-Benutzeroberfläche zu reden. Im engeren Sinne handelt es sich bei Linux ja auch nur um den Kernel, der zwar die technischen Voraussetzungen zum Ansprechen der verschiedenen Ein- und Ausgabegeräte bietet, selbst aber keine eigene Benutzeroberfläche enthält. Diese wird stattdessen von zusätzlichen Programmen und Bibliotheken bereitgestellt, die hierfür auf den vom Kernel bereitgestellten Funktionen aufbauen. Und wie in allen anderen Bereichen auch ist Linux hier sehr flexibel. Das macht es am Anfang schwer durchzublicken, weil es für jede Komponente mehrere Alternativen gibt. Allerdings ist der Grafik Stack von Linux nicht wirklich komplizierter als der von Windows oder macOS. Der Unterschied ist einfach nur, dass man sich mit Linux sein System selbst zusammenstellen kann, was bei den anderen beiden nicht vorgesehen ist. Welche Möglichkeiten es hierbei gibt, zeigt die folgende Abbildung:

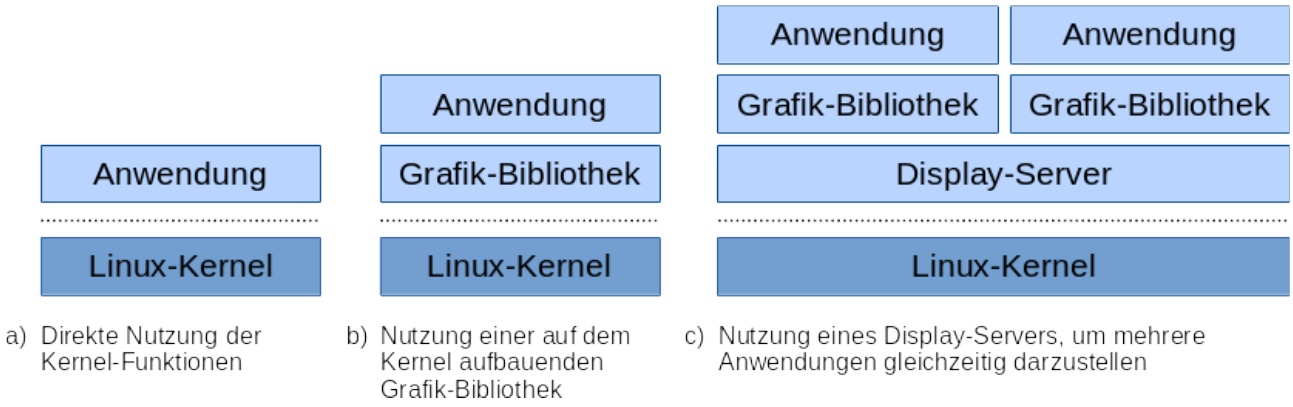

Abb. 23: Vereinfachte Darstellung, wie grafische Ausgaben unter Linux realisiert werden können

Im einfachsten Fall nutzt eine Anwendung direkt die Funktionen des Linux-Kernels, um etwas auf dem Bildschirm anzuzeigen. In diesem Fall bekommt eine einzelne Anwendung quasi exklusiven Zugriff auf den Bildschirm, da der Kernel keine Möglichkeit vorsieht, mehrere Programme gleichzeitig darzustellen. Dies hat den Vorteil, dass das Gesamtsystem sehr schlank ist, die Programmierung ist dafür aber umso aufwändiger. Denn die Grafikfunktionen des Kernels sind sehr hardwarenah ausgelegt, weshalb selbst einfache Dinge, wie das Zeichnen einer Linie, komplett selbst entwickelt werden müssen. Und auch die Tatsache, dass es inzwischen zwei offizielle Grafik-Subsysteme gibt, macht es eher noch komplizierter:

- **Framebuffer:** Beim Framebuffer handelt es sich um die ältere und einfachere API. Sie beinhaltet grundlegende Funktionen, um den Bildschirm als zwei-dimensionales Array von Bildpunkten darzustellen. Grafikbeschleunigung wird nur rudimentär unterstützt.
- **KMS/DRM:** Kernel Mode Setting und Direct Render Manager sind der moderne Nachfolger des Framebuffers. Sie sind direkt darauf ausgelegt, die Möglichkeiten moderner Grafikprozessoren zu nutzen, die Programmierung ist dafür aber sehr komplex.

Zur Vereinfachung der Programmierung bietet es sich daher an, eine Bibliothek zu nutzen, durch die die Komplexität des Kernels versteckt wird und die darauf aufbauend grundlegende Zeichenfunktionen zur Verfügung stellt. Je nach verwendeter Programmiersprache könnten dies zum Beispiel folgende sein:

Programmiersprache	Bibliothek	Beschreibung
Java	JavaFrameBuffer[50]	Bietet nur sehr rudimentären Zugriff auf den Framebuffer und scheint auch nicht wirk-

		lich weiterentwickelt zu werden. Immerhin können aber mit `java.awt.Graphics2D` einfache geometrische Formen und Textliterale gezeichnet werden. Ein Beispiel befindet sich in der Klasse `java.org.tw.pi.frameBuffer.TestFrameBuffer`.
Python	pygame[51]	Bei pygame handelt es sich um Python-Bindings der SDL-Bibliothek. Wie der Name andeutet, ist sie vorwiegend zur Entwicklung von Spielen gedacht. Sie kann aber auch ganz allgemein dafür verwendet werden, um Tastatur, Maus, Bildschirm und Soundkarte anzusteuern. Wenn Ihre Anwendung kein Widget Toolkit à la Swing oder GTK+ benötigt, könnte dies eine interessante Möglichkeit sein.
C/C++	SDL[52]	Wenn Sie nicht vor C oder C++ zurückschrecken, können Sie die SDL-Bibliothek natürlich auch direkt verwenden. Die Nutzung ist nicht wirklich schwieriger als bei pygame, jedoch ist die Integration in die Firmware aufwändiger.
C/C++, Python, …	Cairo[53]	Cairo ist eigentlich eine auf 2D-Zeichnungen ausgelegte API ähnlich wie die Javaklasse `Graphics2D` oder in HTML5 das Canvas-Objekt. Mit etwas Boilerplate-Code kann Cairo direkt in den Linux-Framebuffer zeichnen.[54]
C/C++, Python, …	DirectFB	Eine ältere API, die für die eingebetteten Systeme von Fernsehern entwickelt wurde. Ermöglicht u.a. auch die Darstellung mehrerer Anwendungen gleichzeitig.
C/C++, Python, …	QT	QT ist neben GTK+ eines der beiden großen Widget Toolkits unter Linux. Im Gegensatz zu GTK+ beinhaltet QT aber mehr Funktionen für die Entwicklung eingebetteter Systeme. Beispielsweise bietet es die Möglichkeit, seine Ausgabe direkt in den Linux-Framebuffer zu schreiben.

In vielen Fällen werden Sie aber nicht um einen ausgewachsenen Display Server herumkommen. Und zwar immer dann, wenn Sie eine der folgenden Anforderungen haben:

- Sie wollen mehrere Anwendungen gleichzeitig auf dem Bildschirm darstellen.
- Sie wollen eine vorhandene Anwendung oder Bibliothek nutzen, die nicht vorwiegend für eingebettete Systeme entwickelt wurde.
- Sie wollen das System mehr in Richtung einer Desktopumgebung mit frei beweglichen Fenstern, einer Taskleiste usw. ausbauen.

Der Display Server ist dabei ein spezielles Programm, das eine eigene Schnittstelle für den Zugriff auf den Bildschirm sowie die Eingabegeräte bietet. Im Gegenzug dafür stellt er grundlegende Fensterfunktionen zur Verfügung, so dass sich mehrere Anwendungen den Bildschirm teilen können. Technologisch nähern Sie sich damit den großen Linux-Systemen für Desktopcomputer, Smartphones oder Tablets. Doch wie es der Teufel so will, haben Sie auch hier die Wahl:

- **X11:** Das X Window System ist sozusagen das kampferprobte Schlachtschiff, das seit 1984 auf allen Unix-Systemen daheim ist. Auch unter Linux laufen die meisten Desktopumgebungen traditionell unter dem X-Server, in jüngerer Zeit wird er aber immer mehr von Wayland verdrängt.
- **Wayland:** Wayland ist ein moderner Nachfolger von X11, der X11 im Embedded-Bereich bereits weitgehend verdrängt hat. Aber auch auf dem Desktop wechseln immer mehr Systeme zu Wayland.
- **Mir:** Mir war Canonicals Versuch, einen X11-Nachfolger für Ubuntu zu entwickeln, da man Wayland für die eigenen Bedürfnisse nicht als hinreichend befand. Im April 2017 wurde jedoch der Rückzug aus der Entwicklung bekannt gegeben. Da Mir nur in Ubuntu Phone zum Einsatz kam, kann man davon ausgehen, dass es nun nicht mehr weiterentwickelt wird. Es ist daher auch nicht in Buildroot enthalten.

50 https://github.com/ttww/JavaFrameBuffer
51 https://www.pygame.org/
52 https://www.libsdl.org/
53 https://cairographics.org/
54 https://lists.cairographics.org/archives/cairo/2010-July/020378.html

Da Mir keine wirkliche Alternative ist, haben Sie die Wahl zwischen X11 und Wayland, wobei es sich bei beiden streng genommen nur um Spezifikationen handelt, zu denen es selbst wieder mehrere Implementierungen gibt. Aus diesem Grund ist X11 in Buildroot gleich in Form zwei verschiedener (aber kompatibler) X-Server enthalten, während für Wayland derzeit nur die Referenzimplementierung namens Weston angeboten wird. Welche Vor- und Nachteile die jeweiligen Plattformen haben, zeigt folgende Tabelle:

Display Server	Vorteile	Nachteile
X11	+ Große Softwareauswahl in Buildroot enthalten + Zwei Implementierungen verfügbar: • **X.org:** Großer X-Server mit 3D-Beschleunigung von vollem Funktionsumfang • **TinyX:** Kleiner X-Server für den Framebuffer + Sehr flexible Konfiguration möglich + Modularer Aufbau mit mehreren Komponenten: • **X-Server:** Beinhaltet nur grundlegende Funktionen zur Darstellung der Anwendungen • **Window Manager (optional):** Zeichnet um jedes Fenster den Fensterrahmen und Titelleiste und kümmert sich um die Platzierung der Fenster. • **Compositor (optional):** Ermöglicht die Nutzung moderner Grafikbeschleuniger für die Zusammensetzung des finalen Bildes. + Kann ohne Window Manager betrieben werden wenn nur eine statische Anordnung von ein paar Fenstern ohne Rahmen benötigt wird. Allerdings wird das nicht von jeder Anwendung unterstützt.	− Jede Menge Altlasten, die nicht mehr relevant sind − Daher viele veraltete Dokumentationen im Netz − Teilweise komplexe Konfiguration nötig − Mehr Overhead, geringere Performance als Wayland
Wayland	+ Moderner, performanter Nachfolger für X11 + Deutlich einfachere Architektur als bei X11: Display Server, Window Manager und Compositor stecken in ein und demselben Programm + Daher wesentlich performanter als X11 + Leichter zu konfigurieren + Auch ausgefallene Display Server möglich, welche die Fenster z.B. in einem 3D-Raum anordnen + X11-Anwendungen können dank Xwayland auch in Zukunft weitergenutzt werden	− Technologie im Vergleich zu X11 noch relativ jung − Daher auch weniger brauchbare Implementierungen − Kaum Dokumentationen im Netz − Buildroot enthält nur die „Weston“ genannte Referenzimplementierung, diese ist aber nur bedingt für den Produktiveinsatz gedacht − Generell im Vergleich zu X11 weniger Programme in Buildroot verfügbar − Weston unterstützt keine Vorgabe der Größe und Position eines Fensters beim Starten, diese Funktion muss jede Anwendung selbst implementieren

Wenn Sie die obige Abbildung genauer anschauen, stellen Sie sicher fest, dass auch bei Alternative c jede Anwendung eine Grafik-Bibliothek für ihre Bildschirmausgabe benutzt. Der Grund ist dabei derselbe wie der Variante ohne Display-Server: Zwar bieten die Display-Server grundlegende Funktionen, um Pixel darzustellen, für alles weitere ist aber die Anwendung selbst zuständig. Bis auf ein paar Testprogramme gibt es daher praktisch keine einzige Anwendung, die nicht eine der vielen verfügbaren Bibliotheken nutzt. Und hier ist wirklich für jeden Geschmack etwas dabei. Die nachfolgende Liste ist daher nur ein Auszug der wichtigsten Vertreter:

Programmiersprache	Bibliothek	Beschreibung
Java	Swing	Swing kennen Sie aus den Java-Vorlesungen am Anfang Ihres Studiums. Es handelt sich dabei um das klassische Widget-Toolkit für Java, jedoch sind damit einige moderne Bedienkonzepte nur umständlich realisierbar.
Java	JavaFX	Der designierte Nachfolger von Swing ermöglicht eine elegante und moderne Programmierung. Ähnlich wie mit HTML können die Benutzeroberflächen in einer XML-ähnlichen Syntax beschrieben werden. Da die API intern auf einem Scene Graph basiert, sind auch animierte Benutzeroberflächen und grafische Effekte möglich. Leider wird JavaFX jedoch von Oracle nicht mehr offiziell für ARM-Prozessoren weiterentwickelt, weshalb man hier auf die Community angewiesen ist.
C/C++, Python, …	GTK+	Das andere große Widget Toolkit unter Linux neben QT. GTK+ beinhaltet alles was man braucht, um moderne Fensteranwendungen zu schreiben. Im Vergleich zu QT ist es schlanker und einfacher aufgebaut, bietet dafür aber an manchen Stellen weniger Funktionen.
C/C++, Python, …	QT	QT haben wir oben schon einmal gesehen. Natürlich lassen sich mit QT erstellte Anwendungen auch mit einem Display Server nutzen.
C/C++, Python, …	Clutter	Clutter ist eine im GTK-Umfeld entstandene Scene Graph API. Damit lassen sich animierte Oberflächen erstellen, wie man sie von Smartphones oder Fernsehern her kennt. Falls klassische Widgets wie Buttons oder Eingabefelder benötigt werden, kann GTK+ mit Clutter integriert werden.
C/C++, Python, …	SDL bzw. pygame	Wie oben erwähnt ist SDL eher für Spiele gedacht, man kann damit aber auch ganz allgemein grafische Anwendungen schreiben. SDL besitzt dabei mehrere Backends und kann deshalb auch unter X11 oder Wayland genutzt werden.

5.2 Bevor Sie loslegen: Einschalten der 3D-Grafikbeschleunigung

Bevor Sie sich mit den oben beschriebenen Möglichkeiten zur Grafikausgabe weiter beschäftigen, sollten Sie erst sicherstellen, dass Linux den Grafikprozessor des Raspberry Pi und damit auch die 3D-Grafikbeschleunigung nutzen kann. Denn sonst steht Ihnen nur ein simpler Framebuffer zur Verfügung, der einerseits sehr langsam ist (da alle Grafikausgaben von der CPU berechnet werden) und andererseits auch nicht mehr von allen Bibliotheken unterstützt wird. Denn inzwischen bauen die meisten Display Server und Bibliotheken auf der neueren KMS/DRI-Infrastruktur auf, die ohne entsprechende Treiber jedoch nicht funktioniert. Es steht also außer Frage, dass Sie hierfür das richtige Paket in die Firmware integrieren müssen. Das gesuchte Zauberwort heißt hier *Mesa3D*, da dieses Paket neben einer OpenGL-Implementierung auch die Treiber für verschiedene Grafikprozessoren bereitstellt.[55] Letztlich müssen Sie in Buildroot also folgende Einstellungen aktiveren, wobei auch die Reihenfolge zu beachten ist:

- *Target packages → Graphic libraries and applications → mesa3d*
- *Target packages → Graphic libraries and applications → mesa3d → Gallium vc4 driver*
- *Target packages → Graphic libraries and applications → mesa3d → OpenGL EGL*
- *Target packages → Graphic libraries and applications → mesa3d → OpenGL ES*

Zusätzlich müssen Sie den *udev*-Dienst integrieren und beim Hochfahren des Systems starten. Andernfalls kann es vorkommen, dass Tastatur und Maus nicht richtig funktionieren. Aktiveren Sie deshalb auch folgende Option:

System configuration → /dev management: `Dynamic using devtempfs + eudev`

55 3D-Treiber sind unter Linux immer ein Problem, da die meisten Hersteller die Spezifikationen Ihrer Chips nicht offen legen. Aus diesem Grund gab es für den Raspberry Pi lange Zeit nur proprietäre 3D-Treiber, die aber wie das dann meistens so ist, irgendwann vom Hersteller nicht mehr gepflegt wurden.

Um den Dienst beim Hochfahren zu starten, fügen Sie die unten fett markierten Einträge in der Datei `custom/board/rootfs_overlay/etc/inittab` hinzu. Die zweite Zeile stellt dabei sicher, dass das Kernelmodul mit dem 3D-Treiber geladen wird:

```
buildroot@debian:~/custom/board/rootfs_overlay$ nano etc/inittab
…
::sysinit:/etc/init.d/S10udev start
::once:/sbin/modprobe -a vc4
::sysinit:/bin/hostname -F /etc/hostname
::sysinit:/sbin/syslogd
::sysinit:/sbin/klogd
…
```

Zusätzlich überprüfen Sie, ob in Buildroot die Option *System configuration → Custom scripts to run before creating filesystem images* auch das Skript `$(BR2_EXTERNAL_DHBW_PATH)/config.sh` aufgeführt ist. Es sollte unter dem Namen `config.sh` im `custom`-Verzeichnis liegen und folgenden Inhalt haben:

```
01  #! /bin/sh
02  echo ">> Nehme Änderungen an der config.txt vor"
03  CONFIG_TXT="$BINARIES_DIR/rpi-firmware/config.txt"
04
05  sed -i -e 's/gpu_mem_256=100/gpu_mem_256=128/g' "$CONFIG_TXT"
06  sed -i -e 's/gpu_mem_512=100/gpu_mem_512=128/g' "$CONFIG_TXT"
07  sed -i -e 's/gpu_mem_1024=100/gpu_mem_1024=128/g' "$CONFIG_TXT"
08
09  cat <<EOF >> "$CONFIG_TXT"
10
11  # Aktivieren der 3D-Beschleunigung. Zusätzlich muss in der /etc/inittab
12  # noch der Befehl "modprobe -a vc4" ausgeführt werden.
13  dtoverlay=vc4-fkms-v3d
14  EOF
```

Und zum Schluss installieren Sie noch PAM, damit Sie im Falle von Wayland oder X11 den Display Server nicht mit Rootrechten starten müssen:

Target packages → Libraries → Other → linux-pam

Im Verzeichnis `/usr/share/fonts` können auf der SD-Karte Schriftarten hinterlegt werden, die von allen grafischen Anwendungen genutzt werden können. Damit dies funktioniert, muss in Buildroot die fontconfig-Bibliothek als zu installierendes Paket aktiviert sein:

Target packages → Libraries → Graphics → fontconfig

Für das Handling der Eingabegeräte sollten dann noch *libinput* und *libxkbcommon* installiert werden:

- *Target packages → Libraries → Hardware handling → libinput*
- *Target packages → Libraries → Hardware handling → libxkbcommon*
- *Target packages → Graphic libraries and applications → xkeyboard-config*

5.3 Start eines HTML-Browsers im Vollbildmodus

Die Programmierung einer grafischen Oberfläche mit den in Kapitel 5.1 genannten Techniken und Bibliotheken mag zwar sicher sehr interessant und eine spannende Herausforderung sein, möglicherweise haben Sie aber auch schon mit dem Gedanken gespielt, einfach Ihr Vorwissen in Sachen HTML, CSS und JavaScript gewinnbringend einzusetzen, um sich nicht mit zu vielen neuen Technologien auseinandersetzen zu müssen. Und tatsächlich ist dies nicht nur möglich, es handelt sich dabei auch um einen modernen Trend, da immer mehr Entwickler dazu übergehen, das Web als eine Geräte und Betriebssysteme übergreifende Anwendungsplattform zu betrachten, die selbst zur Entwicklung eingebetteter Benutzeroberflächen eingesetzt werden

kann. WebOS[56], Firefox OS[57], Bada[58] oder Web Platform for Embedded[59] sind nur einige Beispiele aus der kommerziellen Praxis hierzu. Was liegt also näher, als dasselbe auch auf dem Rasbperry Pi zu versuchen?

Letztlich benötigen Sie nur zwei Sachen: Einen Webserver und einen im Vollbild gestarteten Webbrowser. Dabei sollten Sie auf den Webserver nicht verzichten, auch wenn ihre Benutzeroberfläche ohne serverseitiger Logik komplett aus HTML, CSS und clientseitigem JavaScript besteht, denn für den Browser macht es durchaus einen Unterschied, ob eine Webseite aus einer lokalen Datei heraus oder durch einen Webserver bereitgestellt wird. Im Zweifelsfall sind bestimmte JavaScript-Aktionen wie das dynamische Nachladen von Inhalten nur zulässig, wenn die Seite von einem Webserver abgerufen wurde. Welchen Webserver Sie hierfür verwenden, hängt natürlich von Ihrem Vorwissen, Vorlieben und den Projektanforderungen ab. Hier wollen wir uns daher erst mal nur auf den Browser konzentrieren.

Buildroot stellt Ihnen gleich drei verschiedene Browser zur Verfügung, die jedoch alle ihre Vor- und Nachteile haben.

- **Dillo[60]:** Schlanker und einfacher Browser, der selbst ohne Grafikbeschleunigung direkt im Framebuffer lauffähig ist. Dillo ist daher sehr unkompliziert in der Handhabung, jedoch werden aktuelle HTML5-Funktionen nicht unterstützt. Da die Entwicklung nur noch sehr langsam voran schreitet, ist unklar, ob sich der Browser jemals über HTML4 hinaus entwickeln wird.
- **qt-webkit-kiosk[61]:** Kleines C++-Programm, das um die Webkit Browser Engine herum geschrieben wurde. Es werden daher alle modernen Webstandards unterstützt. Webkit benötigt aber zwingend funktionierende 3D-Grafiktreiber und es sind einige Optionen in Buildroot zu setzen, damit der Browser korrekt funktioniert.
- **Midori:** Kompletter Webbrowser, der ebenfalls die Webkit Engine nutzt. Benötigt in jedem Fall einen Display Server, da die zugrunde liegende GTK Bibliothek dies voraussetzt.

Im folgenden soll nur *qt-webkit-kiosk* vorgestellt werden, da hiermit eine komplett HTML-basierte Benutzeroberfläche ohne den Overhead eines Display Servers realisiert werden kann. In Buildroot müssen hierfür neben den in Kapitel 5.2 beschriebenen allgemeinen Optionen zusätzlich noch folgende Optionen eingeschaltet werden:

- *Target packages → Graphic libraries and applications → Qt5*
- *Target packages → Graphic libraries and applications → Qt5 → eglfs support*
- *Target packages → Graphic libraries and applications → Qt5 → OpenGL API: OpenGL ES 2.0+*
- *Target packages → Graphic libraries and applications → Qt5 → fontconfig support*
- *Target packages → Graphic libraries and applications → Qt5 → GIF support*
- *Target packages → Graphic libraries and applications → Qt5 → JPEG support*
- *Target packages → Graphic libraries and applications → Qt5 → PNG support*
- *Target packages → Graphic libraries and applications → Qt5 → qt5svg*
- *Target packages → Graphic libraries and applications → qt-webkit-kiosk*
- *Target packages → Graphic libraries and applications → X.org X Window System*

56 https://de.wikipedia.org/wiki/Open_WebOS
57 https://de.wikipedia.org/wiki/Firefox_OS
58 https://de.wikipedia.org/wiki/Bada_(Betriebssystem)
59 https://github.com/WebPlatformForEmbedded
60 https://www.dillo.org
61 https://github.com/sergey-dryabzhinsky/qt-webkit-kiosk

Anschließend können Sie den Browser mit folgendem Befehl starten:

```
$ qt-webkit-kiosk --uri http://www.dhbw-karlsruhe.de
```

Im Falle eines lokalen Webservers wäre dass dann zum Beispiel:

```
$ qt-webkit-kiosk --uri http://localhost:8080
```

Und so sollte das Ergebnis im Idealfall aussehen:

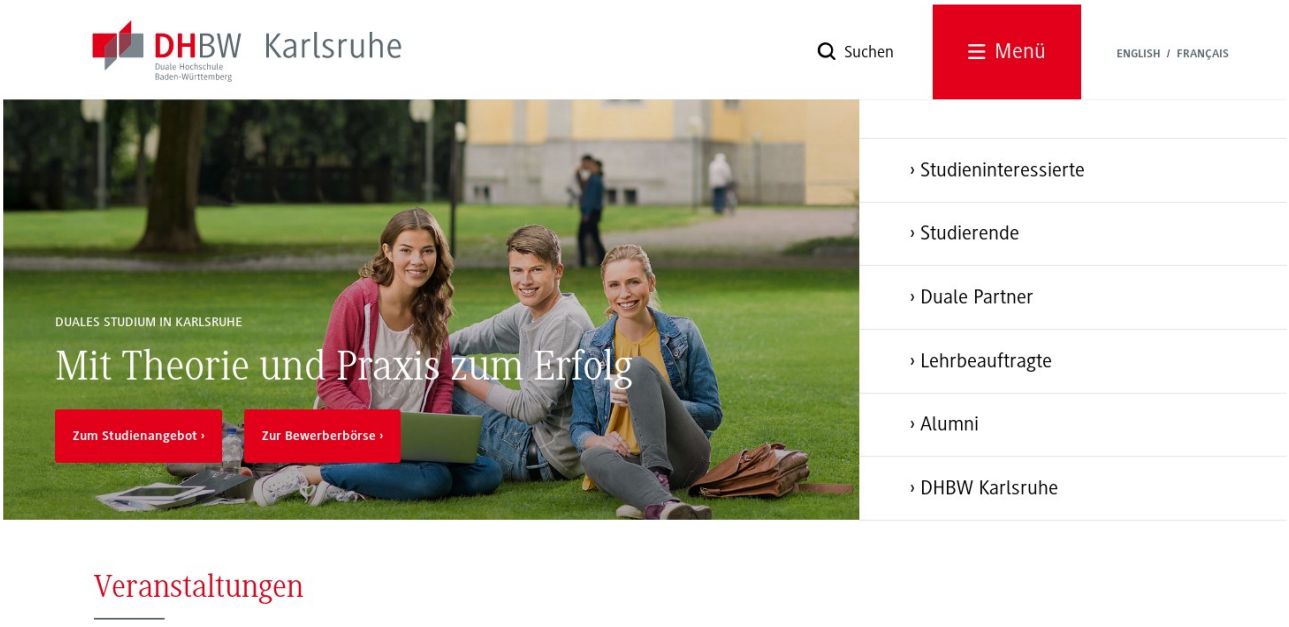

Abb. 24: Die Webseite der DHBW-Karlsruhe im Vollbild

Wenn Sie den Befehl ans Ende der Datei `/etc/inittab` anstelle der getty-Loginkonsole einbauen, können Sie den Browser damit auch direkt beim Hochfahren starten. Allerdings sollten Sie dabei die folgende Variante verwenden, damit der Browser nicht mit vollen Rootrechten ausgeführt wird:

```
buildroot@debian:~/custom/board/rootfs_overlay$ nano etc/inittab
…
#console::respawn:/sbin/getty -L console 0 vt100 # SERIAL
#tty1::respawn:/sbin/getty -L tty1 0 vt100 # HDMI
tty1::respawn:-su -l mulder -c "/bin/qt-webkit-kiosk --uri https://www.pingu-mobil.de/iot/"
```

Standardmäßig blendet *qt-webkit-kiosk* einen Mauszeiger ein, mit dem die Webseite bedient werden kann. Wenn Sie diesen verstecken wollen, können Sie dies durch eine eigene Konfigurationsdatei übersteuern. Als Vorlage können Sie die Datei `/usr/share/qt-webkit-kiosk/qt-webkit-kiosk.ini` verwenden, die Sie entweder auf der SD-Karte des fertigen Linux Images oder im Verzeichnis `~/make/target/usr/share/qt-webkit-kiosk` innerhalb der VM finden:

```
$ cp ~/make/target/usr/share/qt-webkit-kiosk/qt-webkit-kiosk.ini ⏎
  ~/custom/board/rootfs_overlay/etc
```

```
$ nano ~/custom/board/rootfs_overlay/etc/qt-webkit-kiosk.ini
```

Ganz am Ende der Datei ändern Sie `hide_mouse_cursor=false` in `hide_mouse_cursor=true`. Da sich dadurch der Pfad der Konfigurationsdatei ändert, rufen Sie den Browser von nun an über folgenden Befehl auf:

```
$ qt-webkit-browser --config /etc/qt-webkit-kiosk.ini --uri http://localhost:8080
```

5.4 Auf nach Weston: Integration von Wayland

Wenn Sie die in Kapitel 5.2 beschriebenen Einstellungen vorgenommen haben, lässt sich Wayland in Form seiner Referenzimplementierung namens Weston nun ganz einfach integrieren, indem Sie die folgenden Einstellungen in Buildroot aktivieren:

- *Target packages* → *Graphic libraries and applications* → *weston*
- *Target packages* → *Graphic libraries and applications* → *weston* → *DRM compositor*

Zusätzlich sollten Sie noch folgende Einstellungen in der genannten Reihenfolge aktivieren, damit Sie auch X11-Anwendungen unter Wayland nutzen können:

- *Target packages* → *Libraries* → *Graphics* → *libepoxy*
- *Target packages* → *Graphic libraries and applications* → *X.org X Window System*
- *Target packages* → *Graphic libraries and applications* → *weston* → *Xwayland support*

Im nächsten Schritt müssen Sie nun ein Skript schreiben, dass Weston zusammen mit den automatisch zu startenden Desktopanwendungen startet. Nennen Sie dieses Skript einfach `weston` und legen Sie es im Verzeichnis ~/`custom/board/rootfs_overlay/opt/wayland` ab. Es sollte folgenden Inhalt haben:

```
01  #! /bin/sh -i
02
03  export XDG_RUNTIME_DIR=~/.xdg
04  mkdir -p $XDG_RUNTIME_DIR
05  chmod 0700 $XDG_RUNTIME_DIR
06
07  weston-launch -- -c /opt/wayland/weston.ini --tty=1 &
08
09  # Warten, bis Wayland vollständig verfügbar ist
10  while true; do
11      sleep 1
12      if [ -e $XDG_RUNTIME_DIR/wayland-0 ]; then
13          break
14      fi
15  done
16
17  # Automatisch zu startende Anwendungen
18  weston-terminal &
19
20  # Sicherstellen, dass sich Wayland nicht aufhängt
21  while true; do
22      sleep 86400
23  done
```

Wenn Sie anstelle eines Terminals lieber andere Anwendungen zusammen mit dem Desktop starten wollen, tauschen Sie die Zeile 18 einfach aus. Ansonsten behalten Sie aber den Rest des Skrips auf jeden Fall bei, damit sich der Desktop nicht aufhängt, sobald das Skript zu Ende läuft.

Durch einen entsprechenden Eintrag in der Datei `~/custom/permissions` müssen Sie das Skript nun ausführbar machen (s. auch Kapitel 3.11):

```
/opt/wayland/weston   f      755    root   root   -      -      -      -      -
```

Als nächstes benötigen Sie eine Konfigurationsdatei für Weston. Nennen Sie diese `weston.ini` und legen Sie sie ebenfalls in ~/`custom/board/rootfs_overlay/opt/wayland` ab. Übernehmen Sie dabei zunächst folgenden Inhalt:

```
01  [core]
02  backend=drm-backend.so
03  modules=xwayland.so
04  idle-time=0
```

```
05
06  [shell]
07  background-image=/opt/wayland/bg.jpg
08  background-type=scale-crop
09  locking=false
10  num-workspaces=1
11
12  panel-location=top
13
14  [output]
15  transform=normal
16
17  [keyboard]
18  keymap_layout=de
```

Was die einzelnen Einstellungen bedeuten und welche Möglichkeiten es darüber hinaus gibt, ist in der man-Page zur weston.ini dokumentiert, die Sie auch im Netz finden.[62] Nachfolgend nur ein paar Hinweise, mit welchen Einstellungen Sie experimentieren können:

- `panel-location=none`, um das Panel am oberen Bildschirmrand auszublenden.
- Weglassen oder Verändern der `background-…`-Werte, um ein anderes Hintergrundbild zu setzen
- Hinzufügen von `[launcher]`-Abschnitten, um Shortcuts im Panel zu erzeugen

Insbesondere die letzte Option kann ganz interessant sein. Indem Sie der Konfigurationsdatei weitere Abschnitte wie den folgenden hinzufügen, können Sie Shortcuts anlegen, die oben im Panel angezeigt werden:

```
[launcher]
icon=/opt/wayland/icon.png
path=/bin/node /lib/node_modules/beispiel/index.js
```

Zum Schluss müssen Sie nur noch sicherstellen, dass der Display Server beim Hochfahren des Betriebssystems gestartet wird. Entsprechend der Beschreibung aus Kapitel 3.12 müssen Sie daher folgende Änderungen an der Datei `~/custom/board/rootfs_overlay/etc/inittab` vornehmen. Die nachfolgend fett markierten Zeilen kommentieren Sie durch ein vorangestelltes `#` aus und ergänzen stattdessen die Zeile direkt darunter, um das Skript `/opt/wayland/weston` zu starten. Dabei können Sie den Benutzernamen nach `su -l` natürlich gegen jeden Linux-Benutzer tauschen.

```
# Kommentieren Sie die folgenden zwei Zeilen aus, um den Login auf der Konsole
# zu unterbinden. Stattdessen fügen Sie weitere Zeilen hinzu, um die von Ihnen
# bereitgestellten Systemdienste und Programme zu starten.
#console::respawn:/sbin/getty -L console 0 vt100 # SERIAL
#tty1::respawn:/sbin/getty -L tty1 0 vt100 # HDMI
tty1::respawn:-su -l mulder -c /opt/wayland/weston
```

Im diesem Beispiel wird der Display Server unter dem Benutzer `mulder` gestartet. Damit dies klappt, muss der Benutzer in der Gruppe `weston-launch` enthalten sein. Dies können Sie durch eine Anpassung der Datei `~/custom/users` sicherstellen:

```
buildroot@debian:~/custom$ more users
mulder   -1   xfiles   -1   =xfiles   /home/mulder   /bin/sh   wheel,weston-launch   Fox Mulder
scully   -1   xfiles   -1   =xfiles   /home/scully   /bin/sh   wheel,weston-launch   Dana Scully
```

Wenn das allen geklappt hat, sollten Sie beim nächsten Hochfahren mit folgendem Bild begrüßt werden.

62 Und zwar hier: http://manpages.ubuntu.com/manpages/wily/man5/weston.ini.5.html

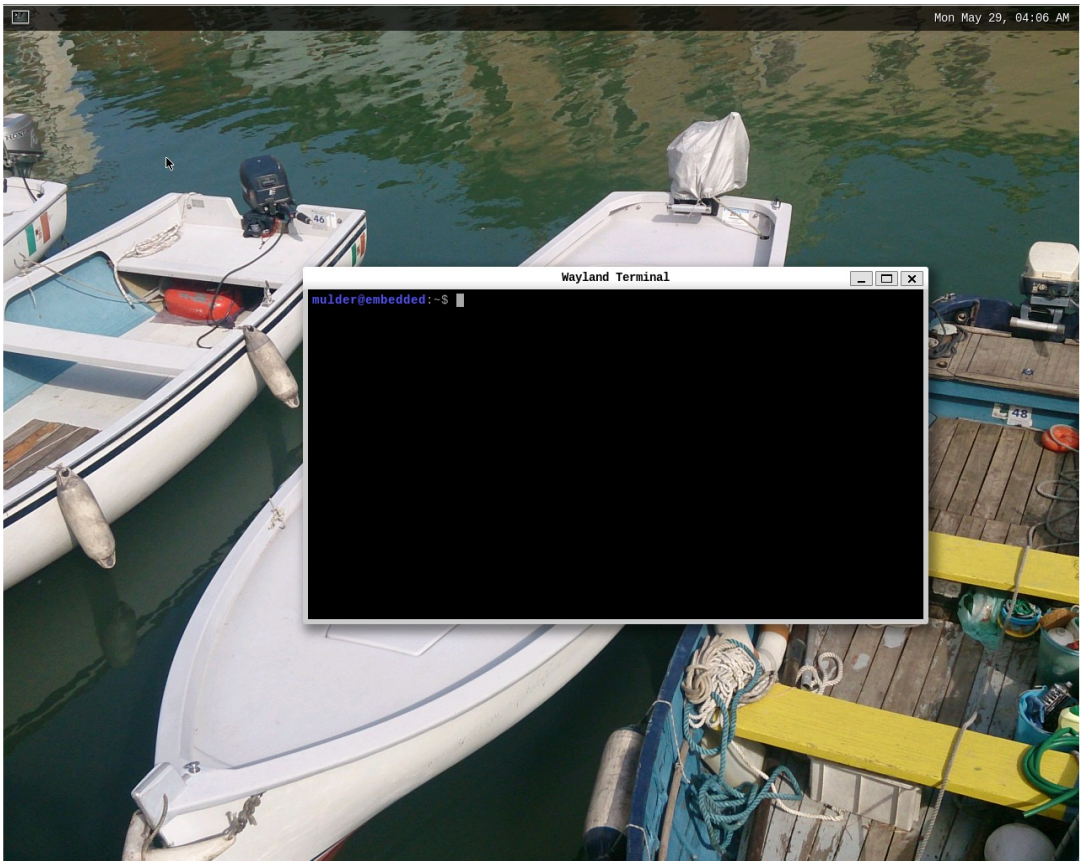

Abb. 25: Der Weston-Desktop mit einem geöffneten Fenster

5.5 Integration eines minimalen X-Servers

Wenn Sie nur eine Anwendung im Vollbildmodus starten wollen, Ihnen Wayland dafür zu kompliziert ist oder von der gewünschten Anwendung nicht unterstützt wird, können Sie auch mit relativ wenigen Handgriffen einen einfachen X-Server aufsetzen. In vielen Fällen können Sie dabei sogar den sonst für die Fensterverwaltung zuständigen Window Manager verzichten, wenn sich die Anwendung dann trotzdem im Vollbildmodus starten lässt. Nicht jede Anwendung unterstützt dies von alleine, der Window Manager lässt sich aber auch ganz einfach nachträglich hinzufügen, falls er doch benötigt wird. Zunächst wählen Sie in Buildroot folgende Einträge aus:

- *Target packages → Graphic libraries and applications → X.org X Window System*
- *Target packages → Graphic libraries and applications → X.org X Window System → X11R7 Servers → xorg-server*
- *Target packages → Graphic libraries and applications → X.org X Window System → X11R7 Servers → xorg-server → X Window System server type:* `Modular X.org`
- *Target packages → Graphic libraries and applications → nodm*

Auf *nodm* könnten Sie dabei sogar verzichten, allerdings müsste der X-Server dann aber mit vollen Rootrechten laufen, was aus Sicherheitsgründen keine gute Idee ist. nodm ist daher ein ganz einfacher Session Manager, der nichts anderes macht, als den X-Server zu starten und dann auf einen anderen Benutzer zu wechseln. Sie erreichen damit also, dass ein zuvor festgelegter Benutzer ohne Passwortabfrage automatisch angemeldet wird, sobald die grafische Umgebung startet.

Fügen Sie in der Datei `~/custom/board/rootfs_overlay/etc/base` folgende Zeile hinzu und kommentieren Sie stattdessen den automatischen Start der getty-Loginkonsole aus, um den X-Server beim Hochfahren automatisch zu starten:

```
buildroot@debian:~/custom/board/rootfs_overlay_base/etc$ nano inittab
::once:/sbin/nodm -stderr
#tty1::respawn:/sbin/getty -L tty1 0 vt100 # HDMI
#tty1::respawn:-su -l mulder -c /opt/wayland/weston
```

Anschließend legen Sie im selben Verzeichnis noch folgende Dateien wie gezeigt an:[63]

```
buildroot@debian:~/custom/board/rootfs_overlay_base/etc$ nano nodm.conf
NODM_USER=mulder
NODM_XSESSION=/opt/xinitrc

buildroot@debian:~/custom/board/rootfs_overlay_base/etc$ nano pam.d/nodm
#%PAM-1.0

auth      include   system-local-login
account   include   system-local-login
password  include   system-local-login
session   include   system-local-login
```

Dadurch haben Sie die grundlegende Konfiguration erstellt. Fehlt nur noch die Datei `/opt/xinitrc`, die Sie ebenfalls innerhalb des Overlayverzeichnisses anlegen müssen. Sie beinhaltet die zu startenden Programme, die zusammen mit der grafischen Oberfläche ausgeführt werden sollen. Legen Sie diese wie folgt an, wobei Sie *xclock* durch das eigentlich zu startende Programm ersetzen müssen. Sie können natürlich auch mehrere Programme nacheinander starten. Jedoch sollte nach jedem Befehl ein & stehen, um das Programm im Hintergrund zu starten. Denn sonst wartet das Skript erst, bis das jeweilige Programme beendet wurde, bevor die nächste Zeile ausgeführt wird:

```
01  #! /bin/sh
02  xclock &
```

5.6 Komplette Desktopumgebung auf Basis von X11

Das oben beschriebene X11-Minimalsetup lässt sich bei Bedarf nun ganz einfach zu einer vollwertigen Desktopumgebung ausbauen. Sie müssen lediglich noch ein paar mehr Programme starten, die die verschiedenen Aufgaben eines Desktops erfüllen. Das heißt, Sie müssen nur die Datei `/opt/xinitrc` anpassen, um einen Window Manager, einen Session Manager, ein Dock[64] und ggf. einen Dateimanager zu starten. Die verschiedenen Programme haben dabei folgende Aufgaben:

- **Window Manager:** Zeichnet den Rahmen und die Titelzeile um jedes Fenster und kümmert sich um die Anordnung und Verschiebbarkeit der Fenster.
- **Session Manager:** Verwaltet die zu startenden Programme und Dienste, wenn sich ein Benutzer in der grafischen Umgebung anmeldet. Da es für unser eingebettetes System aber ohnehin nur einen Benutzer geben kann, der automatisch angemeldet wird, können Sie auf den Session Manager verzichten und stattdessen einfach die Datei `/opt/xinitrc` um alle weiteren, zu startenden Programme erweitern.
- **Dock / Taskbar / Tray:** Unter Windows entspricht dies der Taskleiste, unter macOS am ehesten dem Dock am unteren Bildschirmrand. Hier werden also die ausgeführten Programme angezeigt und oftmals können über das Dock auch weitere Programme gestartet werden. Nicht jeder moderne Linux-Desktop besitzt heutzutage noch ein Dock, die Auswahl ist aber dennoch sehr groß.

63 Vgl. https://wiki.archlinux.org/index.php/Nodm
64 Unter Windows auch als Taskleiste bekannt

- **Dateimanager:** Diesen kennen Sie als den Explorer und Windows oder den Finder unter macOS. Eine Desktopumgebung wäre nicht komplett, hätte sie keinen ausgereiften Dateimanager an Board. Meistens zeichnet der Dateimanager auch den Desktophintergrund sowie die Icons, die direkt auf dem Desktop liegen.

Für jede dieser Komponenten gibt es unter Linux eine schier unendliche Auswahl. An dieser Stelle werden daher nur die *matchbox*-Programme vorgestellt, die speziell für eingebettete Systeme wie Handhelds oder PDAs entwickelt wurden. Ihre Bedienung wirkt zwar etwas altbacken, dafür sind sie aber einfach zu integrieren. Sie müssen lediglich in Buildroot noch folgende Optionen aktivieren:

- *Target packages → Graphic libraries and applications → matchbox*
- *Target packages → Graphic libraries and applications → matchbox → matchbox-desktop*
- *Target packages → Graphic libraries and applications → matchbox → matchbox-panel*

Anschließend erweitern Sie die Datei `/opt/xinitrc` so, dass folgende Programme gestartet werden:

```
01  #! /bin/sh
02  matchbox-window-manager &
03  matchbox-desktop &
04  matchbox-panel &
```

Und fertig ist die Desktopumgebung. Wenn Sie wollen, können Sie natürlich ein wenig in Buildroot stöbern und anstelle von Matchbox andere Window Manager und Werkzeugprogramme ausprobieren.

5.7 Grafische Ausgaben auf den Entwicklungsrechner umleiten

Die ersten Gehversuche bezüglich einer grafischen Ausgabe auf dem Raspberry Pi sollten Sie natürlich ganz klassisch mit einem angeschlossenen Bildschirm oder im Vorlesungsraum mit dem Beamer machen. Sobald Sie aber sicher sind, die grundlegenden Einstellungen korrekt vorgenommen zu haben, kann es aber auch Sinn machen, die Grafikausgabe des Raspberry Pi auf Ihren Entwicklungsrechner umzuleiten, um nicht so viel mitschleppen zu müssen oder vom Beamer abhängig zu sein. Leider gibt es jedoch keinen einheitlichen Weg, der in jedem Fall zum gewünschten Ergebnis führt. Viel mehr hängt es davon ab, für welchen der in Kapitel 5.1 vorgestellten Ansätze Sie sich entschieden haben:

- **Einfache Grafikausgabe über den Framebuffer ohne Grafikbeschleunigung:** x11vnc
- **Nutzung von X11 als Display Server:** xllvnc
- **Nutzung von Weston als Wayland-basierter Display Server:** RDP-Backend für Weston

Falls Sie also nur den Linux-Framebuffer oder eben einen X-Server nutzen, können Sie die Grafikausgabe mit dem Programm *x11vnc* auf Ihren Rechner umleiten. Dieses aktiveren Sie in Buildroot wie folgt:

- *Target packages → Graphic libraries and applications → X.org X Window System*
- *Target packages → Graphic libraries and applications → x11vnc*

Wie es dann weitergeht, hängt aber davon ab, ob Sie einen X-Server nutzen oder nicht. Falls kein X-Server zum Einsatz kommt, bearbeiten Sie die Datei `~/custom/board/rootfs_overlay/etc/inittab`, in der die zu startenden Systemdienste definiert sind. Dort tragen Sie gegen Ende der Datei, kurz bevor die Login-Konsole gestartet wird, fügen Sie folgende Zeile ein:

```
::once:/bin/x11vnc -forever -rawfb console
```

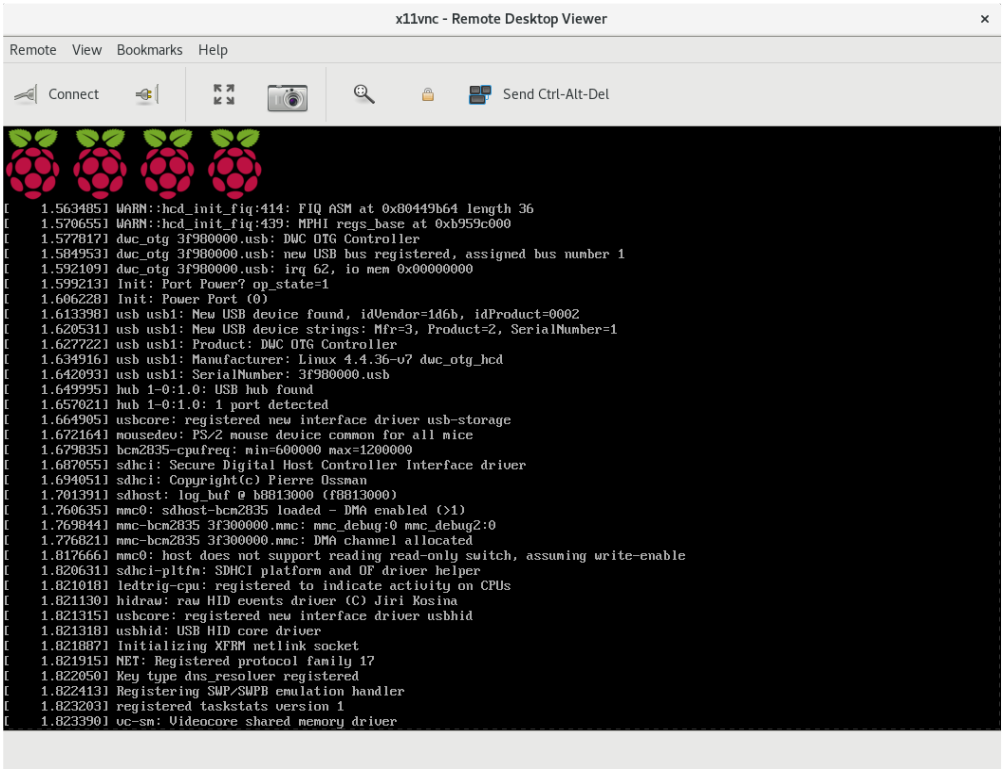

Abb. 26: Hurra es klappt! Zugriff auf den Raspberry Pi mit VNC

Nutzen Sie stattdessen wie in Kapitel 5.5 beschrieben einen X-Server, bearbeiten Sie stattdessen die Datei `~/custom/board/rootfs_overaly/opt/xinitrc` und fügen dort folgende Zeile ein:

```
x11vnc -forever &
```

In beiden Fällen benötigen Sie dann einen VNC Viewer, Um sich mit dem Raspberry Pi zu verbinden. Die meisten Linux-Distributionen sowie macOS haben bereits einen VNC Viewer installiert, der sich meistens meistens *Screen Sharing* bzw. *Remote Desktop* oder ähnlich nennt. Für Windows können Sie einen der folgenden Verwenden:

- TightVNC[65]
- TigerVNC[66]
- RealVNC[67]

Die URL für die Verbindung lautet jeweils `vnc://192.168.99.99`. Beachten Sie jedoch, dass die Datenübertragung über VNC teilweise recht langsam sein kann.

Haben Sie sich anstelle von X11 für die Wayland-Referenzimplementierung Weston entschieden, können Sie das RDP-Backend von Weston nutzen, um die Grafikausgabe umzuleiten. Wie der Name sagt, kommt dabei das von Microsoft erfundene *Remote Desktop Protocol* zum Einsatz, das auch von den Windows Terminal Services verwendet wird. In diesem Fall müssen Sie also eher für Linux und macOS nach einem Drittanbieterprogramm suchen, währen Windows bereits den nötigen Viewer mit an Board hat. Um das RDP-Backend in die Firmware zu integrieren, aktivieren Sie folgenden Menüeintrag in Buildroot:

Target packages → Graphic libraries and applications → weston → RDP compositor

65 http://www.tightvnc.com/download.php
66 http://tigervnc.org/ bzw. https://github.com/TigerVNC/tigervnc/releases
67 https://www.realvnc.com/download/viewer/

Anschließend stellen Sie sicher, dass das Backend in der Datei `/opt/wayland/weston.ini` (zu finden im Overlayverzeichnis) das RDP-Backend anstelle des DRM-Backends verwendet wird. Allerdings werden Sie dann feststellen, dass der Desktop nicht mehr auf einen am Raspberry Pi angeschlossenen Bildschirm ausgegeben wird. Sie müssen sich also stets entscheiden, ob Sie den Bildschirm oder RDP nutzen wollen.

```
01  [core]
02  backend=rdp-backend.so
03  …
```

6 Java-Entwicklung für Raspberry Pi

6.1 Java in die Firmware integrieren

Java ist leider nicht in den Paketquellen von Buildroot enthalten und muss daher auf anderem Wege in die Firmware integriert werden. Am einfachsten ist es dabei, sich die offizielle Java Runtime Edition für Linux auf ARMv6-Prozessoren herunterzuladen und dieser über ein *Filesystem Overlay* in die Firmware zu integrieren. Wie Filesystem Overlays allgemein funktionieren ist dabei in Kapitel 3.9 beschrieben. Am einfachsten ist es jedoch, wenn Sie mit der Minimalkonfiguration für Javaprojekte starten und dann den Anweisungen des Skripts `~/custom/unpack_java_overlay.sh` folgen. Starten Sie daher zunächst mit folgenden Befehlen ein neues Buildroot-Projekt:

```
$ cd ~/make
```

```
$ make dhbw_java_defconfig
```

Anschließend besuchen Sie die Java Downloadseite von Oracle und laden sich das offizielle *Java SE Embedded* für *ARM v6/v7 Linux - VFP, HardFP ABI, Little Endian* herunter. Die gesuchte Datei wird dabei als JDK angeboten und hat ungefähr folgenden Namen: `jdk-8u131-linux-arm64-vfp-hflt.tar.gz`. Legen Sie die Datei im ~/custom-Verzeichnis ab und starten Sie danach das Hilfsskript. Es führt sie durch alle weiteren Schritte:

```
buildroot@debian:~$ cd custom
buildroot@debian:~/custom$ ./unpack_java_overlay.sh
================================================
Filesystem Overlay mit Java Runtime vorbereiten
================================================

Dieses Skript hilft Ihnen, das Oracle JDK zu entpacken, um die Java Runtime
und JavaFX in die Firmware aufnehmen zu können.

  » Suche Datei ejdk-*-linux-arm*.tar.gz: OK
  » Lösche Overlay-Verzeichnis board/rootfs_overlay_java: OK
  » Entpacke Archiv ejdk-*-linux-arm*.tar.gz: …
...
```

Im Idealfall bestätigt das Skript, dass alle Schritte erfolgreich durchlaufen wurden und Sie können die Firmware nun erstellen. Falls das Entpacken stattdessen aber mit einem `Fehler 2` abbricht, unterstützt das Dateisystem keine symbolischen Links. In diesem Fall fragen Sie nach einem ZIP-Archiv, dessen Inhalt Sie einfach nach *~/custom/board* entpacken können. Die Befehle hierfür wären folgende:

```
$ cd ~/custom/board
```

```
$ unzip rootfs_overlay_java.zip
```

Anschließend versuchen Sie die Firmware zu bauen, und prüfen Sie, ob Sie auf dem Raspberry Pi folgenden Befehl ausführen können:

```
$ java -version
```

6.2 (Cross Development und Remote Debugging mit Netbeans)

6.3 (Deployment einer einfachen Konsolenanwendung)

6.4 (Deployment einer grafischen Anwendung)

6.5 (Direkte Nutzung der GPIO-Pins ohne zusätzliche Bibliotheken)

6.6 (Nutzung der GPIO-Pins mit Pi4J)

6.7 (Schlanke Java-Webanwendungen ohne Java EE)

7 (Python-Entwicklung für Raspberry Pi)

7.1 (Python in die Firmware integrieren)

7.2 (Deployment einer einfachen Konsolenanwendung)

7.3 (Grafik und Sound mit pygame (SDL))

7.4 (Entwicklung eines einfachen Socket-Servers mit Python)

7.5 (Entwicklung eines Webservers mit Python)

7.6 (Empfang und Versand von MQTT-Nachrichten mit Python)

8 Node.js-Entwicklung für Raspberry Pi

8.1 Grundlagen der Node.js-Entwicklung

Node.js[68] ist eine Plattform zur Anwendungsentwicklung mit JavaScript. Technologisch basiert es auf der JavaScript-Engine von Chrome, verpackt diese aber so, dass sie auch außerhalb des Browsers nutzbar ist. Ursprünglich für die Entwicklung von Serveranwendungen gedacht, können inzwischen auch eingebettete und Desktop-Anwendungen mit Node.js entwickelt werden. Möglich wird dies dank `npm`, dem Package Manager für Node.js. Er ermöglicht es, eine Vielzahl von Paketen für die unterschiedlichsten Aufgabenstellungen zu installieren, auf die man bei der Entwicklung seiner eigenen Anwendung zurückgreifen kann. Unter anderem gibt es Pakete zur Entwicklung von Webanwendungen, für den Zugriff auf die GPIO-Pins des Raspberry Pi oder gar Pakete, mit denen grafische Bildschirmausgaben realisiert werden können. Folgende Pakete könnten zum Beispiel ganz interessant sein:

- **express.js[69]:** Weit verbreitetes Web-Framework für Node.js
- **Loopback[70]:** Datenbankgestütztes REST-Framework basierend auf express.js
- **node-rpio[71]:** Nutzung der GPIO-Pins des Raspberry Pi
- **onoff[72]:** Weiteres Paket zur Nutzung der GPIO-Pins
- **node-pitft[73]:** Grafische Ausgaben für Adafruit TFT-Displays
- **node-openvg-canvas[74]:** Grafische Ausgaben auf dem HDMI-Bildschirm mit einer zur HTML5 Canvas kompatiblen API

Weitere Pakete lassen sich auf https://www.npmjs.com finden. Wie wir später noch sehen werden, lassen sie sich ganz einfach mit Buildroot in die eigene Firmware aufnehmen. Node.js kann daher eine gute Alternative zu Java sein, das einerseits mehr Ressourcen verbraucht und andererseits nur mit manuellem Aufwand in Buildroot integriert werden kann. Ein weiterer Vorteil von Node.js ist, dass Sie Ihre Anwendung während der Entwicklung auch lokal auf Ihrem Entwicklungsrechner laufen lassen können, solange Sie keine spezifischen Funktionen des Raspberry Pi nutzen. Gerade am Anfang des Projekts können Sie sich also ganz auf die Programmierung konzentrieren, bevor Sie sich dann Buildroot zuwenden.

Um mit der Entwicklung starten zu können, benötigen Sie zunächst Node.js auf Ihrem Entwicklungsrechner. Zusätzlich empfiehlt es sich, den alternativen Package Manager Yarn zu installieren:

- https://nodejs.org
- https://yarnpkg.com

Anschließend legen Sie ein neues Verzeichnis auf Ihrem Entwicklungsrechner an und führen darin folgenden Befehl aus, um ein neues Node.js-Projekt zu starten[75]:

```
$ yarn init
```

68 https://nodejs.org
69 http://expressjs.com
70 http://loopback.io
71 https://github.com/jperkin/node-rpio
72 https://github.com/fivdi/onoff
73 https://github.com/vesteraas/node-pitft
74 https://github.com/luismreis/node-openvg-canvas
75 Wenn Sie Yarn nicht nutzen wollen, können Sie an den meisten Stellen statt `yarn` auch `npm` aufrufen. Yarn hat aber den Vorteil, dass es die exakte Version der abhängigen Node.js-Module speichert und sich somit das Projekt später exakt auf den Raspberry Pi übertragen lässt.

Daraufhin fragt Sie das Skript nach dem Namen, der Versionsnummer und so weiter des Projekts. Hierbei gelten folgende Regeln:

- **Projektname:** Darf nur die Zeichen A-Z, a-z sowie – und _ enthalten. Es dürfen auch keine Leerzeichen enthalten sein,
- **Versionsnummer:** Muss dem SemVer-Schema[76] entsprechen. Sprich sie muss drei durch einen Punkt getrennte Nummern enthalten, z.B. 1.2.0 oder 4.2.21. Die drei Nummern werden Major, Minor und Patch genannt, also Hauptversion, Unterversion und Patch. Finden ausschließlich Fehlerkorrekturen statt, wird die Patchnummer hochgezählt. Bei kompatiblen Änderungen wird die Unterversion hochgezählt und bei inkompatiblen Änderungen die Hauptversion.
- Die restlichen Werte können Sie leer lassen.

Der Mühe Lohn ist dann die Datei package.json mit ungefähr folgendem Inhalt:

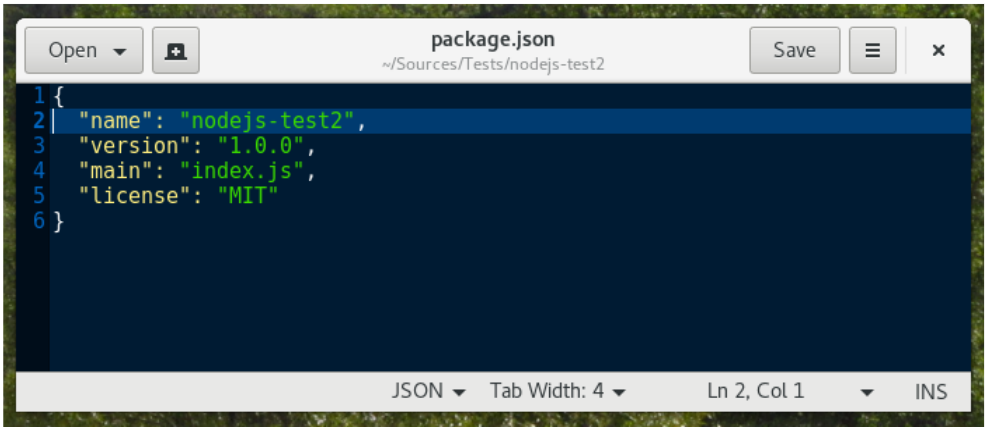

Abb. 27: Inhalt der Datei package.json eines neuen Node.js-Projekts

Auf den ersten Blick sieht das nach nicht sehr viel aus. Allerdings hat diese Datei eine ganz zentrale Bedeutung, da durch Sie das aktuelle Verzeichnis zu einem Node.js-Modul wird, dessen Quellcode mit Node.js ausgeführt werden und später auch auf npmjs.com veröffentlicht werden kann. Eine gute Idee ist es dabei, das Verzeichnis mit der Versionsverwaltung git zu überwachen und ggf. auf Github hochzuladen, da beides Voraussetzung für die Veröffentlichung auf nmpjs.com ist. Außerdem können Sie Buildroot dann später einfach anweisen, sich den Quellcode der Anwendung aus dem git-Repository zu besorgen.

Wenn Sie sich den Inhalt der `package.json` näher ansehen, fällt Ihnen dabei sicher auch der Verweis auf eine Datei namens `index.js` auf. Diese Datei, die wir allerdings erst nach anlegen müssen, dient eigentlich dazu, die Variablen, Funktionen und Objekte zu definieren, die fremde Module importieren können, wenn Sie unser Node.js-Modul importieren. Da wir hier allerdings kein wiederverwendbares Modul im Sinne einer Funktionsbibliothek schreiben, wollen wir die Datei stattdessen als Startdatei für unser Projekt auswählen. Es soll also die Datei sein, deren Quellcode als erstes ausgeführt wird, wenn unsere neue Anwendung gestartet wird. Legen Sie die Datei daher an und übernehmen Sie zum Ausprobieren zunächst folgenden Quellcode (natürlich ohne Zeilennummern). Das Beispiel lädt den deutschen Wikipedia-Artikel zu Node.js im XML-Format herunter und gibt ihn auf der Konsole aus:

```
01  #! /usr/bin/env node
02  let https = require("https");
03
04  https.get({
05      hostname: "de.wikipedia.org",
06      port: 443,
```

76 http://semver.org

```
07      path: "/wiki/Spezial:Exportieren/Node.js",
08      agent: false
09  }, (response) => {
10      response.on("data", (data) => {
11          console.log(data.toString());
12      });
13  });
```

Lassen Sie sich dabei nicht von `(response) => { … }` bzw `(data) => { … }` irritieren. Es handelt sich dabei um sog. Pfeilfunktionen[77], die mit ES6 eingeführt wurden und sich in mancherlei Dingen logischer als die alten `functions` verhalten. Generell haben die Zeilen folgende Bedeutung:

- **01:** Shebang, damit die Datei unter Linux direkt ausgeführt werden kann
- **02:** Import des `https`-Moduls, mit dem wir eine HTTPS-Anfrage absetzen können
- **04:** Wir wollen eine GET-Anfrage an eine HTTPS-gesicherte Seite schicken
- **05 – 08:** Anfragedaten wie Hostname, Port und Pfad der aufzurufenden URL
- **09 – 13:** Rückruffunktion, die bei Eintreffen der Antwort aufgerufen wird
- **10 – 12:** Event Handler für das `data`-Event des Antwortobjekts, der immer dann aufgerufen wird, wenn der Server Daten als Teil der Antwort geschickt hat.[78]
- **11:** Umwandlung der empfangenen Daten in einen String und Ausgabe auf der Konsole

Um die Datei auszuführen, geben Sie folgenden Befehl ein:

```
$ node index.js
```

Unter macOS und Linux können Sie die Datei auch mit `chmod` ausführbar machen und dann dank dem Kommentar in der ersten Zeile direkt starten. Für die Integration in die spätere Linux-Firmware kann das ganz interessant sein:

```
$ chmod +x index.js
```

```
$ ./index.js
```

Damit wissen Sie im Prinzip schon alles, was Sie für die Entwicklungen einfacher Anwendungen mit Node.js benötigen. Allerdings lohnt es sich, sich mit dem Neuerungen von ES6 vertraut zu machen, um einige der Probleme des „alten“ JavaScripts zu umgehen. Die Seite es6-features.org[79] bietet hier einen guten Überblick. Ebenso sollten Sie mindestens auch noch das folgende Kapitel lesen, um zu lernen, wie Sie den Quellcode sauber gliedern und fremde Node.js-Module nutzen können.

8.2 Gliederung des Quellcodes und Import fremder Module

Das Beispiel aus dem vorherigen Kapitel ist natürlich noch sehr einfach gestrickt: Der gesamte Quellcode ist in der Datei `index.js` enthalten und es wurden keine fremden Module benutzt, die nicht bereits mit Node.js ausgeliefert werden. Praxisrelevant ist das allerdings nicht. Denn bei den meisten Projekten lohnt es sich, gleich von Anfang an auf eine saubere Gliederung des Quellcodes in mehrere Dateien und Verzeichnisse zu achten. Ebenso muss man ja auch nicht immer das Rad neu erfinden, wenn es im Internet schon für viele Aufgaben fertige Module gibt, die man nur verwenden muss.

77 https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Functions/Pfeilfunktionen

78 In unserem Fall wird das `data`-Event nur einmal ausgelöst. Falls der Server das sog. Chunked Encoding unterstützt, kann er beispielsweise eine große Datei in kleinere Chunks zerlegen und diese einzeln verschicken. In diesem Fall wird das Event für jeden empfangenen Chunk gefeuert.

79 http://es6-features.org

Die Lösung für das erste Problem sind die sog. Module, die mit ES6 in JavaScript eingeführt wurden.[80] Durch sie wird es möglich, den Quellcode in mehre Dateien in einer beliebigen Verzeichnisstruktur zu verteilen und dabei für jede Datei zu bestimmen, welche Inhalte für die Verwendung in den anderen Dateien freigegeben werden. Die Lösung für das zweite Problem heißt `npm` bzw. `yarn` und hängt unmittelbar mit dem ersten Problem zusammen, wie Sie gleich sehen werden.

Im vorherigen Kapitel haben wir gesagt, dass unser gesamtes Projekt ein sog. Node.js-Modul darstellt, da darin eine Datei namens `package.json` enthalten ist. Zusätzlich handelt es sich bei jeder JavaScript-Datei aber auch um ein Modul, das bis auf den Namen zunächst aber erst einmal Nichts mit den Node.js-Modulen gemeinsam hat. Gemeint ist stattdessen tatsächlich einfach eine Quellcode-Datei, die Variablen, Funktionen oder Klassen enthält, die man in seiner Anwendung so braucht und auf die man daher auch aus den anderen Dateien[81] heraus zugreifen möchte. Hierfür müssen die Inhalte einer Datei mit einer einfachen Export-Syntax explizit freigegeben werden, damit sie innerhalb einer anderen Datei importiert werden können.

Ein Beispiel soll das Prinzip verdeutlichen. Legen Sie hierfür eine Datei namens `utils.js` mit folgendem Inhalt an:

```
01  let PI = 3.14159265359;
02
03  let quadrat = (n) => {
04      return n * n;
05  }
06
07  class Rechner {
08      static summe(a, b) {
09          return a + b;
10      }
11
12      static kreisFlaeche(r) {
13          return PI * quadrat(r);
14      }
15  }
16
17  export { PI, Rechner };
```

Die entscheidende Zeile ist dabei die letzte Zeile mit der `export`-Anweisung. Dort werden die Variable `PI` sowie die Klasse `Rechner` für die Verwendung in den anderen Dateien freigegeben. Die Funktion `quadrat` fehlt hier, weshalb sie nur innerhalb derselben Datei genutzt werden kann. Innerhalb der Datei `index.js` könnten `PI` und `Rechner` nun wie folgt genutzt werden:

```
01  import { PI, Rechner } from "./utils.js";
02
03  console.log(`Die Kreiszahl PI: ${PI}`);
04
05  let flaeche = Rechner.kreisFlaeche(7);
06  console.log(`Fläche eines Kreises mit Radius 7cm: ${flaeche}cm`);
```

Über die `import`-Anweisung werden also einfach alle Objekte importiert, die man aus einer anderen Daten verwenden möchte. Dabei muss es sich natürlich nicht, wie oben gezeigt, um alle Objekte handeln, die von der anderen Datei exportiert werden. Es genügt, nur die Objekte anzugeben, die man auch tatsächlich nutzen will. Der Name der exportierenden Datei wird dabei als String angegeben, wobei `./` am Anfang bedeutet, dass die Datei im selben Verzeichnis liegt. Liegt die Datei stattdessen in einem anderen Verzeichnis, kommt es darauf an, wo sich das andere relativ zum eigenen Verzeichnis befindet. Hier ein paar Beispiele:[82]

- **`./utils/rechner.js`:** Die Datei `rechner.js`, die im Unterverzeichnis `utils` liegt
- **`./utils/math/rechner.js`:** Die Datei `rechner.js`, die im Unterverzeichnis `math` innerhalb des Unterverzeichnisses `utils` liegt.

80 https://www.sitepoint.com/understanding-es6-modules/ bietet eine gute Erklärung.
81 Korrekt: Modulen
82 Vgl. hierzu auch: http://www.tommyherrmanndesign.com/nof/Relative-Pfade/

- **`../utils.js`:** Die Datei `utils.js`, die im übergeordneten Verzeichnis liegt
- **`../utils/rechner.js`:** Die Datei `rechner.js`, die um Unterverzeichnis `utils` des übergeordneten Verzeichnisses liegt.

Stand Juni 2017 funktioniert der hier gezeigte Export und Import allerdings noch nicht mit Node.js, da die dazugehörige *Module Loader Specification* noch nicht in der von Node.js verwendeten JavaScript-Engine umgesetzt wurde. Das obige Beispiel ist daher leider noch Zukunftsmusik. Node.js hat hierfür jedoch folgenden Workaround parat. Hier wieder die Datei `utils.js`, die sich nur in den letzten beiden Zeilen von der vorherigen Version unterscheidet:

```
01  let PI = 3.14159265359;
02
03  let quadrat = (n) => {
04      return n * n;
05  }
06
07  class Rechner {
08      static summe(a, b) {
09          return a + b;
10      }
11
12      static kreisFlaeche(r) {
13          return PI * quadrat(r);
14      }
15  }
16
17  module.exports.PI = PI;
18  module.exports.Rechner = Rechner;
```

Innerhalb der `index.js` werden die beiden Objekte dann nicht mit `import` sondern mit der `require()`-Funktion importiert. Die Regeln für den Dateipfad sind dabei dieselben wie zuvor:

```
01  let utils = require("./utils.js");
02
03  console.log(`Die Kreiszahl PI: ${utils.PI}`);
04
05  let flaeche = utils.Rechner.kreisFlaeche(7);
06  console.log(`Fläche eines Kreises mit Radius 7cm: ${flaeche}cm`);
```

Im Gegensatz zum gezeigten Beispiel ist es allerdings üblich, nicht mehr als ein Objekt zu exportieren, da dadurch der Quellcode des Verwenders übersichtlicher wird. Dies würde dann so aussehen:

```
01  # rechner.js
02  class Rechner {
03      ...
04  }
05
06  module.exports = Rechner;
```

```
01  # index.js
02  let Rechner = require("./rechner.js");
03
04  let flaeche = Rechner.kreisFlaeche(7);
05  ...
```

Das schöne an den Imports ist, dass man auch Objekte aus anderen Node.js-Modulen importieren und sich somit der riesigen Auswahl auf npmjs.com bedienen kann. Alles was man hierfür machen muss, ist die Module mit `npm` bzw. `yarn` zu installieren und sie dann ohne das führende `./` zu importieren. Oder anders ausgedrückt: Wenn Sie bei einem Import das `./` weglassen, ist damit nicht eine Quellcodedatei innerhalb des eigenen Projekts sondern ein fremdes Node.js-Modul gemeint, das zuvor mit `npm` bzw. `yarn` heruntergeladen und installiert wurde.

Schauen wir uns das Prinzip anhand des `request`-Moduls an, um damit das einleitende Beispiel aus Kapitel 8.1 zu vereinfachen. Zunächst müssen wir das Modul installieren, wofür Sie folgenden Befehl innerhalb des Projektverzeichnisses ausführen müssen:

```
$ yarn add request
```[83]

Wenn Sie sich danach das Projektverzeichnis ansehen, werden Sie feststellen, dass nun einige Dateien und Verzeichnisse hinzugekommen sind:

```
├─node_modules/
│  ├─ajv/
│  ├─asn1/
│  ├─…
│  ├─request/
│  └─…
├─index.js
├─package.json
└─yarn.lock
```

Insbesondere das neue Verzeichnis `node_modules` beinhaltet nun richtig viele Unterverzeichnisse. Das liegt daran, das hier die in unserem Projekt verwendeten Node.js-Module gespeichert werden. Dabei haben wir zwar nur das `request`-Modul hinzugefügt, dieses wiederum besitzt jedoch Abhängigkeiten zu anderen Modulen, die selbst von anderen Modulen abhängen. Diese befinden sich daher nun alle im `node_modules`-Verzeichnis. Die exakte Version der Module ist dabei in der Datei `yarn.lock` festgehalten, so dass wenn Sie das Projekt auf einen anderen Rechner umziehen, die exakt gleiche Konstellation dort wiederhergestellt werden kann. Denn üblicherweise teilt man das `node_modules`-Verzeichnis nicht mit anderen Rechnern und checkt es auch nicht in der Versionsverwaltung ein.

Auch die Datei `package.json` sieht nun anders aus. Neu ist das Dictionary `dependencies`, in dem das `request`-Modul nun als Abhängigkeit unserer Anwendung eingetragen ist:

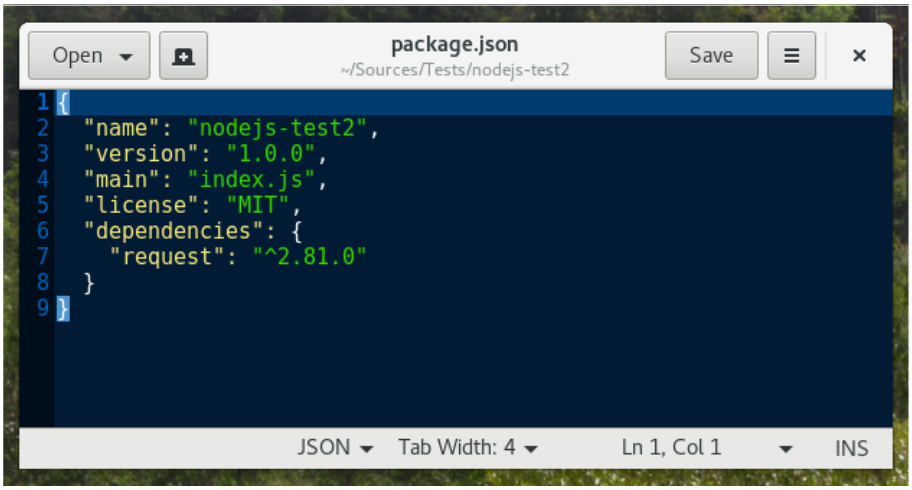

Abb. 28: Inhalt der Datei package.json, nachdem das request-Modul hinzugefügt wurde

Bleibt uns nur noch, das neue Modul zu nutzen. Ändern Sie daher den Inhalt der `index.js` wie folgt ab. Wie Sie sehen, kann das Modul nun einfach mit der `require()`-Funktion importiert werden, jedoch ohne vorangestelltes `./`, da es sich ja um ein fremdes Modul handelt.

```
01  let request = require("request");
02
03  request("https://de.wikipedia.org/wiki/Spezial:Exportieren/Node.js", (err, res, body) => {
04      console.log(body);
05  });
```

83 Wie gesagt wäre auch `npm add request` möglich, wenn Sie Yarn nicht verwenden wollen.

Das war's. Nun können Sie mit Node.js loslegen. Die nachfolgenden Kapitel geben Ihnen dabei ein paar Beispiele für häufig benötigte Aufgabenstellungen.

8.3 Umzug eines Node.js-Projekts auf einen anderen Rechner

Wenn Sie Ihr Node.js-Projekt in ein anderes Verzeichnis verschieben oder auf einem anderen Rechner zum Laufen bringen wollen, geht das aufgrund der aufgezeichneten Abhängigkeiten in der `package.json` und auch der `yarn.lock`-Datei sehr einfach. Kopieren Sie einfach den Quellcode ohne das `node_modules`-Verzeichnis an die neue Position und installieren Sie dort die Abhängigkeiten erneut:

```
$ yarn install
```

Dies ist notwendig, da das `node_modules`-Verzeichnis nicht dafür gedacht ist, an eine andere Stelle kopiert zu werden. Yarn stellt bei der Neuinstallation aber sicher, dass exakt die gleichen Module in exakt der gleichen Version wie im ursprünglichen Setup installiert werden.

8.4 Wiederkehrende Aufgaben mit npm-Skripten automatisieren

Während der Entwicklung werden Sie vielleicht feststellen, dass Sie häufig dieselben Befehle auf der Konsole eingeben müssen, um eine bestimmte Aktion auszuführen. Beispielsweise kann es sein, dass Sie öfters einen Fehler beheben und die Korrektur dann in git einchecken und auf Github veröffentlichen wollen. Hierfür müssen Sie jedes mal folgende zwei Befehle eingeben:

```
$ git commit -m "Fehler korrigiert"
```

```
$ git push
```

Das ist zwar nicht viel, aber es wäre trotzdem schön, wenn sich das vereinfachen ließe. Dank npm-Skripten kann man das.[84] Hierfür muss in der Datei `package.json` nur ein Skript für diese beiden Befehle definiert werden, das zum Beispiel `push_fix` heißen könnte. Dieses könnten Sie dann wie folgt aufrufen:

```
$ npm run push_fix
```

Der entsprechende Eintrag in der `package.json` sieht dann wie folgt aus:

```
01  {
02      "name": "nodejs-test2",
03      "version": "1.0.0",
04      "main": "index.js",
05      "license": "MIT",
06      "scripts": {
07          "push_fix": "git commit -m 'Fehler korrigiert'; git push"
08      }
09  }
```

Zugegeben, das Beispiel ist etwas konstruiert, aber Sie verstehen, worum es geht. Die Befehle, die Sie in der `package.json` hinterlegen, sind einfache Shell-Befehle. Deshalb sollten an dieser Stelle nicht allzu komplexe Konstrukte hinterlegt werden, die sich auf den verschiedenen Betriebssystemen unterschiedlich verhalten. Denn Insbesondere die Windows-Kommandozeile unterscheidet sich in einigen Punkte deutlich von Linux oder macOS. Allerdings ist das gar nicht so dramatisch, wie es zunächst klingt. Denn mit JavaScript steht Ihnen ja eine wundervolle, plattformunabhängige Programmiersprache zur Verfügung. Das heißt, anstatt eine komplexe Aufgabe mit bandwurmlangen Shell-Befehlen umzusetzen, können Sie einfach eine JavaScript-Datei aufrufen, die dieselbe Aufgabe viel eleganter erledigt. Zum Beispiel so:

84 Eine über dieses Kapitel hinausgehende Erklärung mit praxisnäheren Beispielen findet sich in folgendem Blog-Eintrag: https://www.keithcirkel.co.uk/how-to-use-npm-as-a-build-tool/ Der Autor argumentiert, dass die guten alten npm-Skripte in vielen Fällen wesentlich problemloser funktionieren als die vielen neuen Automatisierungswerkzeuge, die seither für JavaScript veröffentlicht wurden. Wie Recht er doch hat …

```
01  {
02      ...
03      "scripts": {
04          "clean": "node script/clean.js"
05      }
06  }
07
```

Das Skript können Sie dann einfach mit `npm run clean`[85] aufrufen. Innerhalb von `script/clean.js` könnten Sie dann zum Beispiel das `shelljs`-Modul nutzen, um damit plattformunabhängig das Quellcodeverzeichnis von allen überflüssigen Zwischendateien zu bereinigen. Hierfür würden Sie das `shelljs`-Modul dann auch nur als Entwicklungsabhängigkeit installieren, da es für die spätere Nutzung der Anwendung nicht benötigt wird. Der entsprechende Befehl dazu lautet:

```
$ yarn add -D shelljs
```

Durch den Schalter `-D` bzw. `--dev` wird das Modul in der `package.json` in der Rubrik `devDependencies` eingetragen und dadurch später auch nicht auf dem Raspberry Pi installiert.

8.5 Node.js in die Firmware integrieren

Wenn Sie nun eine kleine Node.js-Anwendung haben, ist die Integration in einer mit Buildroot erstellte Firmware ziemlich einfach. Sie müssen lediglich unter *Target packages → Interpreter languages and scripting* das Paket *nodejs* aktivieren und dann bei *Additional modules* den richtigen Pfad zu Ihrem Quellcode eintragen. Am einfachsten ist es dabei, wenn die Anwendung entweder aus einem git-Repository gezogen werden kann oder (falls Sie kein git nutzen) im altbekannten `custom`-Verzeichnis von Buildroot liegt.

Im ersten Fall tragen Sie einfach die URL des git-Repositories ein, wie zum Beispiel:

```
https://github.com/Beispiel/beispiel.git
```

Im zweiten Fall geben Sie den Pfad innerhalb des custom-Verzeichnisses wie folgt ein:

```
$(BR2_EXTERNAL_DHBW_PATH)/beispiel
```

Innerhalb der Firmware befindet sich das Modul dann in einem Unterverzeichnis von `/usr/lib/node_modules`, in unserem Fall also in `/usr/lib/node_modules/beispiel`. Dort befindet sich dann also auch die `index.js`-Datei, die zur Ausführung der Anwendung gestartet werden muss. Diese könnten Sie dann mit folgendem Befehl auf dem Raspberry Pi starten:

```
$ node /usr/lib/node_modules/beispiel/index.js
```

Oder Sie machen die Datei wie in Kapitel 3.11 beschrieben ausführbar, so dass sie auch ohne die Angabe des `node`-Interpreters gestartet werden kann:

```
$ /usr/lib/node_modules/beispiel/index.js
```

Falls Ihnen das immer noch zu lang ist, könnten Sie mit derselben Technik im `/bin`-Verzeichnis einen symbolischen Link erzeugen, zum Beispiel mit dem Namen `beispiel`. Dann könnten Sie die Anwendung aus jedem Verzeichnis heraus starten, ohne den ganzen Pfad angeben zu müssen:

```
$ beispiel
```

Wichtiger Hinweis: Wenn Sie unter *Target packages → Interpreter languages and scripting → Additional modules* ein neues Modul eintragen, müssen Sie Node.js neu bauen, damit Buildroot das neue Modul in die Firmware integriert. Dies können Sie durch den folgenden Befehl sicherstellen.

```
$ make nodejs-dirclean
```

85 Bzw. in diesem Fall sogar mit `npm clean`, da dies ein Alias für `npm run clean` ist. Für einige häufig benötigte Skripts gibt es solche Aliase, so dass das `run` in der Mitte wegfallen kann.

8.6 Quellcodeänderungen automatisch auf den Raspberry Pi übertragen

Solange Ihre Node.js-Anwendung keine Besonderheiten des Raspberry Pi nutzt, können Sie die Anwendung vollständig auf Ihrem Entwicklungsrechner programmieren und testen. Vermutlich werden Sie aber recht bald an einen Punkt kommen, ab dem Sie die Anwendung eigentlich nur noch auf dem Raspberry Pi sinnvoll testen können, da Sie sonst zu viele Eigenheiten der Firmware auf Ihrem Entwicklungsrechner nachstellen müssten. Damit Sie dann nicht nach jeder kleinen Quellcodeänderung die Firmware neu bauen und auf SD-Karte schreiben müssen, wird an dieser Stelle eine Technik erläutert, wie die Quellcode nach dem Speichern automatisch auf den Raspberry Pi übertragen und die Anwendung dort ohne das Betriebssystem herunterfahren neugestartet werden kann. Diese Technik wird in verschiedenen Quellen *Hot Reload* bzw. *Live Reload* genannt und ist mit ein wenig JavaScript-Quellcode relativ einfach umzusetzen.

☞ Die hier vorgestellte Technik ist nur für die Entwicklung gedacht ☜

Bitte nehmen Sie obenstehenden Hinweis wirklich ernst. Die beschriebene Technik stellt nur sicher, dass die Anwendung während der Entwicklung kontinuierlich auf den Raspberry Pi übertragen und dort neugestartet wird. Für das fertige Produkt müssen Sie die Anwendung wie im Kapitel 8.5 beschrieben richtig in die Firmware integrieren und sehr wahrscheinlich auch wie in Kapitel 3.12 beschrieben beim Hochfahren des Systems automatisch starten.

Auf den Entwicklungsrechner benötigen Sie zunächst die beiden Node.js-Module `wait-run` und `node-ssh`. `wait-run` dient dabei der automatischen Überwachung des Quellcodeverzeichnisses auf Veränderungen. Mit `node-ssh` kann dann der Quellcode auf den Raspberry Pi kopiert und die Anwendung dort neugestartet werden. Die Module müssen daher als Entwicklungsabhängigkeit hinzugefügt werden:

```
$ yarn add --dev wait-run
```

```
$ yarn add --dev node-ssh
```

Anschließend ändern Sie die Datei `package.json` so ab, dass darin zwei Skripte mit den Namen `reload` und `watch` definiert wird. `reload` dient der einmaligen manuellen Übertragung des Quellcodes auf den Raspberry Pi, `watch` der automatischen Übertragung, sobald eine Datei geändert wurde. Für einen einmaligen Übertrag würden Sie daher später folgenden Befehl ausführen:

```
$ npm run reload
```

Und zur permanenten Überwachung während einer Entwicklungssession würden Sie stattdessen folgenden Befehl ausführen:

```
$ npm run watch
```

In der `package.json` ergänzen Sie daher zunächst die nachfolgend gelb markierten Inhalte:

```
01  {
02      "name": "nodejs-test2",
03      "version": "1.0.0",
04      "main": "index.js",
05      "license": "MIT",
06      "scripts": {
07          "reload": "node script/reload.js",
08          "watch": "wait-run --pattern '*.*' -- npm run reload"
09      },
10      "config": {
11          "ssh_host": "192.168.99.99",
12          "ssh_port": 22,
13          "ssh_user": "mulder",
14          "ssh_pass": "xfiles",
15          "ssh_dir": "/home/mulder/nodejs_app",
16          "ssh_install": "yarn install",
17          "ssh_start": "index.js"
18      }
19  }
```

Die Werte innerhalb des config-Objekts müssen Sie natürlich entsprechend anpassen, wenn Sie nicht mit der statischen IP-Adresse aus Kapitel 3.5 arbeiten oder einen anderen Systembenutzer verwenden. Der Teil zur automatischen Überwachung des Quellcodes ist damit schon fertiggestellt. Fehlt nur noch die Datei `script/reload.sh`, in der die Anwendung auf den Raspberry Pi übertragen und dort neugestartet wird. Diese hat folgenden Inhalt:

```
01  /**
02   * Hilfsskript, das den Quellcode der Anwendung auf den Raspberry Pi
03   * kopiert und dort die Anwendung neustartet. Zusammen mit wait-run
04   * wird somit ein Hot Reloading während der Entwicklung ermöglicht.
05   */
06  const path = require("path");
07  const node_ssh = require("node-ssh");
08  const simple_ssh = require("simple-ssh");
09
10  // Zugangsdaten für die SSH-Verbindung
11  const config = {
12      host: process.env.npm_package_config_ssh_host,
13      port: process.env.npm_package_config_ssh_port,
14      username: process.env.npm_package_config_ssh_user,
15      password: process.env.npm_package_config_ssh_pass,
16  }
17
18  let source_dir = path.normalize(path.join(__dirname, ".."));
19  let remote_dir = process.env.npm_package_config_ssh_dir;
20  let main_script = process.env.npm_package_config_ssh_start;
21  let install_cmd = process.env.npm_package_config_ssh_install;
22
23  // SSH-Verbindung zum Raspberry Pi herstellen
24  let ssh = new node_ssh();
25
26  ssh.connect(config).then(() => {
27      // Node-Prozesse beenden
28      console.log("» Beende alle node-Prozesse auf der Zielmaschine");
29      return ssh.execCommand("killall node");
30  }).then(() => {
31      // Quellcodeverzeichnis auf der Zielmaschine löschen
32      console.log(`» Lösche ${remote_dir} auf der Zielmaschine`);
33      return ssh.execCommand(`rm -fr ${remote_dir}`);
34  }).then(() => {
35      // Quellcode der Anwendung hochladen
36      console.log("» Lade neue Version des Quellcodes hoch");
37      return ssh.putDirectory(source_dir, remote_dir, {
38          recursive: true,
39          validate: (localPath) => !localPath.includes("node_modules"),
40      });
41  }).then(() => {
42      // Verbindung mit node-ssh trennen
43      ssh.dispose();
44  }).then(() => {
45      // Verbindung mit simple-ssh herstellen und die Anwendung neustarten,
46      // weil so die Umleitung der Konsole hierher funktioniert
47      console.log("» Starte Anwendung neu");
48
49      let ssh2 = new simple_ssh({
50          host: config.host,
51          port: config.port,
52          user: config.username,
53          pass: config.password,
54      });
55
56      ssh2.exec(
57          `cd ${remote_dir}; ${install_cmd}; node ${main_script}`,
58          {
59              pty: true,
60              out: (data) => console.log(data.replace(/\s+$/g, "")),
61          }
62      ).start();
63  }).catch((error) => {
64      // Irgendetwas ist schief gelaufen
65      console.log(error);
66  });
```

Bei den vielen `then(() => {…})`-Aufrufen handelt es sich übrigens um Promises[86], eine der neuen Techniken um asynchronen Code noch einigermaßen übersichtlich zu halten.

Auf dem Raspberry Pi muss, damit die Anwendung zum Laufen gebracht werden, `npm` und `yarn` vorhanden sein. Für das fertige Produkt werden `npm` und `yarn` nicht benötigt, jedoch werden Sie hier genutzt, um die anhängigen Pakete der Node.js-Anwendung auf dem Raspberry Pi zu aktualisieren. Zusätzlich wird noch ausreichend Platz auf der SD-Karte benötigt, um die heruntergeladenen Module zu speichern. Stellen Sie daher sicher, dass folgende Einstellungen in Buildroot vorgenommen werden. Wie viel zusätzlichen Platz Sie dabei auf der SD-Karte reservieren, müssen Sie im Zweifelsfall durch Ausprobieren herausfinden (s. Kapitel 3.15).

- *Target packages*
  → *Interpreter languages and scripting*
  → *NPM for the target*
  → *Additional modules:* `yarn`

- *Filesystem images*
  → *Extra size in blocks:* `102400`

Das war‘s auch schon. Starten Sie nun `npm run watch`, ändern Sie eine Datei und beobachten Sie, wie die Anwendung auf dem Raspberry Pi automatisch neugestartet wird.

8.7 Entwicklung eines einfachen Socket-Servers mit Node.js

Der nachfolgende Quellcode ist an das Beispiel im deutschen Wikipedia-Artikel zu Node.js angelehnt und zeigt, wie ein einfacher Socketserver mit Node.js realisiert werden kann. Aufgrund der asynchronen Verarbeitung von JavaScript werden hierfür im Gegensatz zu Java keine Threads oder komplizierte Klassen wie die des Pakets `java.nio` benötigt.

```
01  const net = require("net");
02
03  const listen_ip = "localhost";
04  const listen_port = 7000;
05
06  let server = net.createServer((socket) => {
07      // Begrüßungsnachricht an den Client senden
08      socket.setEncoding("utf-8");
09      socket.setNoDelay();
10
11      socket.write("READY!\n");
12
13      // Nachrichten des Clients empfangen und verarbeiten
14      socket.on("data", (data) => {
15          data = data.replace(/\s+$/g, "");
16          let cmd = data.split(" ")[0];
17          let val = data.split(" ").slice(1).join(" ");
18
19          switch (cmd) {
20              case "IAM":
21                  socket.write(`HELLO ${val}\n`);
22                  break;
23              case "BYE":
24                  socket.end("BYE\n");
25                  break;
26              default:
27                  socket.write("UNKNOWN MESSAGE\n");
28                  break;
29          }
30      });
31  });
32
33  console.log(`Server empfängt auf ${listen_ip}:${listen_port}`)
34  server.listen(listen_port, listen_ip);
```

86 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Es handelt sich um ein simples, zeilenbasiertes Protokoll, wie es im Internet häufiger anzutreffen ist.[87] Direkt nach dem Verbindungsaufbau begrüßt der Server den Client mit der Nachricht `READY!` und wartet anschließend auf die Befehle des Clients. Diese könnten in diesem einfachen Beispiel `IAM …` und `BYE` sein, woraufhin der Server mit `HELLO …` bzw. ebenfalls mit `BYE` antwortet und die Verbindung trennt.

Ausprobieren können Sie den Server zum Beispiel mit `netcat`:

```
    buildroot@debian:~/custom/nodejs-socket-server$ nc localhost 7000
»»  READY!
««  IAM Mordack, the Preventor of Information Technology!
»»  HELLO Mordack, the Preventor of Information Technology!
««  BYE
»»  BYE
```

8.8 Entwicklung eines Webservers mit Node.js

In einem Punkt unterscheidet sich Node.js deutlich von anderen Programmiersprachen: Wenn man eine Webanwendung schreibt, schreibt man nicht einen Quellcode, der später durch einen Webserver ausgeführt wird, sondern man schreibt einen komplett eigenen Webserver. Im Vergleich zu Java oder Python mag das zunächst aufwändig klingen, dank der großen Auswahl auf npmjs.com ist es tatsächlich aber ganz einfach und bringt ein paar entscheidende Vorteile mit sich:

1. Die Webanwendung ist vollständig in sich geschlossen und muss nicht aufwändig mit einem Webserver integriert werden.
2. Dadurch dass das Deployment in einem vorhandenen Webserver wegfällt, kann es auch keine Inkompatibilitäten zwischen Webanwendung und Webserver geben. Denken Sie hier nur an alle die Kopfschmerzen, die Ihnen der Glassfish-Applikationsserver in Webprogrammierung oder Verteilte Systeme bereitet hat. Solch gruseligen Dinge gibt es für Node.js nicht.
3. Für das Deployment der Webanwendung reicht es, den Quellcode auf die Zielmaschine zu übertragen und dort das Startskript auszuführen.

Alles was Sie für die Entwicklung einer modernen Webanwendung benötigen ist das Express Webframework, das Sie mit folgendem Befehl installieren können:

```
$ yarn add express
```

Zusätzlich wird noch eine Template Engine benötigt, wenn Sie serverseitige Templates zur Erzeugung des HTML-Codes nutzen wollen. Im nachfolgenden Beispiel verwenden wird `nunjucks`[88] dafür.

```
$ yarn add nunjucks
```

```
$ yarn add express-nunjucks
```

Für das Projekt wird dabei folgende Verzeichnisstruktur vorausgesetzt:

```
├─node_modules/
│  └─…
├─static/
│  ├─style.css
│  ├─bg.jpg
│  └─…
├─templates/
│  ├─base.html
│  ├─index.html
│  └─…
```

87 Zwar handelt es sich HTTP/2 im Gegensatz zu seinen Vorgängern um ein rein binäres Protokoll, die am weitesten verbreiteten Protokolle sind jedoch alle textorientiert, da man sie auch ohne Spezialprogramme debuggen kann. Bei `READY!` weiß man eben sofort, was gemeint ist, bei der Bytefolge `0x42,0x03,0x06,0x86` hingegen nicht.

88 https://mozilla.github.io/nunjucks

```
├─index.js
├─package.json
└─yarn.lock
```

Das folgende Beispiel zeigt, wie ein einfacher Webserver aussehen könnte, der einfach alle Dateien aus dem `public`-Verzeichnis als statische Dateien ausliefert, unter `/hello` einen dynamisch erzeugten JSON-String und unter `/template` ein serverseitiges HTML-Template zurückliefert.

Inhalt der Datei `index.js`:

```
01  const path = require("path");
02  const express = require("express");
03  const expressNunjucks = require("express-nunjucks");
04
05  const app = express();
06
07  // Statische Dateien aus dem static-Verzeichnis
08  let staticDir = path.normalize(path.join(__dirname, "static"));
09  app.use(express.static(staticDir));
10
11  // Startseite aus serverseitigem Template erzeugt
12  let templateDir = path.normalize(path.join(__dirname, "templates"));
13  app.set("views", templateDir);
14
15  let isDev = app.get("env") === "development";
16
17  expressNunjucks(app, {
18      watch: isDev,
19      noCache:  isDev
20  });
21
22  app.get("/", (request, response) => {
23      response.render("index", {
24          title: "Startseite",
25          answer: 42,
26          message: "Hallo, Node.js!",
27      });
28  });
29
30  // Verschieden formatierte Strings für AJAX-Aufrufe, mit Prüfung welches
31  // Datenformat der Client gerne empfangen würde
32  app.get("/hello", (request, response) => {
33      response.format({
34          text:    () => response.send("Hallo, Welt!"),
35          html:    () => response.send("<p>Hallo, Welt!</p>"),
36          json:    () => response.json({message: "Hallo, Welt!"}),
37          default: () => response.status(406).send("Nicht untersütztes Format"),
38      });
39  });
40
41  // Hier fängt alles an
42  app.listen(8888, () => {
43      console.log("Webservier steht auf Port 8888 bereit!");
44  });
```

Inhalt der Datei `templates/base.html`:

```
01  <!DOCTYPE html>
02  <html>
03      <head>
04          <title>{{ title }}</title>
05          <meta charset="utf-8" />
06          <link rel="stylesheet" href="/style.css" />
07      </head>
08      <body>
09          {% block content %}
10          {% endblock %}
11      </body>
12  </html>
```

Inhalt der Datei `templates/index.html`:

```
{% extends "base.html" %}

{% block content %}
    <h1>{{ message }}</h1>
    <p>
        Die Antwort auf die Frage nach dem Leben, dem Universum und dem
        ganzen Rest lautet: {{ answer }}!
    </p>
    <p>
        Probieren Sie auch unseren <a href="/hello">Hello-Service</a>.
    </p>
{% endblock %}
```

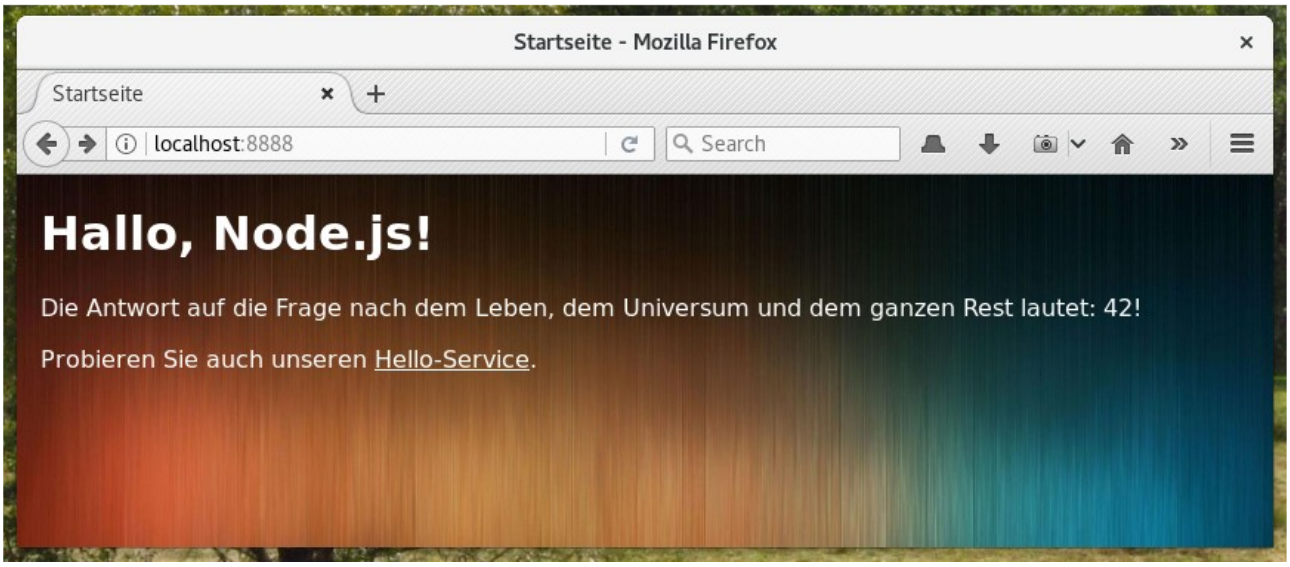

Abb. 29: Und so sieht sie aus, die fertige Webseite

Bezüglich der Anwendungsarchitektur stehen Ihnen jetzt alle Möglichkeiten offen:

- Einfache Bereitstellung statischer HTML-, JavaScript-, CSS- und sonstiger Dateien
- Dynamische Erzeugung des HTML-Codes mit serverseitigen Templates
- Verlagerung der Logik in den Client mit AJAX-Zugriffen auf den Server
- Echtzeitkommunikation mit dem Server über Websockets (hier nicht gezeigt)

Je nach Aufgabenstellung könnten dabei folgende Links ganz interessant sein:

- Einführung und Referenz zu express.js
- Tutorial für einen vollständigen REST-Service auf Basis von express.js
- Alternatives REST-Framework restify.js
- Alternatives Realtime-Framework feathers.js

8.9 Empfang und Versand von MQTT-Nachrichten mit Node.js

Im IoT-Umfeld kommt häufig das MQTT-Protokoll für den Nachrichtenaustausch zwischen eingebetteten Sensoren, Aktoren und einem Kontroll-/Überwachungsserver im Internet zum Einsatz. Es handelt sich dabei um eine für den eingebetteten Kontext angepasste Variante des Publish/Subscribe-Modells, wie Sie es auch

in Verteilte Systeme kennengelernt haben. Für die Nutzung mit Node.js benötigen Sie lediglich einen Message Broker sowie das mqtt-Modul, das Sie wie folgt installieren können:

```
$ yarn add mqtt
```

Das folgende Beispiel[89] zeigt dabei, wie Sie sich mit einem Message Broker verbinden können, um Nachrichten zu senden und zu empfangen:

```
01  const mqtt = require("mqtt");
02  let broker = mqtt.connect("mqtt://test.mosquitto.org");
03
04  broker.on("connect", () => {
05      broker.subscribe("/example");
06      broker.publish("/example", "Hallo, da bin ich!");
07  });
08
09  broker.on("message", (topic, message) => {
10      let text = message.toString();
11
12      switch (topic) {
13          case "/example":
14              console.log(`Empfangen in /example: ${text}`);
15      }
16  });
```

Ziemlich einfach, nicht? Folgendes passiert in den einzelnen Zeilen:

- **01:** Das Modul `mqtt` wird importiert
- **02:** Es wird eine Verbindung zu `mqtt://test.mosquitto.org` hergestellt
- **04:** Es wird ein Event Handler für das `connect`-Ereignis registriert. Dieser wird aufgerufen, sobald die Verbindung erfolgreich hergestellt wurde.
- **05:** Der Client registriert sich für das Topic `/example`, um künftig alle Nachrichten, die an dieses Topics gesendet werden, zu empfangen.
- **06:** Der Client sendet eine Nachricht an `/example`.
- **09:** Es wird ein Event Handler für das `message`-Ergebnis registriert. Dieser wird bei jeder empfangenen Nachricht aufgerufen und bekommnt dann den Namen des Topics sowie ein Buffer-Objekt[90] mit dem Inhalt der Nachricht übergeben.
- **10:** Die empfangenen Bytes werden in einen Textstring umgewandelt.
- **12:** Es wird geprüft, für welches Topic eine Nachricht empfangen wurde
- **14:** Die empfangene Nachricht wird auf der Konsole ausgegeben, wenn sie an das Topic `/example` gesendet wurde

8.10 Nutzung der GPIO-Pins mit Node.js

Der Linux-Kernel stellt für den Zugriff auf die GPIO-Pins ein virtuelles Filesystem Interface bereit, so dass die GPIO-Pins durch Lesen und Schreiben der Dateien im Verzeichnis `/sys/class/gpio` wie in Kapitel 4.10 beschrieben genutzt werden können. Für einfache Anwendungsfälle, wo es zum Beispiel nur darum geht, eine LED ein oder auszuschalten, genügt also schon das in Node.js eingebaute `fs`-Modul, um auf die entsprechenden Dateien zuzugreifen, wie das folgende Beispiel zeigt:

```
01  // Einfache Version mit dem eingebauten fs-Modul
02  // Nicht für den Produktiveinsatz geeignet!
```

89 Angelehnt an https://www.npmjs.com/package/mqtt#example

90 Dadurch ist sichergestellt, dass auch Binärnachrichten ausgetauscht werden können

```
03  const fs = require("fs");
04  let gpio_watcher = null;
05
06  // GPIO-Pin 5 verfügbar machen
07  let gpio5_available = new Promise((resolve, reject) => {
08      gpio_watcher = fs.watch("/sys/class/gpio", {recursive: true},
09          (eventType, filename) => {
10              if (eventType === "rename" && filename === "/sys/class/gpio/gpio5") {
11                  let fd_direction = fs.openSync("/sys/class/gpio/gpio5/direction");
12                  fs.writeFileSync(fd_direction, "out");
13                  fs.close(fd_direction);
14
15                  resolve();
16              }
17          }
18      );
19
20      let fd_export = fs.openSync("/sys/class/gpio/epxort", "w");
21      fs.writeFileSync(fd_export, "5");
22      fs.close(fd_export);
23  });
24
25  // Warten, bis der Pin verfügbar ist
26  gpio5_available.then(() => {
27      // Spannung anlegen
28      let fd_value = fs.openSync("/sys/class/gpio/gpio5/value", "w");
29      fs.writeSync(fd_value, "1");
30      fs.close(fd_value);
31
32      // Pin nicht mehr weiterverwenden
33      let fd_unexport = fs.openSync("/sys/class/gpio/unepxort", "w");
34      fs.writeFileSync(fd_unexport, "5");
35      fs.close(fd_unexport);
36
37      gpio_watcher.close();
38  });
```

Wie Sie sehen, wird aber bereits der Quellcode für dieses einfache Beispiel schon relativ komplex und man muss ziemlich aufpassen, keine Race Conditions zu erzeugen, durch die man ein entscheidendes Ereignis verpasst. Zusätzlich wird der Code sehr schnell ineffizient, wenn Sie nicht nur den Wert eines GPIO-Pins ändern sondern diesen auch auslesen wollen. Dafür müssten Sie dann permanent den Inhalt der entsprechenden Datei in einer Busy Loop prüfen, was sich aufgrund der Single Thread-Natur von JavaScript jedoch verbietet, oder sich besser mit der `select`-API des Linux-Kernels[91] beschäftigen, um auf den Hardware Interrupt des GPIO-Eingangs zu reagieren. Also langer Rede kurzer Sinn: So wie oben gezeigt, machen Sie es bitte nicht! Stattdessen nutzen Sie das `onoff`-Modul, mit dem plötzlich alles ganz einfach wird:

```
$ yarn add onoff
```

```
01  const GPIO = require("onoff").Gpio;
02
03  let gpio5 = new GPIO(5, "out");
04  gpio5.write(1);
05
06  process.on("SIGINT", () => {
07      gpio5.writeSync(0);
08      gpio5.unexport();
09
10      process.exit();
11  });
```

Und so schalten Sie die LED jeweils ein oder aus, wenn ein an einem anderen Pin angeschlossener Schalter betätigt wird:

```
01  const GPIO = require("onoff").Gpio;
02
03  let gpio5 = new GPIO(5, "out");
```

91 http://man7.org/linux/man-pages/man2/select.2.html

```
04  let gpio6 = new GPIO(6, "in");
05
06  gpio6.watch((error, gpio6_value) => {
07      let gpio5_value = (gpio5.readSync() + 1) % 2;
08      gpio5.write(gpio5_value);
09  });
10
11  process.on("SIGINT", () => {
12      gpio5.writeSync(0);
13
14      gpio5.unexport();
15      gpio6.unexport();
16
17      process.exit();
18  });
```