3.3 Cross-Development für den Raspberry Pi

Um für den Raspberry Pi ein eigenes System zu bauen, sind die folgenden Schritte notwendig:

1. Kernel herunterladen, konfigurieren, generieren
2. Userland generieren
3. Installation auf der SD-Karte

Theoretisch ist es möglich, die System- und Anwendungssoftware für den Raspberry Pi auf dem Raspberry Pi selbst zu erzeugen, also eine Hostentwicklung durchzuführen. Praktisch jedoch dauert allein das Generieren des Kernels mehrere Stunden, sodass eine Host-/Target-Entwicklung effizienter ist. Je nach Leistungsfähigkeit des Hostsystems kann die Generierung des Kernels auf unter zehn Minuten reduziert werden.

Für die Host-/Target-Entwicklung wird eine Cross-Development-Toolchain benötigt, die aus Cross-Compiler, Cross-Linker, Cross-Libraries und Cross-Debugger besteht. Als Cross-Development-Toolchain wird im Allgemeinen gern auf die GNU-Toolchain gesetzt, die parallel zu den Hostwerkzeugen (also Compiler für das Hostsystem selbst) installiert wird. Allerdings ist die Installation häufig nicht trivial, sodass wir zunächst auf vorbereitete Pakete zurückgreifen beziehungsweise später ein Werkzeug zur automatisierten Erstellung der Toolchain einsetzen.

Im Folgenden wollen wir das Selbstbausystem auf den Raspberry Pi portieren. Daran lässt sich sehr gut der Umgang mit Cross-Entwicklungswerkzeugen demonstrieren. Wir fangen wieder mit dem Kernel an, um danach das Userland aufzusetzen. Die Entwicklung selbst soll unterhalb des Verzeichnisses `~/embedded/raspi/` stattfinden, das wir bereits angelegt haben.

3.3.1 Cross-Generierung Kernel

Um einen Kernel für den Raspberry Pi zu bauen, ist Folgendes zu tun (siehe Beispiel 3-8):

1. Toolchain installieren
2. Kernelquellen per git herunterladen
3. Kernel konfigurieren
4. Generieren

Für die Cross-Generierung des Kernels wird als Erstes eine Cross-Entwicklungsumgebung benötigt, die sich auf einem Ubuntu per »apt-get install gcc-arm-linux-gnueabi« leicht installieren lässt. Auch wenn diese (in der unter Ubuntu 12.04 zur Verfügung stehenden Version) weder Userland noch Bootloader generiert, wollen wir sie zunächst für den Kernel einsetzen.

Leider können Sie die bereits installierten Kernelquellen nicht weiter benutzen, da der über [http://www.kernel.org] herunterladbare Kernel in der Version 3.10.9 von Linus Torvalds den Raspberry Pi nicht vollständig unterstützt. Nehmen Sie daher einen bereits vorkonfektionierten Quellcode, der sich per `git` über Github unterhalb von `~/embedded/raspi/` installieren lässt. Der Umgang mit `git` — falls unbekannt — ist übrigens im Anhang C, *Git im Einsatz* in Kurzform beschrieben.

Vorteilhafterweise bringt der vorkonfektionierte Kernel die in Tabelle 3-7 aufgeführten Generierungs-Targets für den Raspberry Pi mit, die die Kernelkonfiguration erheblich vereinfachen. Der mit dem Target `bcmrpi_defconfig` generierte Kernel kann übrigens auch mit einem Raspbian eingesetzt werden, wobei einige nicht elementare Treiber als Module generiert werden.

Target	Bedeutung
bcmrpi_defconfig	Konfiguration für ein normales System
bcmrpi_cutdown_defconfig	Auf die wesentliche Funktionen reduziert, z.B. kein Debugging, Tracing oder Firewalling
bcmrpi_quick_defconfig	Schlanker Kernel, kaum Module, wenig Features
bcmrpi_emergency_defconfig	Schlanker Kernel für das Notsystem

Tabelle 3-7 *Vordefinierte Kernelkonfigurationen für den Raspberry Pi*

Für die Konfiguration und die Generierung des Linux-Kernels mit einem Cross-Compiler wird `make` mit dem zusätzlichen Parameter `ARCH=arm` aufgerufen [QuKu2011a]. Zusätzlich muss noch der Parameter `CROSS_COMPILE=arm-linux-gnueabi-` angegeben werden. Nur so werden die richtigen Cross-Compile-Werkzeuge gefunden, deren Namen sich aus den im Parameter stehenden Namensvorsatz »arm-linux-gnueabi-« und dem Werkzeugnamen selbst ergeben. Der Name des Cross-Compilers `gcc` lautet demnach »arm-linux-gnueabi-gcc«. Beispiel 3-8 zeigt im Detail die Kommandos, die in einem Terminal aufzurufen sind, um auf einem Ubuntu einen Raspberry Pi-Kernel zu generieren. Dabei müssen Sie insbesondere für das »Klonen« des Kernels (Herunterladen des gepatchten Kernelquellcodes per git) einige Minuten Zeit einplanen.

Beispiel 3-8 *Kommandos, um den Raspberry-Kernel auf einem Ubuntu zu generieren*

```
quade@felicia:~/embedded>
quade@felicia:~/embedded> cd raspi
quade@felicia:~/embedded/raspi> sudo apt-get \
   install gcc-arm-linux-gnueabi ncurses-dev
...
quade@felicia:~/embedded/raspi> git clone \
   https://github.com/raspberrypi/linux.git
...
quade@felicia:~/embedded/raspi> cd linux
quade@felicia:~/embedded/raspi/linux> git branch -a
* rpi-3.6.y
  remotes/origin/HEAD -> origin/rpi-3.6.y
  remotes/origin/master
  remotes/origin/rpi-3.10.y
  remotes/origin/rpi-3.2.27
  remotes/origin/rpi-3.6.y
  remotes/origin/rpi-3.8.y
  remotes/origin/rpi-3.9.y
  remotes/origin/rpi-patches
quade@felicia:~/embedded/raspi/linux> git checkout rpi-3.10.y
quade@felicia:~/embedded/raspi/linux> make ARCH=arm bcmrpi_defconfig
quade@felicia:~/embedded/raspi/linux> make ARCH=arm \
   CROSS_COMPILE=arm-linux-gnueabi- -j4
quade@felicia:~/embedded/raspi/linux>
```

Der Parameter »-j4« sorgt wieder für die Parallelverarbeitung auf einem Multicore-System. Der Kernel befindet sich nach einer erfolgreich durchgelaufenen Generierung im Linux-Quellcodeverzeichnis unter `~/embedded/raspi/linux/arch/arm/boot/zImage`.

Die Installation des Kernels auf der SD-Karte des Raspberry Pi wird nach der Generierung des Userlands beschrieben.

Kernel auf dem Raspberry Pi selbst bauen

Auch wenn es lange dauert, wenn der Raspberry Pi mit einem Raspbian betrieben wird, lässt sich der Kernel auf diesem nativ kompilieren. Loggen Sie sich dazu auf dem Raspberry Pi mit dem Login »pi« (Passwort »raspberry«) ein und werden Sie Superuser (`sudo su`). Sie können dann die gepatchten Kernelquellen installieren, konfigurieren und genererieren. Beispiel 3-9 zeigt Ihnen die dazu benötigten Befehle und zusätzlich noch die Kommandos, mit denen Kernel und Module installiert werden. Da der Kernel zusätzlich zum Standardkernel installiert wird, wird noch die Datei `/boot/config.txt` modifiziert. Dieser Schritt darf allerdings nur einmalig durchgeführt werden .

```
pi@raspberrypi ~ $ sudo su
root@raspberrypi:/home/pi# cd /usr/src
root@raspberrypi:/usr/src# git clone \
   https://github.com/raspberrypi/linux.git -b rpi-3.10.y
root@raspberrypi:# cd linux
root@raspberrypi:# make bcmrpi_defconfig
root@raspberrypi:# make menuconfig
root@raspberrypi:# make
root@raspberrypi:# make modules_install
root@raspberrypi:# cp arch/arm/boot/zImage /boot/linux-3.10.y
root@raspberrypi:# echo "kernel=linux-3.10.y" >>/boot/config.txt
```

Beispiel 3-9
Kernel auf dem Raspberry Pi selbst bauen

3.3.2 Cross-Generierung Userland

Um das Userland für den Raspberry Pi zu bauen, ist Folgendes zu tun:

1. Das für Qemu erstellte Userland kopieren
2. Das Init-Skript `rcS` anpassen
3. Die `inittab` anpassen
4. Cross-Entwicklungsumgebung erstellen
5. Busybox für die Cross-Entwicklung konfigurieren und Busybox kompilieren
6. Generierungsskript `mkrootfs.sh` anpassen
7. Das Userland per `mkrootfs.sh` zusammenbauen

Um das bereits in Abschnitt 3.2 vorbereitete Userland auf dem Raspberry Pi einsetzen zu können, muss es angepasst werden. Dazu kopieren Sie als Erstes die Quelldateien in das Verzeichnis `raspi`, legen für Skripte ein eigenes Verzeichnis an und kopieren dann noch das Generierungsskript `mkrootfs.sh` in das neue Verzeichnis:

```
quade@felicia:~> cd embedded/raspi/
quade@felicia:~/embedded/raspi> cp -r ../qemu/userland/ .
quade@felicia:~/embedded/raspi> mkdir scripts
quade@felicia:~/embedded/raspi/scripts> cp ../../qemu/mkrootfs.sh .
quade@felicia:~/embedded/raspi/scripts>
```

Die Anpassungen des Userlands betreffen zunächst die Dateien `inittab` (zur Unterstützung der seriellen Schnittstelle) und `rcS` (zur Konfiguration des Netzwerkinterface). Außerdem müssen noch zusätzliche Gerätedateien angelegt und die Busybox cross-kompiliert werden.

Beispiel 3-10 *Einfache Inittab für den Raspberry Pi <inittab>*

```
::sysinit:/etc/rcS
::askfirst:/bin/ash
ttyAMA0::askfirst:/bin/ash
```

Um im Userland nicht nur mit einer über USB angeschlossenen Tastatur und einem per HDMI angeschlossenen Monitor Ein- und Ausgaben tätigen zu können, soll die im Bereich eingebetteter Systeme häufig anzutreffende serielle Schnittstelle unterstützt werden. Dazu wird `init` über die `inittab` so konfiguriert, dass eine zweite Shell auf der seriellen Schnittstelle gestartet wird. Beispiel 3-10 zeigt die modifizierte Konfigurationsdatei, die um eine Zeile ergänzt wurde. Dieser entnehmen Sie, dass die serielle Schnittstelle auf dem Raspberry Pi über den Namen `ttyAMA0` angesprochen wird. Die zugehörige Gerätedatei `/dev/ttyAMA0` wird später über das Skript `mkrootfs.sh` angelegt.

```
quade@felicia:~/embedded/raspi> cd userland/target
quade@felicia:~/embedded/raspi/userland/target> gedit inittab &
```

Die Netzwerkkonfiguration muss angepasst werden, da das Ethernet-Interface auf dem Raspberry Pi eine Initialisierungszeit benötigt. Vorher lässt sich keine IP-Adresse konfigurieren. Die IP-Adresse selbst ist ebenfalls auszutauschen. Anstelle der für Qemu verwendeten IP-Adresse benutzen Sie eine aus Ihrem Netz. Die um die Initialisierungszeit von drei Sekunden (`sleep 3`) erweiterte Datei `rcS` inklusive der Änderung auf eine andere IP-Adresse (`192.168.178.99`) finden Sie unter Beispiel 3-11. Auskommentiert ist übrigens eine Zeile, bei der anstelle einer festen IP-Adresse ein DHCP-Client gestartet wird, der sich von einem heute meist im Netzwerk vorhandenen DHCP-Server die IP-Adresse dynamisch zweisen lässt. Falls Sie die Konfiguration per DHCP wünschen, müssen Sie die Zeile mit `ifconfig` durch Einfügen eines »#« am Zeilenanfang auskommentieren und die nachfolgende Zeile durch Löschen des ersten Zeichens aktivieren.

```
quade@felicia:~/embedded/raspi/userland/target> gedit rcS &
```

Im nächsten Schritt muss Busybox für den Raspberry Pi cross-kompiliert werden. Da die mit `apt-get` installierte Cross-Entwicklungsumgebung kein Userland generieren kann, ist eine eigene Umgebung aufzusetzen. Das ist allerdings sehr komplex und fehlerbehaftet. Die Raspberry Pi-Homepage schlägt hierfür `crosstool-ng` vor, besser und einfacher geht das allerdings über den Systembuilder `buildroot`, der von uns ohnehin in Kapitel 4 eingesetzt wird.

```
#!/bin/ash

mount -t proc none /proc
mount -t sysfs none /sys
sleep 3
ifconfig eth0 192.168.178.99
# udhcpc -s /etc/simple.script
httpd -c /etc/httpd.conf
```

Beispiel 3-11
Einfaches Skript zur Systemkonfiguration <rcS>

Mit den folgenden Kommandos laden Sie sich über das Internet den Buildroot-Quellcode herunter, konfigurieren diesen und lassen über diesen mithilfe des Targets `make toolchain` die Toolchain generieren:

```
quade@felicia:~/embedded/raspi/userland/target> cd ../..
quade@felicia:~/embedded/raspi> wget \
    http://www.buildroot.net/downloads/buildroot-2013.05.tar.bz2
quade@felicia:~/embedded/raspi> tar xvf buildroot-2013.05.tar.bz2
quade@felicia:~/embedded/raspi/buildroot-2013.05> make rpi_defconfig
quade@felicia:~/embedded/raspi/buildroot-2013.05> make toolchain
```

Insbesondere der Download dauert eine geraume Zeit. Wenn die Generierung der Toolchain erfolgreich verläuft, liegen im Verzeichnis `~/embedded/raspi/buildroot-2013.05/output/host/usr/bin/` die notwendigen Programme. Unter `~/embedded/raspi/buildroot-2013.05/output/host/usr/arm-buildroot-linux-uclibcgnueabi/sysroot` befinden sich die zugehörigen Headerdateien und Bibliotheken. Verläuft die Generierung nicht erfolgreich, liegt das meistens daran, dass erforderliche Pakete nicht installiert sind. In diesem Fall ist es notwendig, die Fehlermeldung von Buildroot genau zu studieren und die fehlenden Pakete auf dem Entwicklungsrechner mithilfe von `apt-get` nachzuinstallieren.

Damit Busybox cross-kompiliert wird, rufen Sie im Unterverzeichnis `~/embedded/raspi/userland/busybox-1.21.1/make menuconfig` auf. Wählen Sie `Busybox Settings` und danach `Build Options`. Unter `Cross Compiler Prefix` tragen Sie jetzt `arm-linux-` ein und unter `Path to Sysroot` den Pfad zu den Headerdateien und Bibliotheken der Cross-Toolchain, die mit `sysroot` bezeichnet werden. Wichtig dabei: Sie müssen den Pfad absolut und nicht relativ angeben! In meinem Fall ist das `/home/quade/embedded/raspi/buildroot-2013.05/output/host/usr/arm-buildroot-linux-uclibcgnueabi/sysroot/` (Abb. 3-9). Vergessen Sie nicht, meinen Loginnamen im Pfad durch den Ihrigen auszutauschen. Sollte der Pfad zu Sysroot falsch eingegeben sein, wird `make` mit einem Fehler abbrechen und dabei melden, dass es die Datei `limits.h` nicht finden kann.

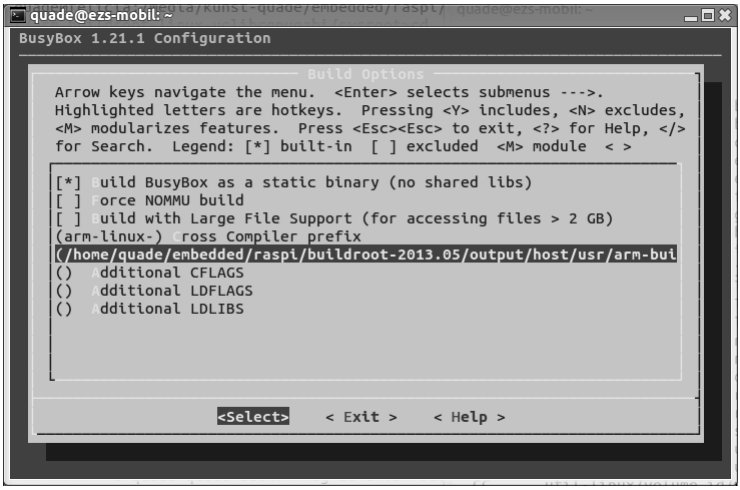

Abb. 3-9
Einstellungen in Busybox zur Cross-Kompilation

Falls Sie anstelle der festen IP-Adresse DHCP verwenden wollen, müssen Sie hierzu bei der Busybox-Konfiguration unter `Networking Utilities` den udhcp-client (`udhcpc`) auswählen. Udhcpc ruft, wenn es eine IP-Adresse zugewiesen bekommen hat, ein Skript auf, das dann für das Setzen der IP-Adresse zuständig ist. Busybox stellt im Verzeichnis `examples/udhcp/` das Skript `simple.script` zur Verfügung, das für unsere Zwecke ausreichend ist.

Außerdem muss vor dem Aufruf von `make` noch die Umgebungsvariable »PATH« so angepasst werden, dass der Pfad, in dem sich Cross-Compiler, Cross-Linker und die übrigen Cross-Entwicklungswerkzeuge befinden (`~/embedded/raspi/buildroot-2013.05/output/`), angehängt wird. Nach diesen Vorbereitungen cross-generiert ein `make` die Busybox und ein `make install` installiert die Systemprogramme im Verzeichnis `_install`.

```
quade@felicia:~/embedded/raspi> cd userland/busybox-1.21.1
quade@felicia:~/embedded/raspi/userland/busybox-1.21.1> export \
   PATH=$PATH:~/embedded/raspi/buildroot-2013.05/output/host/usr/bin/
quade@felicia:~/embedded/raspi/userland/busybox-1.21.1> make menuconfig
...
# [Busybox Settings][Build Options] Cross-Werkzeug-Präfix
# [Busybox Settings][Build Options] Absoluten Pfad zu Sysroot eintragen
# [Networking Utilities][udhcpc] DHCP-Client auswählen
...
quade@felicia:~/embedded/raspi/userland/busybox-1.21.1> make
quade@felicia:~/embedded/raspi/userland/busybox-1.21.1> make install
```

Damit ist Busybox cross-generiert, das Userland kann zusammengebaut werden. Allerdings sind zuvor noch kleine Änderungen in `mkrootfs.sh` notwendig. Es muss noch die zusätzliche Gerätedatei `/dev/ttyAMA0` (Majornummer 204, Minornummer 64) angelegt werden, sodass Aus- und

Eingaben über die serielle Schnittstelle des Raspberry Pi möglich sind. Außerdem erwartet der Kernel das Programm `init` nicht wie bei der emulierten Variante unter `/sbin/init`, sondern im Root-Verzeichnis. Um das Problem zu lösen, legen wir mit `ln -s sbin/init loop/init` einen symbolischen Link an. Als Drittes schließlich muss für `udhcpc` (so er denn von Ihnen verwendet wird) noch das Skript kopiert werden, das aufgerufen wird, sobald der DHCP-Client eine IP-Adresse zugewiesen bekommen hat. Tragen Sie also die nachfolgenden Zeilen hinter `# MARK C` ein:

```
# MARK C
sudo mknod loop/dev/ttyAMA0 c 204 64
sudo ln -s sbin/init loop/init
sudo install -m 755 busybox-1.21.1/examples/udhcp/simple.script loop/etc
```

Last, but not least dürfen Sie nicht vergessen, den im Skript vorkommenden Pfadnamen »qemu« durch »raspi« zu ersetzen! Hier die Kommandos dazu im Überblick:

```
quade@felicia:~/embedded/raspi/userland/busybox-1.21.1> cd ../..
quade@felicia:~/embedded/raspi> cd scripts
quade@felicia:~/embedded/raspi/scripts> gedit mkrootfs.sh & # Änderungen
vornehmen
# ACHTUNG: Pfadnamen "qemu" durch "raspi" ersetzen
#          cd $MAINDIR/raspi/userland
# Ergänzungen vornehmen
quade@felicia:~/embedded/raspi/scripts> ./mkrootfs.sh # Userland generieren
```

3.3.3 Installation auf dem Raspberry Pi

Zur Installation des Selbstbau-Linux auf einer SD-Karte sind die folgenden Schritte notwendig:

1. Bootpartition erstellen, entweder im Selbstbau oder durch Installation von Raspbian
2. Eigenbau-Kernel auf der Bootpartition installieren
3. Userland auf der Systempartition installieren

Sind Kernel und Busybox cross-kompiliert und das Userland zusammengebaut, können die Komponenten auf einer SD-Karte installiert werden. Die SD-Karte bekommt zwei Partitionen: Auf der ersten, der Bootpartition, befinden sich die Bootprogramme und der Kernel, auf der zweiten Partition (Systempartition) befindet sich das Rootfilesystem beziehungsweise das Userland.

Bootpartition

Der Raspberry Pi setzt zwingend eine SD-Karte voraus, deren erste Partition mit einem FAT32-Filesystem (bei dem Partitionierungsprogramm `fdisk` ist die ID »c« einzustellen) formatiert ist. Auf dieser Partition müssen sich die in Tabelle 3-8 gelisteten Dateien befinden. Die erste Partition hat eine Größe von typischerweise 50 bis 100 MBytes.

Tabelle 3-8
Dateien auf der Bootpartition des Raspberry Pi

Name	Bedeutung
bootcode.bin	GPU-Firmware, die für das Booten zuständig ist
start.elf	GPU-Firmware, die für das Laden des Kernels zuständig ist
LICENCE.broadcom	Lizenzbestimmungen für die Nutzung des Bootloader-Codes
kernel.img	Linux-Kernel

Sie können diese Partition vorzugsweise wie in Kasten auf Seite 73 gezeigt von Grund auf selbst erstellen, oder alternativ durch Installation eines Raspbian wie in Abschnitt 2.3.1. Bei beiden Methoden stimmt übrigens die im Bootsektor hinterlegte Größe der zweiten Partition nicht mit der realen Größe überein. Der Funktion tut das aber keinen Abbruch. Die SD-Karte ist partitioniert und für die Aufnahme der eigentlichen Daten eingerichtet. Sie sollten übrigens sicherheitshalber die SD-Karte einmal kurz aus- und danach wieder einstecken. Das stellt sicher, dass die durch Ihre Aktionen geänderte Partitionstabelle neu eingelesen wird. Nach dem Einstecken der SD-Karte sollten sowohl die Boot- als auch die Systempartition eingehängt worden sein, die Bootpartition dabei unter dem Verzeichnis `/media/boot/`. Ist die Bootpartition unter einem anderen Verzeichnis eingehängt, tauschen Sie in den folgenden Erläuterungen die Pfadangabe entsprechend aus.

Sie müssen jetzt den selbstkompilierten Kernel auf die Bootpartition kopieren. Sie können diesen unter dem Standardnamen `kernel.img` ablegen oder aber auch einen individuellen Namen, beispielsweise `zImage`, wählen. Falls Sie den individuellen Namen vorziehen, benötigen Sie zusätzlich eine Datei `config.txt`, in der Sie die Zeile »kernel=zImage« eintragen. Nach dem Kopieren des Kernels kann die Bootpartition wieder ausgehängt werden:

```
quade@felicia:~/embedded/raspi> mv \
   /media/boot/kernel.img /media/boot/kernel.img.org

quade@felicia:~/embedded/raspi> cp \
   linux/arch/arm/boot/zImage /media/boot/kernel.img
quade@felicia:~/embedded/raspi> echo "kernel=zImage" \
   >>/media/boot/config.txt
quade@felicia:~/embedded/raspi> sudo umount /media/boot/
```

SD-Karten-Installation from scratch

Zur Partitionierung der SD-Karte wird das Programm `fdisk` eingesetzt, das als Parameter den Namen der Gerätedatei mitbekommt, über die auf die SD-Karte zugegriffen wird. Typischerweise ist das `/dev/sdc` oder `/dev/sdb`. Aber Vorsicht: Wer hier die falsche Gerätedatei eingibt, zerstört sein Hostsystem. Um die richtige Gerätedatei zu identifizieren, verfahren Sie wieder wie in Abbildung 2-9 gezeigt. Die letzten Einträge der nach Eingabe von `lsblk` erscheinenden Tabelle sollten die beiden Partitionen Ihrer SD-Karte sein. Gleich zu Anfang steht das zugehörige Device, beispielsweise `/dev/sdb1` für die Bootpartition und `/dev/sdb2` für das Rootfilesystem. Bei Ihnen könnte beispielsweise auch `/dev/sdc1` und `/dev/sdc2` ausgegeben sein. In der darüber stehenden Zeile ist die Gerätedatei aufgeführt, die die gesamte SD-Karte bezeichnet, beispielsweise `/dev/sdb`.

Starten Sie `fdisk` unter Angabe der Gerätedatei, die für die SD-Karte steht. Sollte es auf der verwendeten SD-Karte bereits irgendwelche Partitionen geben, löschen Sie diese mit dem Kommando »d«. Falls mehrere Partitionen vorhanden sind, müssen Sie noch die Nummer der zu löschenden Partition angeben. Sind alle Partitionen gelöscht, legen Sie durch Eingabe des Buchstabens »n« eine neue Partition vom Typ »primary« an. Diese Partition sollte mindestens 50 MByte groß sein. Der Partitionstyp ist danach auf »c« zu setzen, das einer FAT32-Partition entspricht. Beispiel 3-12 zeigt die Befehlssequenzen für eine unter `/dev/sdb` erreichbare SD-Karte, bei der per `fdisk` zunächst zwei vorhandene Partitionen gelöscht und danach zwei neue Partitionen angelegt wurden. Durch Eingabe des Buchstabens »p« zeigt Ihnen `fdisk` die jeweilige Partitionstabelle an, die aber erst mit Eingabe eines »w« wirklich geschrieben wird.

Beispiel 3-12
Befehlssequenz zum Anlegen der SD-Karte-Partitionen

```
root@felicia:~# fdisk /dev/sdb

Befehl (m für Hilfe): p

Platte /dev/sdb: 1977 MByte, 1977614336 Byte
61 Köpfe, 62 Sektoren/Spur, 1021 Zylinder, zusammen 3862528 Sektoren
Einheiten = Sektoren von 1 × 512 = 512 Bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Festplattenidentifikation: 0x00047c7a

Gerät     boot.    Anfang       Ende     Blöcke   Id  System
/dev/sdb1            2048     206847     102400    c  W95 FAT32 (LBA)
/dev/sdb2          206848    3862527    1827840   83  Linux

Befehl (m für Hilfe): d
Partitionsnummer (1-4): 1

Befehl (m für Hilfe): d
Partition 2 ausgewählt
```

```
Befehl (m für Hilfe): n
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p
Partitionsnummer (1-4, Vorgabe: 1): 1
Erster Sektor (2048-3862527, Vorgabe: 2048): 2048
Last Sektor, +Sektoren or +size{K,M,G} (2048-3862527,
Vorgabe: 3862527): +100M

Befehl (m für Hilfe): t
Partition 1 ausgewählt
Hex code (L um eine Liste anzuzeigen): c
Der Dateisystemtyp der Partition 1 ist nun c (W95 FAT32 (LBA))

Befehl (m für Hilfe): n
Partition type:
   p   primary (1 primary, 0 extended, 3 free)
   e   extended
Select (default p): p
Partitionsnummer (1-4, Vorgabe: 2): 2
Erster Sektor (206848-3862527, Vorgabe: 206848): 206848
Last Sektor, +Sektoren or +size{K,M,G} (206848-3862527,
Vorgabe: 3862527): 3862527

Befehl (m für Hilfe): w
Die Partitionstabelle wurde verändert!

Rufe ioctl() um Partitionstabelle neu einzulesen.
```

Sobald die SD-Karte partitioniert ist, muss noch ein Filesystem angelegt werden. Dazu verwenden Sie die Kommandos `mkfs.vfat` für die Bootpartition und `mkfs.ext2` für die Systempartition. Achten Sie wieder darauf, die richtige Gerätedatei anzugeben, indem Sie das »x« im Namen der Gerätedatei durch den zugehörigen Laufwerksbuchstaben ersetzen:

```
quade@felicia:~/embedded> sudo mkfs.vfat -n boot /dev/sdx1
quade@felicia:~/embedded> sudo mkfs.ext2 -L rootfs -i 1024 /dev/sdx2
```

Jetzt können die Partitionen eingehängt werden. Auf die erste Partition müssen dann die Boot-Dateien (Tabelle 3-8) kopiert werden. Dazu laden Sie die Firmware über [https://github.com/raspberrypi/firmware] herunter und packen sie unter `~/embedded/raspi/` aus. Die gesuchten Dateien finden sich im Verzeichnis `firmware-master/boot/`:

```
quade@felicia:~/embedded> sudo mount /dev/sdX1 /mnt
quade@felicia:~/embedded> cd files
quade@felicia:~/embedded/files> \
   wget https://github.com/raspberrypi/firmware/archive/master.zip
quade@felicia:~/embedded> cd ../raspi
```

```
quade@felicia:~/embedded/raspi> unzip ../files/master.zip
...
quade@felicia:~/embedded/raspi> cd firmware-master/boot
quade@felicia:~/embedded/raspi/firmware-master/boot> \
   cp bootcode.bin start.elf LICENCE.broadcom /mnt/
```

Systempartition

Um das Userland zu installieren, kopieren Sie das zugehörige Image per `dd` auf die ausgehängte, zweite Partition:

```
quade@felicia:~/embedded/raspi> sudo umount /dev/sdx2
quade@felicia:~/embedded/raspi> sudo dd if=userland/rootfs.img of=/dev/sdx2
...
```

Damit ist die SD-Karte vorbereitet und kann im Raspberry Pi getestet werden. Stecken Sie die vorbereitete SD-Karte in den Minicomputer und schalten Sie diesen ein. Beobachten Sie die Ausgaben des Raspberry Pi. Wenn alles gut gegangen ist, müsste wieder eine Shell Ihre Kommandoeingabe ermöglichen. Per `ifconfig` können Sie sich die IP-Adresse ausgeben lassen. Mit dieser können Sie dann in einem Browser hoffentlich auf den Webserver zugreifen.

Wenn's schiefgeht ...

Sollte der Zugriff auf den Webserver nicht gelingen, gilt es herauszufinden, ob die Probleme im Kernel oder im Userland liegen. Installieren Sie dazu wieder den ursprünglichen Kernel, indem Sie auf der Bootpartition der SD-Karte den unter dem Namen `kernel.img.org` geretteten Kernel unter `kernel.img` ablegen. Stecken Sie dazu wieder die SD-Karte in den Entwicklungsrechner, wo diese typischerweise automatisch (in meinem Fall unter `/media/boot/`) gemountet wird. Danach können Sie den Kernel über die Konsole per `cp` zurückkopieren.

```
quade@felicia:~> cp /media/boot/kernel.img.org /media/boot/kernel.img
```

Stecken Sie die SD-Karte wieder in den Raspberry Pi. Sollte das System danach wie gewünscht funktionieren, liegen die Probleme wohl im Kernel. Zeigt sich jedoch das gleiche Symptom wie mit dem selbst gebautem Kernel, ist das Problem im Userland zu suchen.

Probleme mit dem Kernel

Bleibt die Meldung »Uncompressing linux« aus, ist der Kernel nicht richtig auf der SD-Karte in der Bootpartition installiert. Überprüfen Sie, ob in der Bootpartition die Datei »kernel.img« existiert und diese auch identisch mit der von Ihnen generierten Version ist.

Haben Sie für den Kernel nicht `kernel.img`, sondern einen individuellen Namen gewählt? Vielleicht fehlt dann der Eintrag in `config.txt`?

Bleibt der Kernel beim Booten stecken, durchforsten Sie die bis dahin ausgegebenen Logmeldungen. Diese geben typischerweise darüber Aufschluss, welcher Treiber beispielsweise fehlt. Ein häufiger Fehler ist das Fehlen des Treibers für das Filesystem (ext2 und ext4) oder für das Executable-Format (elf).

Probleme im Userland

Meldet der Kernel, dass er das Rootfilesystem nicht mounten kann, ist das Userland eventuell mit einem Filesystem erstellt worden, für das im Kernel keine Unterstützung existiert. Sie können das verwendete Filesystem am einfachsten identifizieren, wenn Sie die SD-Karte in den Entwicklungsrechner stecken und dann auf der Konsole den Befehl `mount` eingeben. Damit werden alle eingehängten Filesysteme inklusive ihres Typs ausgegeben. Überprüfen Sie anhand der Kernelkonfiguration, ob das Filesystem auf der SD-Karte vom Kernel unterstützt wird.

Konnte das Rootfilesystem eingehängt werden, aber deuten die nachfolgenden Meldungen auf dem Bildschirm des Raspberry Pi an, dass die in `rcS` spezifizierten Programme nicht gestartet werden konnten, ist das Userland möglicherweise nicht mit dem richtigen Cross-Compiler übersetzt worden. Um dies zu überprüfen, stecken Sie wiederum die SD-Karte in den Host, sodass die Partitionen gemountet werden. Wechseln Sie per `cd` in einem Terminal in das Rootfilesystem und überprüfen Sie dort mithilfe des Kommandos `file` den Typ des ausführbaren Programms `busybox`. Dieses muss ein Elf-Executable für ARM sein.

```
quade@felicia:/media/rootfs/bin$ file busybox
busybox: ELF 32-bit LSB executable, ARM, version 1 (SYSV), statically
linked, stripped
```

Bekommen Sie keine Shell, dürften die Einträge in der Inittab nicht korrekt sein. Hier ist möglicherweise das falsche Gerät eingetragen? Typischerweise sollte das für die serielle Schnittstelle des Raspberry Pi das Gerät `ttyAMA0` sein.

3.4 Bootloader »Das U-Boot«

Um das U-Boot auf dem Raspberry Pi zu installieren, sind die folgenden Schritte notwendig:

1. Quellcode herunterladen und Bootloader generieren
2. Programm zur Headergenerierung herunterladen
3. Bootloader mit Header versehen
4. Installation des mit Header versehenen Bootloaders auf die Bootpartition