Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 139

Himbeertauglich

Der Kernel für den Raspberry Pi wird gut gepflegt, hinkt aber dem Mainline-Kern hinterher. Dabei ist der aktuelle Linux-Kernel bereits voll himbeertauglich, sodass Sie per Cross-Compile im Handumdrehen einen Betriebssystemkern für den Single Board Computer generieren. Jürgen Quade

Der Autor

Jürgen Quade, Professor an der Hochschule Niederrhein, gibt auch für Unternehmen Schulungen zu den Themen Treiberprogrammierung und Embedded Linux.

Der Raspberry Pi mit seinem Betriebssystem Pi OS ist definitiv ein Erfolgsmodell. Wen wundert es also, dass es regelmäßig neue Hardwareversionen mit passgenauer Software gibt? Tatsächlich pflegt die Raspberry Pi Foundation ein eigenes Repository für die Kernel-Quellen, die ähnlich wie bei Apple für die eigene Hardware optimiert sind. Neben neuen Gerätetreibern finden sich im Repo vorgefertigte Konfigurationen, um einfach einen eigenen Kernel zu generieren. Eine sehr gute Anleitung für Kernel-Bäcker garniert das Ganze.

Die braucht man allerdings auch. Schließlich gibt es nicht weniger als fünf Basisversionen der Hardware, die wiederum jeweils in diversen Varianten daherkommen. In Summe sind das 17 unterschiedliche Typen, die neben der 32-Bit-Architektur zum Großteil (und beim letzten Neuzugang Raspberry Pi 5 ausschließlich) 64 Bit unterstützen. Nur mit 64 Bit lassen sich vernünftig mehr als 4 GByte Hauptspeicher ansteuern.

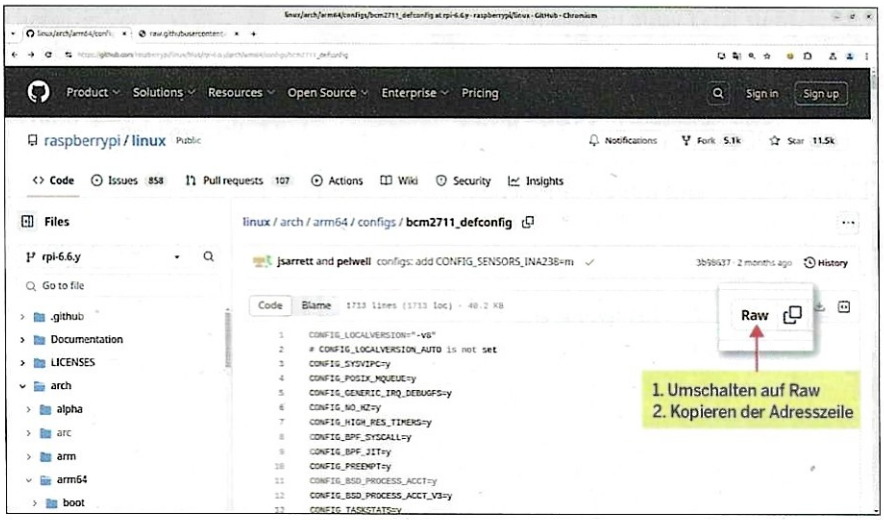
Pflegehinweise

Generell ist der Pi-OS-Kernel also bestens gepflegt, hat aber den Nachteil, dem Mainline-Kernel versionstechnisch hinterherzuhinken. Tatsächlich gab es in den Anfangstagen der Himbeere keine Alternative zum Foundation-Kernel. Ohne spezifische Anpassungen an die Hardware bootete kein Linux-Kernel.

Abgesehen vom Devicetree sieht das heute jedoch anders aus. Tatsächlich lässt sich der Mainline-Kernel von Linus Torvalds ohne Weiteres auf dem Einplatinenrechner starten und betreiben.

Wer also einen aktuellen Kernel auf dem Raspberry Pi, vorzugsweise einem Modell 4 oder 5, ausprobieren will, der bootet das Gerät und installiert die notwendigen Entwicklungswerkzeuge. Die Verwendung eines RasPi 4 oder 5 bietet den charmanten Nebeneffekt, dass Sie keine ganze Nacht für das Generieren des Kernels verplanen müssen, sondern mit rund zwei Stunden Übersetzungszeit auskommen. Alternativ lässt sich das Ganze noch abkürzen, wenn Sie einen leistungsfähigen PC zur Hand haben und etwas Cross-Compiling betreiben – doch dazu später mehr.

Listing 1 zeigt die Kommandos, um den Raspberry Pi startklar zu machen. Die Paketverwaltung Apt bestückt die Maschine dabei mit Compiler, Make, Werkzeugen zur lexikalischen Analyse und Bibliotheken zum Umgang mit Zertifikaten und Executable-Formaten.



1 **Dank Raw-Modus lassen sich die Dateien per Wget kopieren.**

Quellensuche

Davon unabhängig benötigen Sie logischerweise den Quellcode des Linux-Kernel selbst. Der Foundation-Kernel lässt sich mittels `git clone` auf den Raspberry Pi laden. Um aber den aktuellen Mainline-Kernel zu generieren, greifen Sie auf das Repository von Linus Torvalds selbst zu, das sie auf Kernel.org finden.

Am besten wechseln Sie auf dem Raspberry Pi in das Verzeichnis `/usr/src/` und klonen dann die Quellen. Rufen Sie Git ohne weitere Option auf, dann belastet das Ihren Raspberry Pi mit der gesamten Linux-Historie. In den meisten Fällen wollen Sie aber lediglich eine bestimmte, insbesondere die jüngste und aktuellste Version haben.

Das gelingt entweder mithilfe der Option `--depth=1`, mit der Sie die aktuellste Entwicklungsversion ziehen, oder aber über `--branch` *Branch*, womit Sie eine spezifische Version erhalten. Listing 2 zeigt als Beispiel das Kommando, mit dem Sie die Kernel-Version 6.13 herunterladen. Dabei geht das Exempel beim Chown-Kommando in der dritten Zeile davon aus, dass Sie unter dem Standard-Usernamen *pi* arbeiten. Der Mainline-Kernel unterscheidet sich vom Foundation-Kernel nicht nur durch die Versionsnummer, sondern auch durch das Fehlen von Default-Konfigurationen, einem System zum Umgang mit Devicetree-Overlays (siehe Tabelle Kernel-Unterschiede) und einem veralteten Devicetree selbst. Aber an dieser Stelle bedienen Sie sich frei nach dem Motto, dass ja alles Open Source ist, beim Foundation-Kernel.

Kernel-Unterschiede

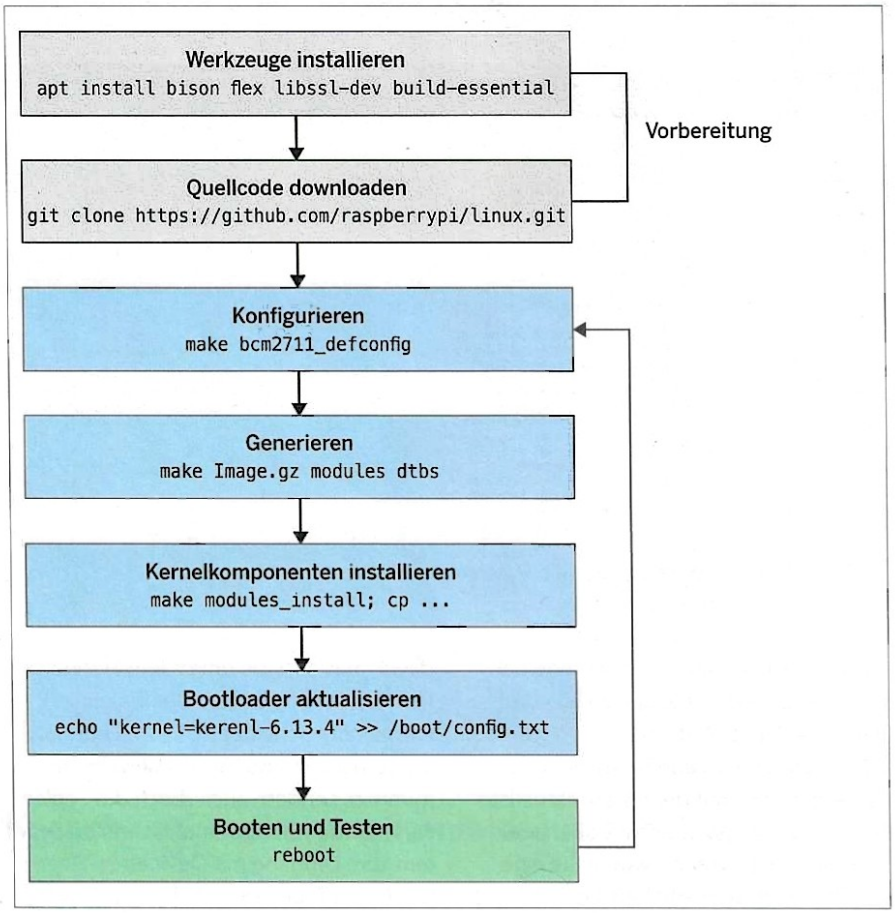
	Mainline-Kernel	Foundation-Kernel
Maintainer	Linus Torvalds	Raspberry Pi Foundation
Quellcode-Repo	git.kernel.org	github.com/raspberrypi/linux
Version	LTS-Versionen	topaktuell
Konfiguration	Standard	optimiert für `bcm2711`, `bcm2712`, `bcmrpi`, `bcm2709`
Devicetree	veraltet	aktuell
DT-Overlay	fehlt	Subsystem vorhanden

Listing 1: Entwicklungswerkzeuge

```
$ sudo apt install git bc bison flex libssl-dev build-essential libncurses-dev libelf-dev
```

Listing 2: Mainline-Kernel-Quellen

```
$ cd /usr/src/
$ sudo mkdir linux
$ sudo chown pi:pi linux
$ git clone --branch=v6.13 git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```



2 **Der Selbstbau-Kernel entsteht in sieben klar definierten Schritten.**

Bei fünf Basisvarianten der Hardware gibt es auch ebenso viele Konfigurationen, die Sie unter Angabe des Pfads via Wget auf Ihren Raspberry Pi laden. Freilich sind die auf diesem Wege akquirierten Dateien nicht ganz so topaktuell wie der Mainline-Kernel selbst, aber erfahrungsgemäß ist dieser Teil unproblematisch.

Listing 3: RasPi-Konfigurationsdateien

```
$ cd /usr/src/linux
### 64-Bit
$ wget -O arch/arm64/configs/bcm2711_defconfig
https://raw.githubusercontent.com/raspberrypi/linux/refs/heads/rpi-
6.6.y/arch/arm64/configs/bcm2711_defconfig
$ wget -O arch/arm64/configs/bcm2712_defconfig
https://raw.githubusercontent.com/raspberrypi/linux/refs/heads/rpi-
6.6.y/arch/arm64/configs/bcm2712_defconfig
### 32-Bit
$ wget -O arch/arm/configs/bcmrpi_defconfig
https://raw.githubusercontent.com/raspberrypi/linux/refs/heads/rpi-
6.6.y/arch/arm/configs/bcmrpi_defconfig
$ wget -O arch/arm/configs/bcm2709_defconfig
https://raw.githubusercontent.com/raspberrypi/linux/refs/heads/rpi-
6.6.y/arch/arm/configs/bcm2709_defconfig
$ wget -O arch/arm/configs/bcm2711_defconfig
https://raw.githubusercontent.com/raspberrypi/linux/refs/heads/rpi-
6.6.y/arch/arm/configs/bcm2711_defconfig
```

Rohkost

Der Download der Konfigurationsdateien gestaltet sich übrigens etwas tricky, wenn man nicht gleich das ganze Archiv auf seine Maschine hieven möchte. Wechseln Sie dazu in das Git-Archiv und klicken Sie sich zur Konfigurationsdatei durch. Die Konfigurationen lagern für die 64-Bit-Versionen im Verzeichnis `arch/arm64/configs/` und für die 32-Bit-Varianten unter `arch/arm/configs/`.

Sobald Sie eine Datei ausgewählt haben, schalten Sie auf *Raw* um und kopieren den in der Adresszeile angezeigten Link 1, den Sie dann mit Wget verknüpfen. Listing 3 zeigt das Ergebnis in Form der auszuführenden Kommandos, mit denen Sie das Mainline-Archiv um sämtliche fünf Default-Konfigurationen der älteren Version 6.6 ergänzen.

Kernel, Kernel

Nun kann es mit der Konfiguration selbst losgehen. Beim Raspberry Pi ist es wichtig, vor der eigentlichen Konfiguration die Umgebungsvariable `KERNEL` zu setzen. Sie legt fest, für welche Architektur der Code generiert werden soll. Für einen Raspberry Pi der Generation 3 oder 4 und einen 64-Bit-Kernel lautet die korrekte Angabe beispielsweise `kernel8`, für einen Raspberry Pi 5 dagegen `kernel_2712`.

Anschließend rufen Sie Make mit dem Namen der zugehörigen Konfigurationsdatei auf (siehe Tabelle Kommandofolgen zur Konfiguration). Eventuelle Änderungen an der Konfiguration nehmen Sie bei Bedarf im Anschluss durch den Aufruf `make menuconfig` vor.

Erste Schritte

Nach der Konfiguration ist vor dem Generieren 2. Tatsächlich gehören zum eigentlichen Kernel noch die Kernel-Module, hinter denen sich vielfach Gerätetreiber verbergen, sowie der Devicetree mit seinen Devicetree-Overlays 3. Alles in allem sind beim Aufruf von Make drei Targets mit anzugeben. Ergänzen Sie

dabei noch die Option `-j6`, nutzt das Generierungswerkzeug alle Prozessoren der Mehrkern-Himbeere optimal aus. Zum Generieren eines 64-Bit-Systems verwenden Sie den Aufruf `make -j6 Image.gz modules dtbs`, für 32 Bit lautet das Kommando `make -j6 zImage modules dtbs`.

Jetzt haben Sie Zeit für ein Kännchen Kaffee oder ein erfrischendes Nickerchen: Der Vorgang dauert auf einem Raspberry Pi 4 gut und gern zwei volle Stunden. Nach der erfolgreichen Generierung gilt es, die Kompilate zu installieren. Das erfolgt in insgesamt vier Schritten: Sie kopieren den Kernel in die Boot-Partition, dazu kommt der Devicetree mit seinen Overlays. Die Module müssen Sie am korrekten Ort in der Root-Partition ablegen, und zu guter Letzt steht noch die Konfiguration des Bootloaders an.

Hier gibt es noch einen Unterschied zwischen einem Mainline- und einem Foundation-Kernel: Ersterer hat nämlich standardmäßig kein eigenes Overlay-System. Das ist jedoch kein großes Manko, denn Sie verwenden in dem Fall einfach die vorhandenen Devicetree-Overlays.

Listing 4: Kernel-Installationsskript (Fortsetzung auf der nächsten Seite)

```
#!/bin/bash
KERNEL="kernel7l"
ARCH="arm"
KERNEL_VERSION="unknown"
BOOTFS="/boot"
ROOTFS="/"

is_kernel_source_dir() {
  REQUIRED_FILES=("Makefile" "Kconfig")
  REQUIRED_DIRS=("arch" "include" "init" "scripts")
  for file in "${REQUIRED_FILES[@]}"; do
    if [ ! -f "$file" ]; then
      echo "kernel source code not found, file
'$file' missing."
      return 1
    fi
  done
  for dir in "${REQUIRED_DIRS[@]}"; do
    if [ ! -d "$dir" ]; then
      echo "kernel source code not found, directory
'$dir' missing."
      return 1
    fi
  done
  return 0
}

if ! is_kernel_source_dir; then
  echo "this script has to be started in the linux
kernel source code dir"
  exit 1
fi

KERNEL_VERSION=$(make kernelversion 2>/dev/null)
if [ -z "$KERNEL_VERSION" ] &&
   [ -f include/config/kernel.release ]; then
  KERNEL_VERSION=$(cat include/config/kernel.
release)
fi

if [ -z "$KERNEL_VERSION" ]; then
  echo "kernel version couldn't be identified"
  exit 1
fi

# Erkennung der Ziel-Architektur beim
Cross-Compiling
if grep -q "CONFIG_64BIT=y" .config 2>/dev/null;
then
  ARCH="arm64"
  KERNEL="kernel8"
else
  echo "Please enter the kernel name (e.g., kernel,
kernel7, kernel7l):"
  read -r KERNEL
  if [ -z "$KERNEL" ]; then
    echo "No kernel name provided, using default
'kernel7l'"
    KERNEL="kernel7l"
  fi
fi

if [ "$(uname -m)" != "arm" ] &&
   [ "$(uname -m)" != "aarch64" ]; then
  echo "cross compile: searching sd-card..."
  if [ "$BOOTFS" == "/boot" ] ||
     [ "$ROOTFS" == "/" ]; then
    SD_CARD=$(lsblk -nr -o NAME | grep -E "^sd|mmc"
| head -n1)
    if [ -n "$SD_CARD" ]; then
      BOOTFS=$(lsblk -nr -o NAME,FSTYPE,MOUNTPOINTS
| awk -v disk="$SD_CARD" '$1 ~ disk"p1" && $2 ==
"vfat" {print $3}')
      ROOTFS=$(lsblk -nr -o NAME,FSTYPE,MOUNTPOINTS
| awk -v disk="$SD_CARD" '$1 ~ disk"p2" && $2 ==
"ext4" {print $3}')
      if [ -n "$BOOTFS" ] && [ -n "$ROOTFS" ]; then
```

Automatismus

Die Installation der Kompilate hängt von der Architektur ab. Listing 4 zeigt ein Bash-Skript, das die lästige Angelegenheit automatisiert. Setzen Sie das Werkzeug aber mit einer Prise Vorsicht ein: Der Autor konnte noch nicht alle möglichen Fälle durchprobieren.

Sie rufen das Skript aus dem Quellcodeverzeichnis heraus auf. Es versucht, selbstständig die notwendigen Konfigurationsparameter zu evaluieren. Dazu gehören neben der Architektur (`arm64`, `arm`) die Art des RasPi-Kernels (`kernel8`, `kernel7`, `kernel7l`, `kernel`) sowie die genaue Version des Linux-Kernels (zum Beispiel `6.13.4`). Darüber hinaus evaluiert das Skript noch das Installationsziel, sodass Sie es auch beim Cross-Compiling einsetzen können.

Da sich die Art des Raspberry-Pi-Kernels nur bei der 64-Bit-Spielart (`kernel8`) leicht bestimmen lässt, erfragt das Skript die entsprechende Information im Fall eines 32-Bit-Kernels direkt beim Benutzer. Es benötigt die Parameter, da die Installation je nach Architektur, eingesetzter Hardwarevariante und Kernel-Version unterschiedlich abläuft.

Zunächst kopiert das Skript den Kernel und den Devicetree auf die Boot-Partition, die der Raspberry Pi in das Verzeichnis `/boot/firmware/` einhängt. Derzeit ist das Kopieren des Devicetrees im Skript allerdings auskommentiert und

Listing 4: Kernel-Installationsskript (Fortsetzung von S. 75)

```
        echo "Found SD card: Boot=$BOOTFS,
Root=$ROOTFS"
      else
        echo "No valid SD card partitions found!"
        exit 1
      fi
    else
      echo "No valid SD card found!"
      exit 1
    fi
  fi
fi

echo "ARCH           = $ARCH"
echo "KERNEL         = $KERNEL"
echo "KERNEL_VERSION = $KERNEL_VERSION"
echo "BOOTFS         = $BOOTFS"
echo "ROOTFS         = $ROOTFS"

echo "copying kernel files..."
if [ "$ARCH" == "arm64" ]; then
  make -j$(nproc) ARCH=arm64 CROSS_COMPILE=aarch64-
linux-gnu- INSTALL_MOD_PATH=$ROOTFS modules_install
  cp arch/arm64/boot/Image $BOOTFS/$KERNEL-$KERNEL_
VERSION.img
  #
  # DTBs of the mainline kernel make problems,
uncomment,
  # if you want to use them...
  #
  # [ -d arch/arm64/boot/dts/broadcom ] &&
cp arch/arm64/boot/dts/broadcom/*.dtb $BOOTFS/
  # [ -d arch/arm64/boot/dts/overlays ] &&
cp arch/arm64/boot/dts/overlays/*.dtb* $BOOTFS/
overlays/
  sed -i '/^kernel=/s/^/#/' $BOOTFS/config.txt
  echo "kernel=$KERNEL-$KERNEL_VERSION.img" >>
$BOOTFS/config.txt
  sed -i 's/^arm_64bit=0/arm_64bit=1/' $BOOTFS/
config.txt
elif [ "$ARCH" == "arm" ]; then
  make -j$(nproc) ARCH=arm CROSS_COMPILE=arm-linux-
gnueabihf- INSTALL_MOD_PATH=$ROOTFS modules_install
  cp arch/arm/boot/zImage $BOOTFS/$KERNEL-$KERNEL_
VERSION.img
  #
  # DTBs of the mainline kernel make problems,
  # uncomment if you want to use them...
  #
  # [ -d arch/arm/boot/dts/overlays ] &&
cp arch/arm/boot/dts/overlays/*.dtb* $BOOTFS/
overlays/
  # if [ "$(printf '%s\n' "6.4" "$KERNEL_VERSION" |
sort -V | head -n1)" == "6.4" ]; then
  #     [ -d arch/arm/boot/dts/broadcom ] &&
cp arch/arm/boot/dts/broadcom/*.dtb $BOOTFS/
  # else
  #     cp arch/arm/boot/dts/*.dtb $BOOTFS/
  # fi
  sed -i '/^kernel=/s/^/#/' $BOOTFS/config.txt
  echo "kernel=$KERNEL-$KERNEL_VERSION.img" >>
$BOOTFS/config.txt
  sed -i 's/^arm_64bit=1/arm_64bit=0/' $BOOTFS/
config.txt
else
  echo "unsupported architecture: $ARCH"
  exit 1
fi
sync

echo "kernel installation complete."
```

erfolgt damit nicht. Tatsächlich weigert sich der Raspberry Pi, mit einem Devicetree aus dem aktuellen Linux-Kernel zu booten. Wollen Sie doch den Mainline-Devicetree einsetzen, erstellen Sie vor dem Kopieren am besten ein Backup.

Versionsprobleme

Bei der 32-Bit-Variante prüft das Skript zusätzlich, ob der generierte Kernel die Versionsnummer 6.4 oder älter trägt. Die beim Foundation-Kernel generierten Overlays kommen in das Verzeichnis `/boot/firmware/overlays/`. Beim Mainline-Kernel spart der Code diesen Schritt aus, in der Annahme, dass die bereits installierten Overlays passen.

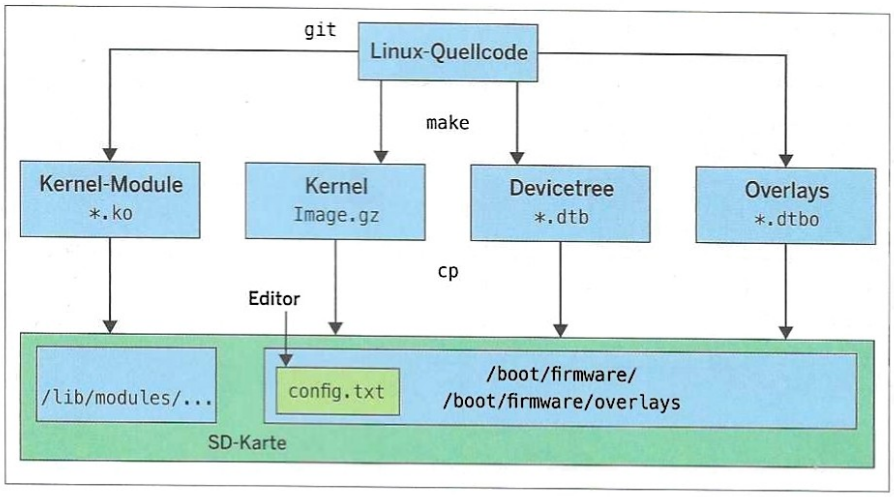
Bleibt zum Schluss noch die Konfiguration des Bootloaders, die auf der Root-Partition in der Datei `config.txt` steckt. Findet der Bootloader in dieser Datei das Schlüsselwort `kernel=`, dann betrachtet er den dahinter stehenden String als Namen des zu bootenden Kernels. Per `echo` generiert das Skript einen solchen Eintrag und platziert ihn am Ende der `config.txt`. Außerdem stellt es mithilfe von Sed sicher, dass die Variable `arm_64bit` bei einem 64-Bit-System auf 1 steht und ansonsten auf `0`.

Der Bootloader ist damit konfiguriert, einem Reboot steht nichts mehr im Wege. Danach loggen Sie sich ein und überprüfen, ob der gerade generierte Kernel aktiv ist, indem Sie im Terminal das Kommando `uname -a` eingeben und die Ausgabe studieren. Sie sollten nun die neue Versionsnummer samt Generierungszeitpunkt sehen. Chapeau!

Strom geben

Ein Raspberry Pi gehört nicht eben zu den schnellsten Rechnern, der Linux-Kernel aber wächst und gedeiht. Damit steigt zwangsläufig die Zeit für die Generierung. Entlastung verspricht hier eine Cross-Entwicklung: Man nehme einen PC, generiere dort Kernel, Devicetree und eventuell die Overlays und installiere das Ergebnis auf dem Kleinrechner.

Für ein schnelleres Generieren benötigen Sie neben den eingangs bereits erwähnten Paketen auf dem Entwicklungsrechner noch den Cross-Compiler. Auf einem Debian-basierten System lassen sich die erforderlichen Pakete erfreulich einfach einrichten, wobei das Kommando aus Listing 5 den Cross-Compiler sowohl für 64- als auch für 32-Bit-Systeme lädt.



3 Zum Linux-Kern gehört mehr als nur der eigentliche Kernel.

Den Quellcode holen Sie wie beschrieben am besten per Git ab, die zugehörigen Default-Konfigurationen per Wget (siehe Listing 2). Bei der Konfiguration und der Generierung selbst gibt es allerdings Änderungen: Sie müssen dem Kernel-Build-System mitteilen, dass Sie Code für eine Plattform generieren möchten, die auf einem ARM-Prozessor läuft. Des Weiteren muss klar sein, wo auf dem Rechner sich die Entwicklungswerkzeuge befinden beziehungsweise wie genau die Tools heißen.

Konventionen

Damit sich unterschiedliche Compiler, Linker, Archiver und die weiteren Werkzeuge nicht in die Quere kommen, hat sich eine Namenskonvention herausgebildet. Sie setzt den Namen aus der Architektur und dem Betriebssystem des Ziels zusammen, hinzu kommen die verwendete Bibliothek und der Name des Werkzeugs selbst. Zu guter Letzt folgt noch eine Versionsnummer 4.

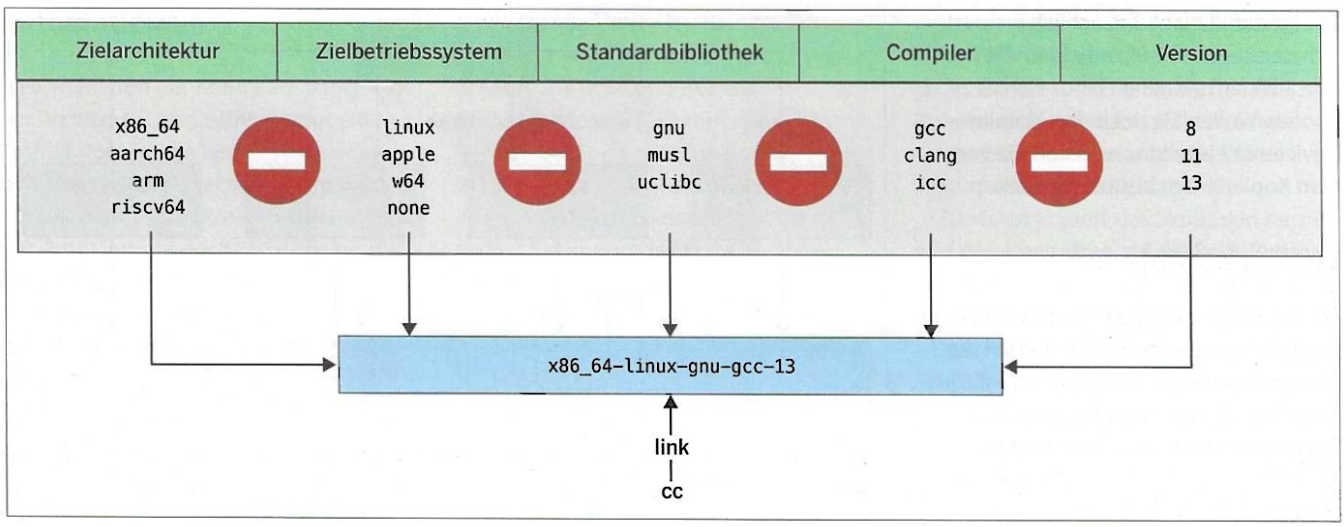
Der tatsächliche Name des C-Compilers, den Sie typischerweise mit dem Kommando `cc` oder `gcc` aufrufen, lautet auf einem PC `x86_64-linux-gnu-gcc-13`, auf dem Raspberry Pi dagegen `arm-`

Kommandofolgen zur Konfiguration

Modell	Architektur	Kommandofolge
Raspberry Pi 1, Raspberry Pi Zero	32 Bit	`KERNEL=kernel` `make bcmrpi_defconfig`
Raspberry Pi 2, Raspberry Pi 3	32 Bit	`KERNEL=kernel7` `make bcm2709_defconfig`
Raspberry Pi 4	32 Bit	`KERNEL=kernel7l` `make bcm2711_defconfig`
Raspberry Pi 3, Raspberry Pi 4	64-Bit	`KERNEL=kernel8` `make bcm2711_defconfig`
Raspberry Pi 5	64 Bit	`KERNEL=kernel_2712` `make bcm2712_defconfig`

Listing 5: Cross-Compiler

```
$ apt install crossbuild-essential-arm64 crossbuild-essential-armhf
```



4 **Dank eines ausgeklügelten eindeutigen Namensschemas koexistieren problemlos mehrere Compiler auf einer Maschine.**

`linux-gnueabihf-gcc-12`. Auf diesen eigentlichen Namen verweisen jetzt eine Reihe von symbolischen Links, unter anderem einer, der die Versionsnummer weglässt. Bei `cc` handelt es sich schließlich um einen symbolischen Link auf den Compiler für die eigene Architektur.

Umgebung bauen

Für das Cross-Compiling geben Sie dem Kernel-Build-System über zwei Umgebungsvariablen den Namensvorsatz (für 64-Bit-ARM beispielsweise `CROSS_COMPILE="arm-linux-gnueabihf-"`) und die Architektur bekannt (hier `ARCH=arm`).

Die Tabelle Raspberry Pi: Kernel-Konfiguration bei der Cross-Generierung zeigt die Kommandofolge, die zur Konfiguration der jeweiligen Raspberry-Pi-Variante (Modell und Architektur) Anwendung findet, zusammen mit dem Aufruf zur Generierung der drei Targets. Hier gilt es wieder die Variablen `KERNEL`, `ARCH` und `CROSS_COMPILE` zu setzen, und schon baut der Host-Rechner den gewünschten Linux-Kernel samt Komponenten.

Es bleibt als letzter Schritt noch der Transfer des Kompilats auf den Zielrechner, den man beim Cross-Compiling als Target bezeichnet. Dazu stecken Sie die SD-Karte des RasPi in das Hostsystem.

Auch hier können Sie wieder das Skript aus Listing 4 anwenden, das versucht, die Pfade zu den Verzeichnissen auf der Boot- und der Root-Partition automatisiert zu evaluieren.

Nach dem Kopieren der Komponenten und der Konfiguration des Bootloaders entnehmen Sie dem Hostsystem die Karte, stecken sie in den Raspberry Pi und erfreuen sich an dem neuen Kernel. *(jlu)*

Weitere Infos und interessante Links
www.lm-online.de/qr/51440

Raspberry Pi: Kernel-Konfiguration bei der Cross-Generierung

Modell	Architektur	Kommandofolge

Raspberry Pi 1, Raspberry Pi Zero	32 Bit	`KERNEL=kernel`
		`make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcmrpi_defconfig`
		`make -j6 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs`

Raspberry Pi 2, Raspberry Pi 3	32 Bit	`KERNEL=kernel7`
		`make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709_defconfig`
		`make -j6 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs`

Raspberry Pi 4	32 Bit	`KERNEL=kernel7l`
		`make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2711_defconfig`
		`make -j6 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs`

Raspberry Pi 3, Raspberry Pi 4	64 Bit	`KERNEL=kernel8`
		`make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2711_defconfig`
		`make -j6 ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image modules dtbs`

Raspberry Pi 5	64 Bit	`KERNEL=kernel_2712`
		`make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2712_defconfig`
		`make -j6 ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image modules dtbs`