Einleitung

Bis vor einiger Zeit bestand der übliche Weg zum Aufbau von Geräte-/Maschinensteuerungen darin, dass das Entwicklungsteam eine CPU auswählte, die am einfachsten an die bestehenden Hardwarekomponenten angepasst werden konnte. Diese CPU wurde in Assembler oder in einer Kombination von Assembler und Hochsprache (häufig C) programmiert.
Bei einem späteren Redesign des Gerätes wurde möglichst versucht, die einmal ausgewählte CPU beizubehalten oder durch eine höherwertige Variante aus der gleichen Typfamilie zu ersetzen, um Änderungsarbeiten an den (hardwarenahen) Softwarekomponenten zu vermeiden.
Eine solche Vorgehensweise ist heute nicht mehr angebracht.

Die vom Markt geforderten Geräteleistungen schlagen sich in immer kürzeren Produktzyklen nieder. Ähnliche Produkte von Mitbewerbern zwingen das Entwicklungsteam, die Funktionalität des Gerätes zu steigern.
Es ist deshalb ökonomischer, schon beim Entwurf einer aktuellen Gerätegeneration auch an den Bedarf einer Nachfolgergeneration zu denken:

- Geräte müssen auch aus der Ferne bedien- und wartbar sein. Eine Anbindung an produktionsinterne Bussysteme und/oder Ethernet kommt dieser Forderung nach.

  Fernwartbarkeit

- Dedizierte Schalter und Taster sollen nur noch in eingegrenztem Umfang eingesetzt werden. Multifunktionstasten werden als ergonomischer empfunden, reduzieren die Frontplattengröße eines Gerätes und damit die Übersicht und sind kostengünstiger. Ähnliche Argumente sprechen für die Verwendung von berührungsempfindlichen Bildschirmen (Touchscreens).

  Ergonomie

- Die Bediener des Gerätes oder der Maschine erwarten einen „Bedienungskomfort“, der sich an dem orientiert, was der PC auf dem Schreibtisch schon lange bietet: Grafische Dialogsysteme, auch oft in der Art und Weise wie sie zum Beispiel Webbrowser bieten.

  Erwartungshaltung der Bediener

- Maschinenparameter und Daten der laufenden Produktion werden von übergeordneten Systemen (Enterprise Resource Planning) vorgegeben oder abgefragt. Die Gerätesteuerung muss entsprechende Datenschnittstellen zur Verfügung stellen.

  Datenschnittstellen

Diese Anforderungen an eine moderne Gerätesteuerung lassen sich nur dann mit einem bezahlbaren Aufwand erfüllen, wenn standardisierte Software- und

Aufwand

Hardwarekomponenten eingesetzt werden, die von dem Entwicklungsteam mit möglichst geringem Aufwand an die gerätetypischen Randbedingungen angepasst werden können.

Lösungsansatz

Wo kommen diese standardisierten Software- und Hardwarekomponenten her?
Einen nicht geringen Teil der geforderten Funktionalität stellen zum Beispiel die auf fast jedem Schreibtisch stehenden Computersysteme schon von Hause aus zur Verfügung. Es liegt deshalb nahe, die dort eingesetzten Betriebssysteme und möglichst viele der dazugehörigen Hardwarekomponenten auch in der Geräte-/Maschinensteuerung einzusetzen. Dem Entwicklungsteam fällt dann die Aufgabe zu, diese Komponenten zu nehmen und für den Bedarf der Gerätesteuerung anzupassen. Die durch diese Integration („Einbettung“) entstehenden Systeme nennt man nicht nur im englischen Sprachraum „Embedded Systems“.
Nach dem Durcharbeiten dieser Kurseinheit

- sind Sie in der Lage, Ihre Gerätesteuerung so auszulegen, dass auch der Bedarf zukünftiger Gerätegenerationen mit möglichst geringem Aufwand integriert werden kann.
- wissen Sie, welche grundlegenden Funktionen der Linuxkernel bietet.
- können Sie Ihre Applikation so implementieren, dass diese nach dem Booten des Linuxkernels in Ihrem Gerät automatisch ausgeführt werden kann.
- sind Sie in der Lage, einfache Probleme schnell mit einem Skript zu lösen.
- sind Sie in der Lage, Fehler Ihres Embedded Systems mit einfachen Mitteln detektieren und beheben zu können.
- können Sie aus den üblichen Dateisystemen das für Sie optimale auswählen.
- kennen Sie die Bedingungen beim Einsatz von üblichen Schnittstellen zur Kommunikation Ihres Embedded Systems zu anderen Geräten.
- überblicken Sie grob die Randbedingungen, die Sie beachten müssen, wenn Sie aus dem Pool freier Software einzelne Komponenten für den Einsatz in Ihrem Unternehmen integrieren wollen.

Inhaltsverzeichnis

1 Embedded Systems - warum Linux?

Linux bietet sich in mehrfacher Hinsicht als Kandidat für den Einsatz in Gerätesteuerungen an:

1. Linux ist ein weitverbreitetes Betriebssystem: **Linux ist weitverbreitet**
   - Es gibt marktreife Distributionen.
   - Aktuelle und vergangene Versionen sind frei verfügbar.
   - Zahlreiche Teams pflegen das komplette System.
   - Eine kaum zu überschauende Anzahl von Entwicklungskomponenten (Compiler, Tools, Kommandozeilenprogramme) steht zur freien Verfügung. **Compiler, Tools**

2. Bei der Weitergabe von Geräten mit Linux fallen keine Lizenzgebühren an. **Keine Lizenzgebühren**

3. Linux ist skalierbar: Linux läuft auf einem MP4-Player, in einem Cluster oder auch als hoch verfügbarer Webserver für zum Beispiel Versandhäuser. **Linux ist skalierbar**

4. Es stehen Treiber für sehr viele Hardwarekomponenten zur Verfügung. Sollten gerätespezifische Anpassungen von Treibern notwendig werden, so stehen unter Linux die entsprechenden Quellen frei zur Verfügung. **Treiberquellen sind verfügbar**

5. Eine Entwicklung unter Linux ist auch zukunftssicher. Das Entwicklerteam kann während der Entwicklung einen Abzug des Linuxkernels (snapshot) anfertigen und für einen zukünftigen Bedarf sichern. Etwaige geringfügige Änderungen wären so möglich, ohne auf einen komplett neuen Kernel zurückgreifen zu müssen. **Zukunftssicher**

6. Die Zahl der CPU-Familien (Plattformen), die von Linux unterstützt werden, nimmt sehr schnell zu. Für folgende CPUs sieht der offizielle Kernel unter `ftp.kernel.org` Unterstützung vor: **Sehr viele CPUs**

alpha	arm	avr32	blackfin	cris
frv	h8300	i386	ia64	m32r
m68k	m68knommu	mips	parisc	powerpc
ppc	s390	sh	sh64	sparc
sparc64	um	v850	x86_64	xtensa

7. Der Miniaturisierungs- und Kostendruck, der auf dem Embedded System-Entwicklerteam lastet, ist auch bei den Halbleiter- und Chipherstellern

Systems on Chip

angekommen. In Embedded Systems kommen nicht nur die traditionellen CPU-Bauformen zum Einsatz, sondern auch sogenannte „Systems on Chip“ (SOC). Hierbei handelt es sich um die Integration einer CPU und wichtiger Peripheriebausteine in einen einzigen Chip: zum Beispiel eine CPU mit Interfaces für LCD/TFT-Bildschirme, Ethernet-, USB-, RS-232C-, I^2C-, SPI- und PS2-Schnittstellen. Ein amerikanischer Hersteller hat Spezialhardware zur Krypto-Beschleunigung direkt in einen SOC integriert, der vorwiegend in WLAN-Routern zum Einsatz kommt.

2 Grober Aufbau eines Embedded Systems unter Linux

Neben unseren eigenen Softwarekomponenten ruht die Funktionalität unseres kompletten Systems auf dem Linuxkernel. Der grobe Aufbau und die Funktionsweise eines Linuxkernels werden dargestellt. Soweit möglich wird hier auch auf Wissen zugegriffen, das die vorausgegangenen Kurseinheiten vermittelt haben.

Lernziele

Nach dem Durcharbeiten dieses Kapitels

- überblicken Sie die Struktur der Quellprogramme des Linuxkernels.
- können Sie einen Linuxkernel grob konfigurieren und testen.
- kennen Sie die Hauptphasen, die der Linuxkernel beim Starten durchläuft.
- kennen Sie Möglichkeiten, die Ihnen der Linuxkernel für die Initialisierung Ihrer eigenen Softwarekomponenten bietet.

2.1 User- und Kernel-Space

Der Hauptspeicher eines Systems wird in zwei Bereiche aufgeteilt [VOGT]:

- den User-Space, den die Applikationen der Anwender nutzen, und den **User-Space**
- Kernel-Space, der für Kernelzwecke reserviert ist. **Kernel-Space**

Diese Aufteilung dient primär der Stabilität und Sicherheit des Gesamtsystems und verhindert zum Beispiel, dass die Applikation eines Anwenders auf Kernelbereiche zugreifen und diese eventuell zum Nachteil einer anderen Applikation verändern kann. Zugriffe auf die Hardwarekomponenten sind grundsätzlich dem Kernel vorbehalten. **Applikation**

User- und Kernel-Space existieren gleichzeitig im Adressraum einer CPU. Nur so ist es möglich, dass aus dem User-Space eine Servicefunktion des Kernels aufgerufen werden kann.

Unter Linux (32 Bit) ist der Adressraum so aufgeteilt, dass

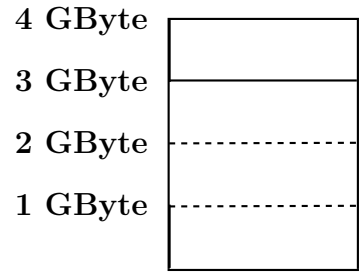

Abbildung 1: Aufteilung des Adressraums unter Linux (32 Bit)

- die untersten 3 GByte zum User-Space und
- das oberste GByte zum Kernel-Space gehören (Abbildung 1).

MMU

Wichtig für uns ist, dass, auch wenn viele Applikationen gleichzeitig aktiv sind, der RAM-Speicher, der tatsächlich für den Kernel zur Verfügung steht, nur einfach belegt ist. Die Memory Management Unit (MMU) sorgt dafür, dass der RAM-Bereich mit dem Kernel in den Adressbereich jeder Applikation eingeblendet wird. Zusätzlich sorgt die MMU dafür, dass keine unberechtigten Zugriffe auf den Kernel-Space stattfinden.

2.2 Kernelkonfiguration

Die Quellen des offiziellen Linuxkernels finden Sie unter `ftp.kernel.org`. Das Bezeichnungsschema der einzelnen Kernelversionen ist so aufgebaut:

- Auf die Versionsnummer folgt
- der Patchlevel. Danach kommt der
- Sublevel, dem die
- Extraversionsnummer folgt. Manchmal wird noch ein
- Name angehängt.

Beispiel

Beispiel:
Ein Kernel 2.6.38.1 wird zur Generation der 2.6-Kernel gezählt. Der Sublevel 38 und die Extraversionsnummer 1 beschreiben, wie stark sich der Kernel innerhalb der Generation weiterentwickelt hat.

Bis vor einiger Zeit galt darüber hinaus, dass der Patchlevel etwas über den Entwicklungsstand aussagte:

- Kernel 2.4.x konnten direkt für Produktionszwecke eingesetzt werden.
- Kernel 2.5.x waren im Experimentierstadium.
- Kernel 2.6.x war wieder für Produktionszwecke angelegt.

Allerdings beschloss das Entwicklungsteam um Linus Torvalds während der Entwicklungszeit 2.6.x die Trennung in Experimentier- und Produktionsstand aufzugeben.
In der Abbildung 2 auf der Seite 5 ist grob der Umfang der Quellen des Kernels `2.6.38.1` dargestellt:

- Fast die Hälfte (ca. 49 %) des Speicherplatzes wird von den Quellen der Systemtreiber eingenommen.
- Der architekturspezische Teil des Kernels umfasst ca. 24 %.

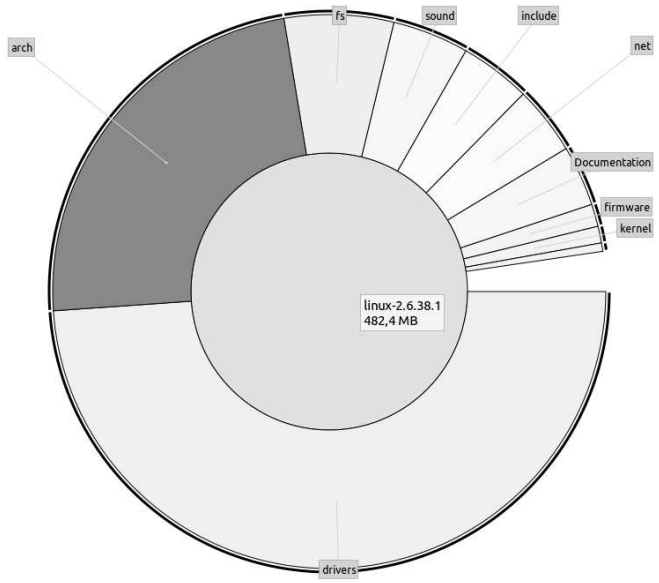

Abbildung 2: Strukturierung der Quelldateien des Linuxkernels `2.6.38.1`

Ein Teil des Linuxkernels dient zur Anpassung der jeweiligen CPU an den Bedarf des Kernels. Das Verzeichnis `arch` des Linuxkernels beinhaltet die

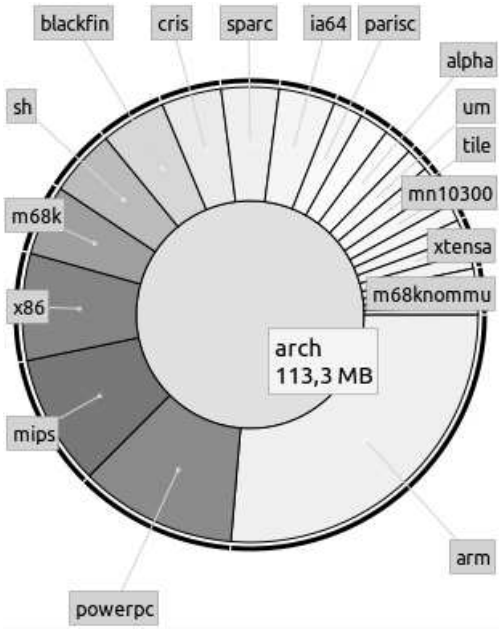

Abbildung 3: Im Verzeichnis `arch` ist der Code zu finden, der zur Anpassung der zahlreichen CPUs an den Kernel dient

entsprechenden Quellen. In der Abbildung 3 auf der Seite 6 ist dargestellt, wie sich der architekturspezifische Quellcode auf die einzelnen CPUs verteilt.

Offizieller Kernel

Neben dem offiziellen Linuxkernel unter `ftp.kernel.org` gibt es noch weitere. Diese Kernel sind ursprünglich aus dem offiziellen Kernel hervorgegangen und von einzelnen Teams an ihren Bedarf angepasst worden. Es ist nicht immer sichergestellt, dass die Änderungen bzw. Anpassungen dieser (inoffiziellen) Kernel auch wieder in den offiziellen Kernel zurückfließen. Auch die umgekehrte Richtung wird nicht automatisch eingeschlagen: Werden beispielsweise sicherheitsrelevante Änderungen im offiziellen Kernel durchgeführt, so liegt es auch in dem Ermessen der Entwicklungsteams, diese in die inoffiziellen Kernel zu übernehmen.

Inoffizieller Kernel

BSP

Es gibt eine Reihe von Firmen, die als Dienstleistung architekturspezifische Kernel anbieten. Solche Kernel sind häufig ein Bestandteil eines Paketes (board support package `BSP`) und werden als Option beim Kauf von Mainboards oder kompletten Rechnern angeboten. Eine solche Dienstleistung kann durchaus für Sie interessant sein.

Tipp

Tipp:
Sie sollten bei Ihren Lieferanten prüfen, ob und wie oft sicherheitsrelevante Änderungen des offiziellen Linuxkernels eingepflegt werden.

Sie können sich einen Linuxkernel herunterladen und an Ihren Bedarf anpassen. Dafür gibt es Hilfsmittel. Eine solche Konfigurierung und Generierung eines Linuxkernels wird schnell an einem Beispiel klar.
Beispiel: **Beispiel**

1. Quellen des Kernels bereitstellen: **Quellen**

```
1 hbo@PB:~$ bunzip2 linux-2.6.38.1.tar.bz2
2 hbo@PB:~$ ls -ls
3 insgesamt 430184
4 ... -rw-r--r-- 1 hbo hbo 440504320 ... linux-2.6.38.1.tar
5 hbo@PB:~$ tar -xf linux-2.6.38.1.tar
6 hbo@PB:~$ ls -ls
7 insgesamt 430188
8 ... drwxr-xr-x 23 hbo hbo      4096 ... linux-2.6.38.1
9 ... 1 hbo hbo 440504320 2012-05-13 13:36 linux-2.6.38.1.tar
```

   Die Quellen des Linuxkernels sind komprimiert. In Zeile 1 wird der Kernel entkomprimiert. Das Archiv `linux-2.6.38.1.tar` mit den Quellen wird in Zeile 5 entpackt. Es entsteht ein Verzeichnis `linux-2.6.38.1` (Zeile 8).

2. Kernel konfigurieren: **Konfigurieren**

   Mit dem Kommando `make menuconfig` leiten Sie die Konfigurierung ein.

   In der Abbildung 4 auf der Seite 8 ist der Dialog zur Konfigurierung eines Kernels dargestellt. Der Dialog ist hierarchisch strukturiert; mit den Pfeiltasten navigieren Sie sich durch die einzelnen Dialogseiten und wählen einzelne Optionen aus.

   Die Konfigurierung kann sich langwierig gestalten. Es ist ratsam, hier in mehreren Schritten vorzugehen und sich so iterativ dem gewünschten Ziel zu nähern. Es ist nicht selten, dass die möglichen Optionen nicht so deutlich beschrieben sind, wie es notwendig wäre. Sie entscheiden sich dann vielleicht für eine Variante, die in Widerspruch zu einer an einer anderen Dialogseite getroffenen Entscheidung steht. (Dieser Widerspruch löst später unter Umständen Compilier- oder Linkfehler aus.)

   Schrittweises Vorgehen

   Fehler

   Tipp: **Tipp**

   Mit dem Kommando `make defconfig` können Sie einen Kernel erzeugen, der komplett und widerspruchsfrei erzeugt werden kann. Das Linuxteam verwendet hierbei einen vorgefertigten Satz von Optionen, die

Ihnen einen Kernel definieren, der als Ausgangsbasis für Ihren Start dienen kann.

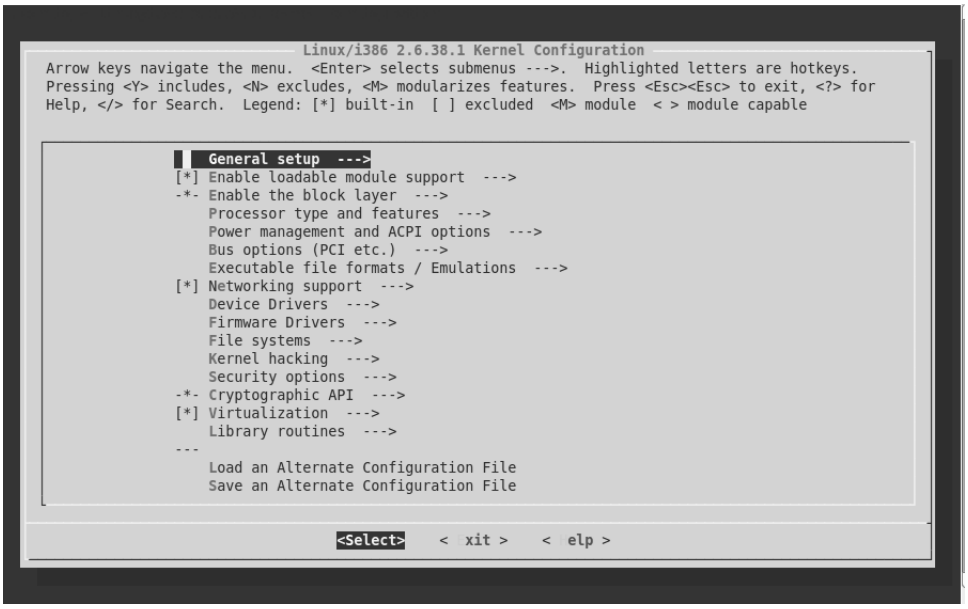

Abbildung 4: Dialog zur Konfigurierung des Kernels

Hinweis

Hinweis:

Alle Antworten, die Sie beim Konfigurieren des Kernels eingeben, werden in einer Datei `.config` gespeichert. Ein Ausschnitt aus dieser Datei sieht so aus:

```
1  #
2  # Automatically generated make config: don't edit
3  # Linux/i386 2.6.38.1 Kernel Configuration
4  # Sat May 19 18:49:13 2012
5  #
6  # CONFIG_64BIT is not set
7  CONFIG_X86_32=y
8  # CONFIG_X86_64 is not set
9  CONFIG_X86=y
10 CONFIG_INSTRUCTION_DECODER=y
11 CONFIG_OUTPUT_FORMAT="elf32-i386"
12 CONFIG_ARCH_DEFCONFIG="arch/x86/configs/i386_defconfig"
13 CONFIG_GENERIC_CMOS_UPDATE=y
14 CONFIG_CLOCKSOURCE_WATCHDOG=y
15 ...
```

Legen Sie sich Sicherheitskopien von `.config` an. Sollten Sie bei späteren Sitzungen auf Probleme beim Konfigurieren stoßen, könnten Sie mit einer Sicherheitskopie einen gesicherten Neuanfang versuchen.

3. Kernel compilieren und erzeugen: **Compilieren**

Mit der Eingabe `make` starten Sie das Zusammenbauen des Kernels. Auf Ihrem Terminal erscheinen Ausgaben wie:

```
1    CHK     include/linux/version.h
2    CHK     include/generated/utsrelease.h
3    CALL    scripts/checksyscalls.sh
4    CHK     include/generated/compile.h
5    VDSOSYM arch/x86/vdso/vdso32-int80-syms.lds
6    VDSOSYM arch/x86/vdso/vdso32-sysenter-syms.lds
7    VDSOSYM arch/x86/vdso/vdso32-syms.lds
8    LD      arch/x86/vdso/built-in.o
9    LD      arch/x86/built-in.o
10   ...
11   CC      arch/x86/boot/version.o
12   LD      arch/x86/boot/setup.elf
13   OBJCOPY arch/x86/boot/setup.bin
14   OBJCOPY arch/x86/boot/vmlinux.bin
15   BUILD   arch/x86/boot/bzImage
16 Root device is (8, 5)
17 Setup is 12588 bytes (padded to 12800 bytes).
18 System is 4053 kB
19 CRC b82b3400
20 Kernel: arch/x86/boot/bzImage is ready  (#2)
21   Building modules, stage 2.
22   MODPOST 3 modules
```

Je nach der Geschwindigkeit Ihres Rechners kann das Zusammenbauen des Kernels schon einige Zeit kosten, da viele Hundert Dateien übersetzt und gelinkt werden müssen.

Das Ergebnis ist in der Zeile 15 abzulesen: eine Datei mit Namen `bzImage`.

(Üblicherweise wird nicht nur der Kernel verwahrt, sondern auch eine Datei mit wichtigen Adressinformationen:

```
1 hbo@PB:~$ ls -ls *.map
2 1892 -rw-r--r-- 1 hbo hbo 1933539 ...  System.map
3 hbo@PB:~$ ls -ls arch/x86/boot/bzImage
4 4068 -rw-r--r-- 1 hbo hbo 4162544 ... arch/x86/boot/bzImage
```

Wenn Sie die beiden Dateien in Ihr `/boot`-Verzeichnis kopieren würden, sollten Sie die Namen `System.map-2.6.38-1` bzw. `vmlinuz-2.6.38-1`

verwenden. Üblich ist es, auch die `.config`-Datei dort als `config-2.6.38-1` zu deponieren.)

Test

4. Kernel testen:

   Auch wenn Sie einige Hintergründe erst in späteren Kapiteln erfahren, soll hier unser gerade erzeugter Kernel getestet werden. Wir starten ihn in dem Emulator QEMU (Kapitel 3.2 auf der Seite 31) und erhalten folgende Ausgaben auf dem Terminal:

```
1  [0.000000] Initializing cgroup subsys cpuset
2  [0.000000] Initializing cgroup subsys cpu
3  [0.000000] Linux version 2.6.38.1 (hbo@PB) (gcc version 4.5.2
   ..
4  [0.000000] BIOS-provided physical RAM map:
5  [0.000000]  BIOS-e820: 0000000000000000 - 000000000009f400 (usable)
6  [0.000000]  BIOS-e820: 000000000009f400 - 00000000000a0000 (reserved)
7  [0.000000]  BIOS-e820: 00000000000f0000 - 0000000000100000 (reserved)
8  [0.000000]  BIOS-e820: 0000000000100000 - 0000000017ffd000 (usable)
9  [0.000000]  BIOS-e820: 0000000017ffd000 - 0000000018000000 (reserved)
10 [0.000000]  BIOS-e820: 00000000fffc0000 - 0000000100000000 (reserved)
11 [0.000000] Notice: NX (Execute Disable) protection missing in CPU!
12 [0.000000] DMI 2.4 present.
13 [0.000000] last_pfn = 0x17ffd max_arch_pfn = 0x100000
14 [0.000000] found SMP MP-table at [c00fd7a0] fd7a0
15 [0.000000] init_memory_mapping: 0000000000000000-0000000017ffd000
16 [0.000000] RAMDISK: 17faa000 - 17ff0000
17 ...
18 [3.004928] registered taskstats version 1
19 [3.008895]   Magic number: 12:254:593
20 [3.020431] Freeing unused kernel memory: 404k freed
21 [3.054395] Write protecting the kernel text: 4660k
22 [3.055308] Write protecting the kernel read-only data: 1864k
23 Hello world!
```

Test ist erfolgreich

In der Zeile 24 wird ein beim Start verwendetes Testprogramm gestartet. Damit kann festgestellt werden, dass der von uns generierte Kernel funktioniert.

Die anderen Meldungen, die vom Kernel beim Start ausgegeben werden, zeigen an, welche Einstellungen gewählt wurden und welche Hard- und Softwarekomponenten er gefunden hat. In den Zeilen 5 bis 10 werden beispielsweise die vom BIOS ermittelten RAM-Speicherbereiche ausgegeben.

2.3 Kernelinitialisierung

Die Kernelinitialisierung beginnt

- beim Einschalten der CPU und endet
- mit dem Starten des ersten Prozesses im User-Space.

Gerade die ersten Phasen der Kernelinitialisierung sind stark abhängig von dem eingesetzten CPU-Typ. Im Folgenden wird angenommen, dass eine x86-CPU mit 32 Bit eingesetzt wird.
Die einzelnen Phasen der Initialisierung lassen sich grob in zwei Abschnitte aufteilen (Abbildung 5 auf Seite 11):

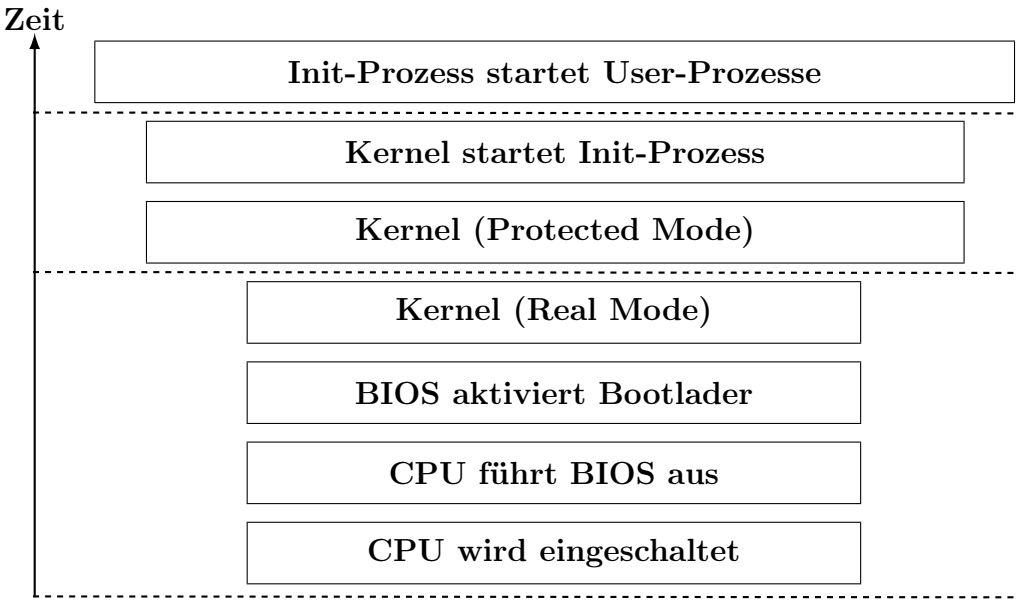

Abbildung 5: Die einzelnen Phasen der Kernelinitialisierung

Abschnitt 1:

1. Nach dem Einschalten startet die CPU auf einer von ihrem Hersteller fest definierten Adresse. Diese Adresse ist die Startadresse des BIOS-Programms. Das BIOS wurde vom Hersteller des Mainboards an die darauf befindlichen Hardwarekomponenten angepasst und bietet nach außen eine herstellerunabhängige Schnittstelle.

2. Das BIOS sucht auf den bekannten Datenträgern des Systems nach einem Bootlader und startet diesen.

3. Der Bootlader lädt die Datei mit dem Kernel vom Datenträger in das RAM und bringt den Kernel zur Ausführung. In einem Suchlauf stellt der Kernel fest, wie viel RAM-Speicher insgesamt zur Verfügung steht.

RAM-Speicher ermitteln

Einige Geräte initialisieren

Danach werden einige Geräte oder Hardwarekomponenten des Mainboards initialisiert. (Die Videokarte muss beispielsweise bereit sein, um Ausgaben auf dem Monitor tätigen zu können.)

4. Der Kernel verschiebt sich im RAM-Speicher so, dass er bei der späteren Entkomprimierung nicht überschrieben werden kann.

Protected Mode

5. Der Kernel schaltet die CPU in den „Protected Mode“, da nur mit dieser Adressierungsart der Linuxkernel uneingeschränkt (und schnell) arbeiten kann.

Historisches

Direkt nach dem Einschalten der CPU (`Reset`) arbeitet jede x86-CPU wie der `8086`-Prozessor. Dieser Prozessor war der erste 16 Bit-Prozessor der Firma Intel und dient als „Basismodell“ für alle nachfolgenden CPUs. Der `8086`-Prozessor beherrscht nur den „Real Adress Mode“ zum Ansprechen des RAM-Speichers. Dabei wird ein 20 Bit-Adressraum mit nur 16 Bit breiten Registern bedient. Der `8086`-Prozessor adressiert immer nur 64 KByte große Segmente in einem RAM-Speicher, der maximal 1 MByte groß sein kann.

Die Nachfolgemodelle des `8086`-Prozessors sind in der Lage, den kompletten Adressraum mit 32 Bit linear zu adressieren („Protected Mode“).

Warum schaltet der Bootlader nicht die CPU direkt in den „Protected Mode“?

Beim Booten ist das sogenannte BIOS aktiv. Dieses BIOS ist im „Real Adress Mode“ programmiert. Da der Bootlader nur in Zusammenhang mit dem BIOS arbeiten kann, sind die ersten Schritte nicht im „Protected Mode“ zu erledigen. Erschwerend kommt hinzu, dass nur vom „Real Mode“ in den „Protected Mode“ umgeschaltet werden kann. Die umgekehrte Richtung funktioniert leider nur durch einen Reset.

In dem Listing 1 auf der Seite 13 sind Ausschnitte aus der Kerneldatei dargestellt, die im Kernel während des Abschnitts 1 der Initialisierung aktiv ist. Sie erkennen:

Architekturspezifischer Code

- Der Code ist architekturspezifisch (Zeile 1: Real Mode) und zum Teil in einer sehr frühen Phase (Zeile 4: 1991) von Linux erstellt worden.
- Es gibt „magische“ Zahlen, zum Beispiel die Adresse, auf die der Bootsektor nach dem Laden vom BIOS abgelegt wird (Zeile 24).
- Es wird nicht in Hochsprache, sondern in Assembler programmiert, da nur auf dieser Ebene bestimmte Einstellungen der CPU durchgeführt werden können (ab Zeile 26).

Listing 1: Ausschnitt der Kerneldatei `arch/x86/boot/header.S` (Kernel 2.6.38)

```
1  /*
2   *       header.S
3   *
4   *       Copyright (C) 1991, 1992 Linus Torvalds
5   *
6   *       Based on bootsect.S and setup.S
7   *       modified by more people than can be counted
8   *
9   *       Rewritten as a common file by H. Peter Anvin (Apr 2007)
10  *
11  * BIG FAT NOTE: We're in real mode using 64k segments.  ...
12  * addresses must be multiplied by 16 ...
13  * addresses. To avoid confusion, linear addresses ...
14  * hex while segment addresses ...  as segment:offset.
15  *
16  */
17  #include <asm/segment.h>
18  ...
19  #include <asm/e820.h>
20  #include <asm/page_types.h>
21  #include <asm/setup.h>
22  #include "boot.h"
23  ...
24 BOOTSEG  = 0x07C0   /* original address of boot-sector */
25  ...
26  .code16
27          .section ".bstext", "ax"
28          .global bootsect_start
29  bootsect_start:
30
31          # Normalize the start address
32          ljmp    $BOOTSEG, $start2
33  start2:
34          movw    %cs, %ax
```

```
35              movw    %ax, %ds
36  ...
```

Abschnitt 2:

Kerneldatei ist komprimiert

Entkomprimierung

1. Die Kerneldatei ist in der Regel komprimiert. Im Kernel ist eine Funktion, die ähnlich wie das Kommandozeilenprogramm `unzip` arbeitet, eingebaut, die es erlaubt, dass der Kernel sich selbst entpacken kann.

Startup_32

2. Im „Protected Mode“ führt der Kernel den Code der `Startup_32`-Funktion aus der Datei `arch/x86/boot/compressed/head_32.S` aus. Ausschnitte der Kerneldatei `arch/x86/boot/compressed/head_32.S`, die im Kernel während des Abschnitts 2 der Initialisierung aktiv ist, sind in dem Listing 2 auf der Seite 14 dargestellt. Hier wird der Kernel entpackt und dann gestartet.

Listing 2: Ausschnitt der Kerneldatei `arch/x86/boot/compressed/head_32.S` (Kernel 2.6.38)

```
1  /*
2   *  linux/boot/head.S
3   *
4   *  Copyright (C) 1991, 1992, 1993  Linus Torvalds
5   */
6
7  /*
8   *  head.S contains the 32-bit startup code.
9   *
10 ...
11 */
12
13 /*
14  * Copy the compressed kernel to the end of our buffer
15  * where decompression in place becomes safe.
16  */
17          pushl   %esi
18 ...
19 /*
20  * Do the decompression, and jump to the new kernel..
21  */
22          leal    z_extract_offset_negative(%ebx), %ebp
23 ...
```

Architekturunabhängiger Code

3. Nachdem der vollständig entpackte Kernel gestartet ist, werden in der Funktion `start_kernel` architekturunabhängige Initialisierungen vorgenommen (Systemtimer und Scheduler starten usw).

Ausschnitte der Kerneldatei `init/main.c`, die im Kernel während des Abschnitts 2 der Initialisierung aktiv ist, sind in dem Listing 3 auf der Seite 15 dargestellt.

Listing 3: Ausschnitt der Kerneldatei `init/main.c` (Kernel 2.6.38)

```
1  ... start_kernel(void)
2  {
3   smp_setup_processor_id();
4  ...
5    tick_init();
6    boot_cpu_init();
7    page_address_init();
8    build_all_zonelists(NULL);
9    page_alloc_init();
10
11   printk(KERN_NOTICE "Kernel command line: %s\n",
12                                   boot_command_line);
13   parse_early_param();
14   parse_args("Booting kernel", static_command_line,
15                 __start___param,
16                 __stop___param - __start___param,
17                &unknown_bootoption);
18 ...
19   sched_init();
20   init_IRQ();
21 ...
22   init_timers();
23   hrtimers_init();
24 ...}
```

4. An diesem Punkt bietet sich uns eine Gelegenheit, den Kernel in seinem Lauf relativ einfach zu lenken. Über den Bootlader können wir dem Kernel Parameter übergeben, die in der Funktion `start_kernel` ausgewertet werden können. Die Liste mit diesen Parametern heißt `Kernel Command Line` und sieht zum Beispiel so aus:

Kernel Command Line

   ```
   console=ttyS0,115200 root=ramfs BOOT_IMAGE=linux
   ```

   Es gibt eine Vielzahl von Parametern, denen wir Werte zuweisen können, die später vom Kernel ausgewertet werden. Damit können Sie zum Beispiel ohne Compilierung bestimmte Funktionen des Kernels lenken, ein- oder ausschalten. In dem obigen Beispiel legen Sie Folgendes fest:

- Die Aus- und Eingaben des Kernels, die ansonsten auf dem Bildschirm des angeschlossenen Terminals ausgegeben werden, werden auf die serielle Schnittstelle `ttyS0` umgelenkt. Diese Schnittstelle soll auf die Geschwindigkeit `115200` Baud eingestellt werden.
- Nach dem Booten dient das Dateisystem `ramfs` als Root-Dateisystem.
- Die Kerneldatei heißt `linux`.

(In einem späteren Kapitel wird Ihnen vorgestellt, wie Sie über den Bootlader dem Kernel die `Kernel Command Line` übergeben können. Eine Liste mit erlaubten Parametern finden Sie unter [KCL]).

5. Der Kernel ist nun bereit, Aufgaben zu erfüllen, die von außen an ihn herangetragen werden. Dem Ausschnitt der Kerneldatei `init/do_mounts_initrd.c`, der in dem Listing 4 auf der Seite 16 dargestellt ist, ist zu entnehmen, dass `/linuxrc` im Root-Verzeichnis gestartet wird. `/linuxrc` kann ein Skript oder eine compilierte Datei sein, die Initialisierungen für die Anwendungen unseres Embedded Systems ausführt.

   Listing 4: Ausschnitt der Kerneldatei `init/do_mounts_initrd.c` (Kernel 2.6.38)

```
1  ...
2   sys_chdir("/root");
3   sys_mount(".", "/", NULL, MS_MOVE, NULL);
4   sys_chroot(".");
5
6   /*
7    * In case that a resume from disk is carried out by linuxrc
8    *    or one of
9    * its children, we need to tell the freezer not to wait for us.
10  */
11  current->flags |= PF_FREEZER_SKIP;
12
13  pid = kernel_thread(do_linuxrc, "/linuxrc", SIGCHLD);
14  if (pid > 0)
15          while (pid != sys_wait4(-1, NULL, 0, NULL))
16                  yield();
17  ...
```

6. Weitere Initialisierungen übernimmt der `init`-Prozess. Dieser Prozess wird in der Kerneldatei `init/main.c` gestartet, die in dem Listing 5 auf der Seite 17 dargestellt ist.

   Wir erkennen, dass der Kernel die `init`-Datei in verschiedenen Verzeichnissen des Root-Dateisystems sucht:

(1) `/sbin/init`
(2) `/etc/init`
(3) `/bin/init`

Wenn keine `init`-Datei gefunden werden konnte, versucht der Kernel eine Shell zu starten (`/bin/sh`-Datei). Mit einer Shell hätten Anwender Gelegenheit, Kommandos beispielsweise über Tastatur einzugeben. Scheitert auch der Versuch, eine Shell zu starten, wird ein Kernel-Crash ausgelöst.

Shell starten

Kernel-Crash

Listing 5: Ausschnitt der Kerneldatei `init/main.c` (Kernel 2.6.38)

```
1  /*
2   * We try each of these until one succeeds.
3   *
4   * The Bourne shell can be used instead of init if we are
5   * trying to recover a really broken machine.
6   */
7   if (execute_command) {
8     run_init_process(execute_command);
9     printk(KERN_WARNING "Failed to execute %s.  Attempting "
10                         "defaults...\n", execute_command);
11  }
12  run_init_process("/sbin/init");
13  run_init_process("/etc/init");
14  run_init_process("/bin/init");
15  run_init_process("/bin/sh");
16
17  panic("No init found.  Try passing init= option to kernel. "
18        "See Linux Documentation/init.txt for guidance.");
19  ...
```

Wenn davon ausgegangen werden kann, dass wir über einen funktionierenden Kernel verfügen, dann geht die Verantwortung für das Gesamtsystem in der letzten Phase des Abschnitts 2 auf uns über.
Unsere Aufgaben sind:

1. Wir müssen eine `init`-Datei zur Verfügung stellen.

2. Diese `init`-Datei kann an verschiedenen Stellen im Root-Verzeichnisbaum liegen.

3. Falls es Probleme mit der `init`-Datei gibt, können wir eine Shell bereithalten, die dann aufgerufen wird.

4. Wenn es Probleme bei der Initialisierung gibt, die der Kernel nicht lösen kann, wird die Meldung `No init found. Try passing ...` auf dem Terminal ausgegeben.

5. Sollten Dinge erledigt werden, bevor `init` gestartet wird, dann ist `/linuxrc` dafür zuständig.

2.4 Der init-Prozess

Allererster Prozess

Durch die Ausführung der `init`-Datei entsteht der allererste Prozess im System, der an die Belange der Anwender angepasst werden kann. Der `init`-Prozess hat die PID 1 und ist deshalb in der Vielzahl der Prozesse eines kompletten Systems leicht zu identifizieren. `init` startet direkt oder indirekt alle nachfolgenden Prozesse. Sollte `init` terminieren oder sogar abstürzen, dann können keine weiteren Prozesse mehr gestartet werden.

PID 1

Wichtig

Die Aufgaben von `init` werden in Desktop-Systemen durch vielfältige skriptähnliche Tabellen gesteuert.
Für die Bedürfnisse eines Embedded Systems reicht in der Regel Folgendes aus:

1. Unser Embedded System verfügt über eine Reihe von Applikationen, die von einer zentralen Applikation gesteuert werden. In diesem Fall reicht es aus, wenn diese zentrale Applikation von uns als `init`-Prozess deklariert wird. Konkret heißt dies, dass diese Applikation `/sbin/init`, `/etc/init` oder `/bin/init` heißen muss oder dass wir Softlinks erzeugen, die `init` heißen und auf unsere Applikation verweisen.

Applikation heißt init

2. Sind die Anforderungen komplexer, dann bedienen wir uns der `busybox`. In dieser Programmsammlung existiert eine `init`-Implementierung, die von uns mittels einer Konfigurationsdatei an unseren Bedarf angepasst werden kann.

busybox: init

2.5 Zusammenfassung

Der Hauptspeicher eines Systems wird in zwei Bereiche aufgeteilt:

- den User-Space, den die Applikationen der Anwender nutzen, und den **User-Space**
- Kernel-Space, der für Kernelzwecke reserviert ist. **Kernel-Space**

User- und Kernel-Space existieren gleichzeitig im Adressraum einer CPU. Nur so ist es möglich, dass aus dem User-Space eine Servicefunktion des Kernels aufgerufen werden kann.
Unter Linux (32 Bit) ist der Adressraum so aufgeteilt, dass

- die untersten 3 GByte zum User-Space und
- das oberste GByte zum Kernel-Space gehören.

Die Quellen des offiziellen Linuxkernels finden Sie unter `ftp.kernel.org`.
Neben dem offiziellen Linuxkernel unter `ftp.kernel.org` gibt es noch weitere. **Offizieller Kernel**

Sie können sich einen Linuxkernel herunterladen und an Ihren Bedarf anpassen. Nach dem Entkomprimieren starten Sie mit dem Kommando `make menuconfig` einen Dialog auf Ihrem Terminal, in dem Sie die Bestandteile Ihres Kernels auswählen können. (Alle Antworten, die Sie beim Konfigurieren des Kernels eingeben, werden in einer Datei `.config` gespeichert.) Mit der Eingabe `make` starten Sie das Zusammenbauen des Kernels.
Im Anschluss liegen Ihnen im Wesentlichen zwei Komponenten vor: eine Datei mit Adressinformationen und der Kernel selbst. Wenn Sie die beiden Dateien in Ihr `/boot`-Verzeichnis kopieren würden, sollten Sie die Namen `System.map-2.6.38-1` bzw. `vmlinuz-2.6.38-1` verwenden (2.6.38-1 ist die Linuxversionsnummer). (Üblich ist es, auch die `.config`-Datei dort als `config-2.6.38-1` zu deponieren.)
Einen schnellen Test unserer Arbeit können wir mit dem Emulator `QEMU` durchführen.
Die Kernelinitialisierung beginnt

- beim Einschalten der CPU und endet
- mit dem Starten des ersten Prozesses im User-Space.

Die einzelnen Phasen der Initialisierung lassen sich grob in zwei Abschnitte aufteilen. Der erste Abschnitt sieht so aus:

1. Nach dem Einschalten startet die CPU auf einer von ihrem Hersteller fest definierten Adresse. Diese Adresse ist die Startadresse des BIOS-Programms.

2. Das BIOS sucht auf den bekannten Datenträgern des Systems nach einem Bootlader und startet diesen.

3. Der Bootlader lädt die Datei mit dem Kernel vom Datenträger in das RAM und bringt den Kernel zur Ausführung. Danach werden einige Geräte oder Hardwarekomponenten des Mainboards initialisiert.

Einige Geräte initialisieren

4. Der Kernel verschiebt sich im RAM-Speicher so, dass er bei der späteren Entkomprimierung nicht überschrieben werden kann.

Protected Mode

5. Der Kernel schaltet die CPU in den „Protected Mode“, da nur mit dieser Adressierungsart der Linuxkernel uneingeschränkt (und schnell) arbeiten kann.

Im zweiten Abschnitt der Initialisierung wird:

Kerneldatei ist komprimiert

1. Die Kerneldatei ist in der Regel komprimiert. Der Kernel besitzt eine Funktion, die es erlaubt, dass er sich selbst entpacken kann.

Startup_32

2. Im „Protected Mode“ führt der Kernel die `Startup_32`-Funktion aus.

Architekturunabhängiger Code

3. Nachdem der vollständig entpackte Kernel gestartet ist, werden in der Funktion `start_kernel` architekturunabhängige Initialisierungen vorgenommen.

Kernel Command Line

4. Über den Bootlader können wir dem Kernel Parameter übergeben, die in der Funktion `start_kernel` ausgewertet werden können. Die Liste mit diesen Parametern heißt `Kernel Command Line`.

Shell starten Kernel-Crash

5. Der Kernel ist nun bereit, Aufgaben zu erfüllen, die von außen an ihn herangetragen werden. Zuerst wird `/linuxrc` im Root-Verzeichnis gestartet. Danach wird der `init`-Prozess gestartet. Wenn keine `init`-Datei gefunden werden konnte, versucht der Kernel eine Shell zu starten (`/bin/sh`-Datei). Scheitert auch der Versuch, eine Shell zu starten, wird ein Kernel-Crash ausgelöst.

Durch die Ausführung der `init`-Datei entsteht der allererste Prozess im System, der an die Belange der Anwender angepasst werden kann. Der `init`-Prozess hat die PID 1 und ist deshalb in der Vielzahl der Prozesse eines kompletten Systems leicht zu identifizieren. `init` startet direkt oder indirekt alle nachfolgenden Prozesse. Sollte `init` terminieren oder sogar abstürzen, dann können keine weiteren Prozesse mehr gestartet werden.

Allererster Prozess

PID 1

Wichtig

Für die Bedürfnisse eines Embedded Systems reicht in der Regel Folgendes aus:

1. Unser Embedded System verfügt über eine Reihe von Applikationen, die von einer zentralen Applikation gesteuert werden. In diesem Fall reicht es aus, wenn diese zentrale Applikation von uns als `init`-Prozess deklariert wird. Konkret heißt dies, dass diese Applikation `/sbin/init`, `/etc/init` oder `/bin/init` heißen muss oder dass wir Softlinks erzeugen, die `init` heißen und auf unsere Applikation verweisen.

   Applikation heißt init

2. Sind die Anforderungen komplexer, dann bedienen wir uns der `busybox`. In dieser Programmsammlung existiert eine `init`-Implementierung, die von uns mittels einer Konfigurationsdatei an unseren Bedarf angepasst werden kann.

   busybox: init