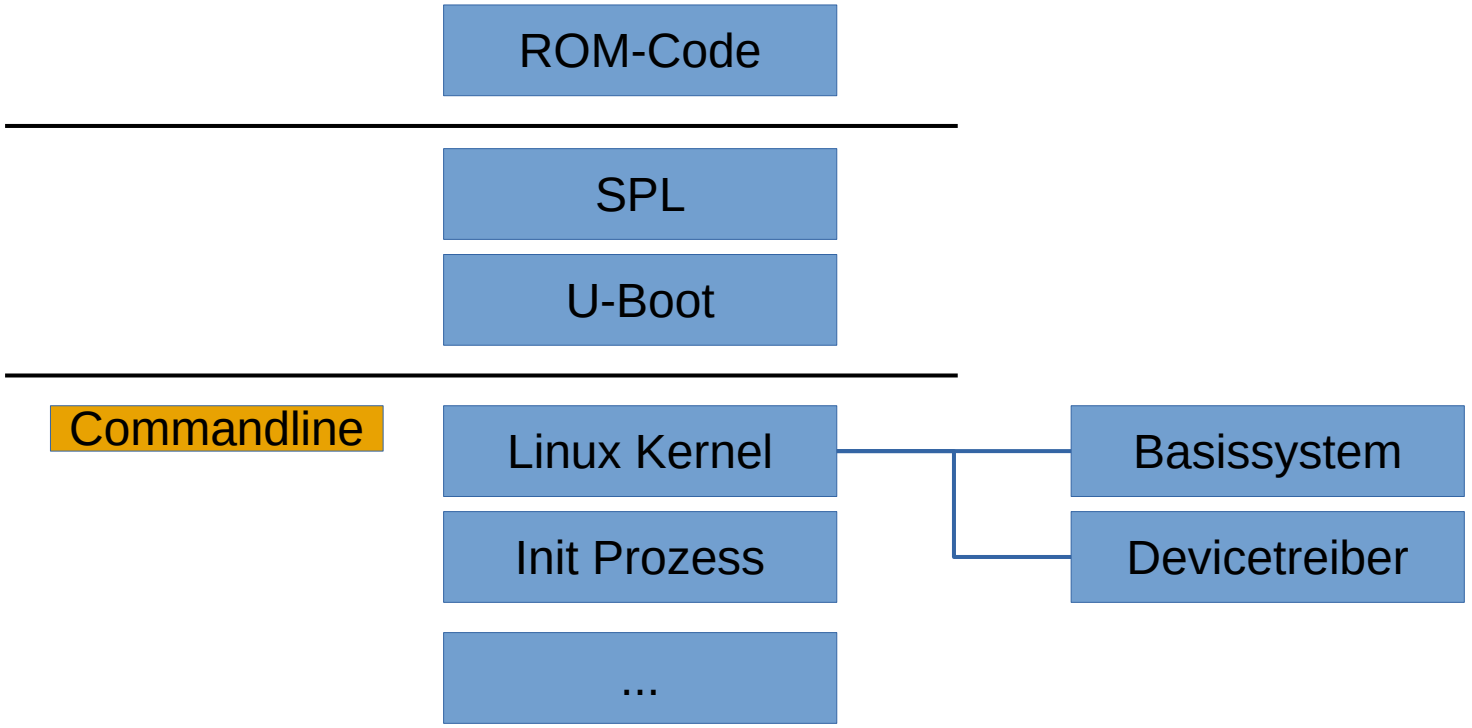
Linux Treiber

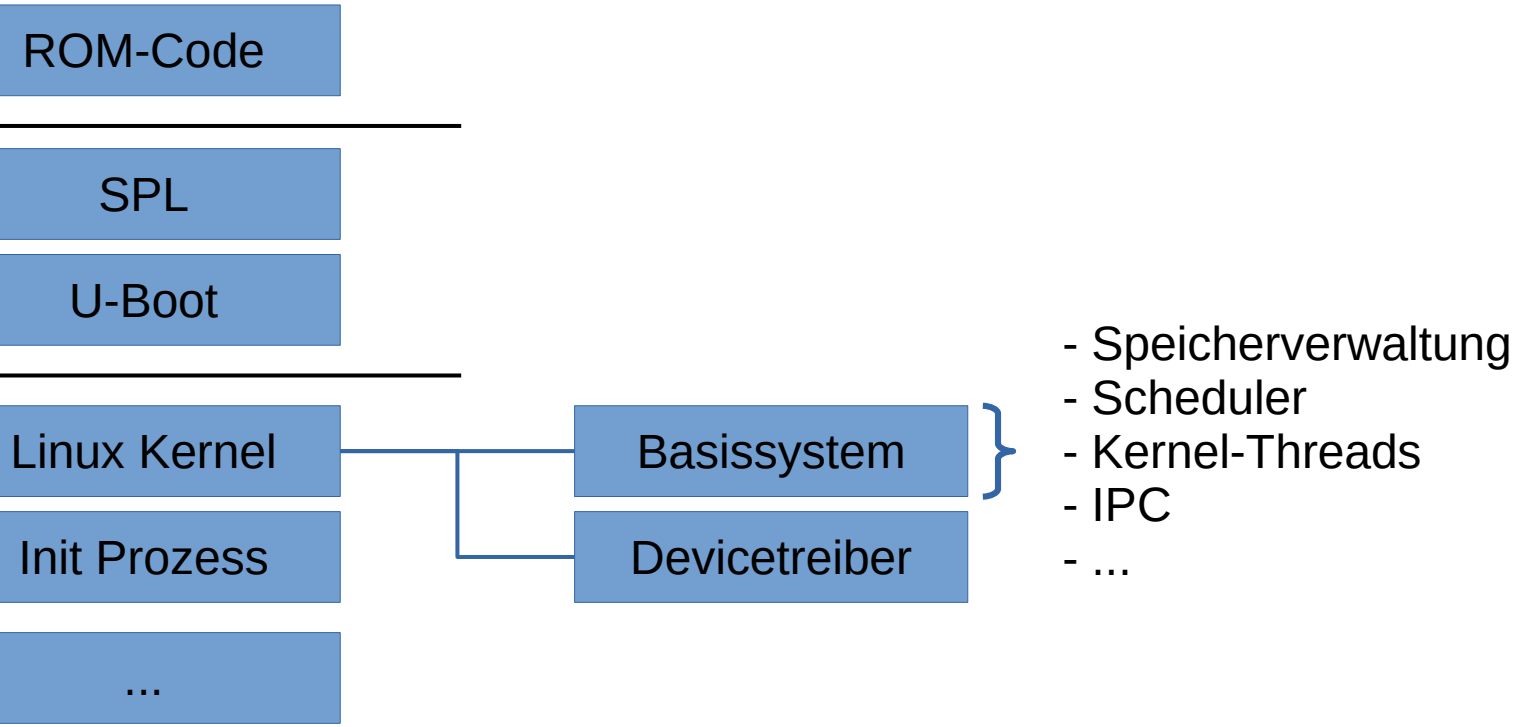
EIM/INFM

Frank Erdrich
frank.erdrich@emtrion.de

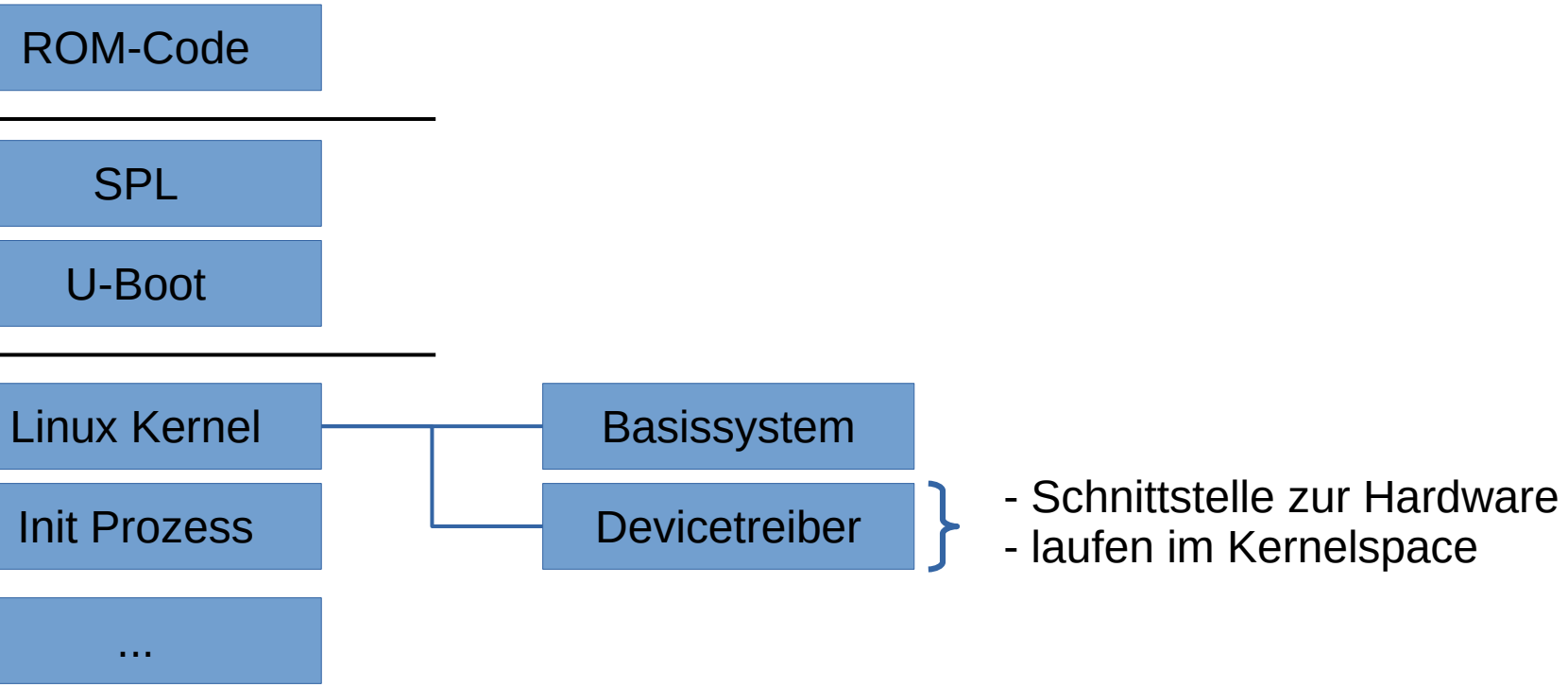
Aktueller Stand



Aktueller Stand



Aktueller Stand

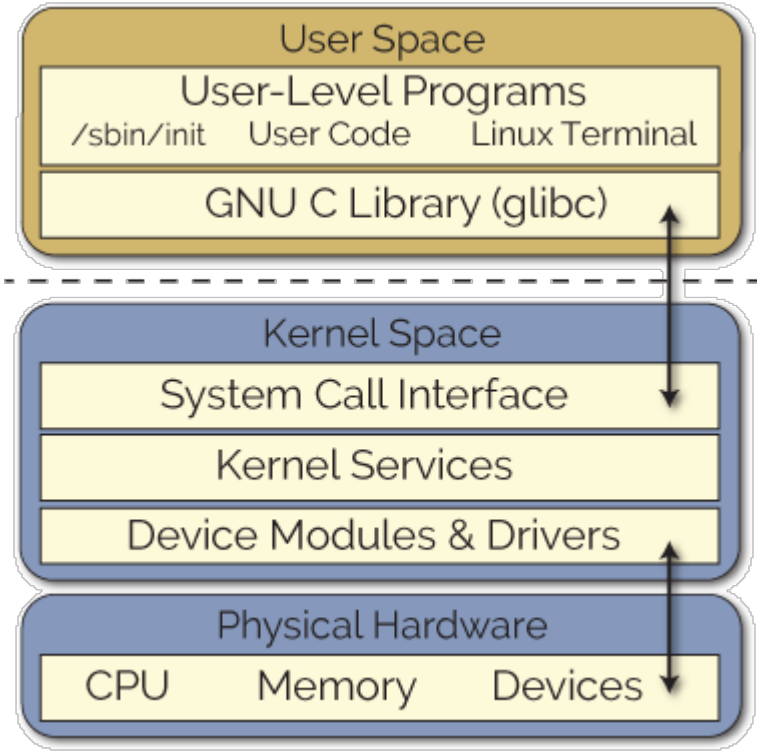


Recap User-/Kernelspace

- Trennung zwischen Userspace und Kernelspace
- Warum?
- Welche Vor- und Nachteile?

Gerätetreiber

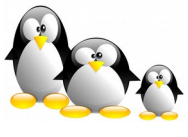
- Interface zur Hardware
- Treibertypen
  - Character
  - Block
  - Other (bspw. Netzwerk und USB)



Quelle: http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction

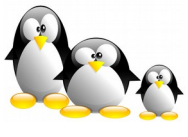
Gerätetreiber

- Initialisieren die Hardware
- Schnittstelle zum Userspace
  - SysFS
  - Input/Output-Control (ioctl)
- Powermanagement
- Konfiguration (etwa über Devicetree)



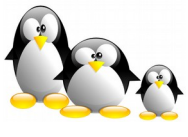
Gerätetreiber

- Alles ist eine Datei (mit Ausnahmen)
- Treiber implementieren File-IO
  - open()
  - read()
  - write()
  - close()
  - ...



Gerätetreiber

- Haben in der Regel einen Eintrag im /dev/-Verzeichnis (/dev/ttyUSB0, /dev/hwrng, ...)
- Character-Devices
  - lesen/schreiben zeichenweise (Tastatur, serielle Schnittstelle, ...)
- Block-Devices
  - Blockbasierte Zugriffe (etwa Festspeicher u. ä.)



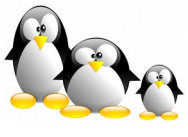
Gerätetreiber

- Char- und Blockdevices werden über Nummern identifiziert
  - Major: Gerätetyp (Treiberidentifizierung)
  - Minor: Instanz des Gerätetyps
- Major-Nummer kann fest oder dynamisch vergeben werden

```
int register_chrdev(unsigned int major, const char *name,
                    const struct file_operations *fops)
```

Kernelmodule

- Gerätetreiber können als Modul gebaut werden (sind nicht fest im Kernel integriert)
  - sind in folgendem Verzeichnis gespeichert: /lib/modules/$(uname -r)
- Module lassen sich zur Laufzeit laden und entladen
- Vereinfacht die Entwicklung

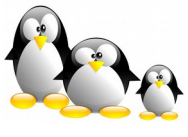


Kernelmodule Build

- Kann über menuconfig eingestellt werden
- Extra Kernel-Buildstep notwendig
  - `$ make modules`
  - `$ make modules_install INSTALL_MOD_PATH=/path/to/rootfs`

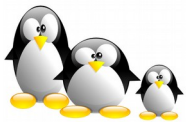
Kernelmodule (ent)laden

- `$ modprobe (-r) modulname`
- Alternativ (löst keine Abhängigkeiten auf):
  - `$ insmod modulname`
  - `$ rmmod modulname`
- Geladene Module auflisten
  - `$ lsmod`
- Info zu eine Modul
  - `$ modinfo modulname`



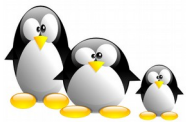
Modul-Besonderheiten

- Arbeiten nach dem Event-Prinzip
- Kein printf, da keine stdlib
- Höherer Privilegierungslevel als User-Space
- Kein Floating-Point Support
- Muss selbst verwendete Ressourcen aufräumen
- Nebenläufigkeit beachten (reentrant!?)



Exkurs: printk

- Log-Nachrichten des Kernels
  - keine stdlib, deshalb kein printf
- Verschiedene Log-Level
  - Immer die Macros verwenden (siehe *linux/kern_levels.h*)
- Abrufbar über *dmesg* oder auslesbar aus /var/log/messages

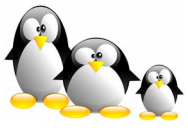


Loglevel

Level	Name	Bedeutung
0	KERN_EMERG	system is unusable
1	KERN_ALERT	action must be taken immediately
2	KERN_KRIT	critical conditions
3	KERN_ERR	error conditions
4	KERN_WARNING	warning conditions
5	KERN_NOTICE	normal but significant condition
6	KERN_INFO	informational
7	KERN_DEBUG	debug-level messages

Über Kernel Commandline mit „loglevel=level“ einstellbar.

level < loglevel ist immer mit aktiv



Exkurs: printk

Prototyp

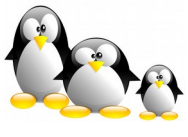
```
int printk(const char *fmt, ...)
```

Verwendung

```
printk(„%s: Log Message %d\n“, __func__, 1);
printk(KERN_ERR „%s: an error has occured\n“, __func__);
```

Alternativ (siehe include/linux/printk.h)

```
pr_emerg(fmt, ...)
pr_alert(fmt, ...)
pr_crit(fmt, ...)
pr_err(fmt, ...)
pr_warning(fmt, ...) oder pr_warn(fmt, ...)
pr_notice(fmt, ...)
pr_info(fmt, ...)
```



Aufbau Treiber (Modul)

- Header und Metadaten

```
#include <linux/init.h>      // Macros für Funktionen, z.B. __init, __exit
#include <linux/module.h>    // Header für ladbare Kernelmodule
#include <linux/kernel.h>    // diverses, printk und andere Macros

MODULE_LICENSE("GPL");       // Modullizenz
MODULE_AUTHOR("Ich");        // Author, siehe modinfo
MODULE_DESCRIPTION("Simple driver example.");  // siehe modinfo
MODULE_VERSION("0.1");       // Modulversion
```

Aufbau Treiber (Modul)

- Modulrahmen

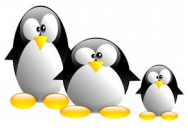
```
/* Modul init function */
static int __init sample_init(void){
   printk(KERN_INFO "Loading sample module.\n“);
   return 0;
}

/* Modul cleanup function */
static void __exit sample_exit(void){
   printk(KERN_INFO "Exit sample module.\n");
}

/* Macros for modul startup and cleanup */
module_init(sample_init);
module_exit(sample_exit);
```

Modulinit

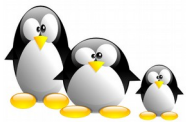
- `module_init` verweist auf `__initcall(x)` (siehe include/linux/module.h)
- `__initcall` verweist auf `device_initcall(x)` (siehe include/linux/init.h)
- Intern baut der Compiler/Linker Listen aus Funktionszeigern in speziellen Sektionen auf
- Diese werden vom Kernel zur Laufzeit abgearbeitet



Modulparameter

- Module können mit Parameter aufgerufen werden (ähnlich Userspace Apps)
- $ modinfo zeigt mögliche Parameter mit Datentyp an
- Beispiel KVM

```
parm:           ignore_msrs:bool
parm:           report_ignored_msrs:bool
parm:           min_timer_period_us:uint
...
```



Aufbau Treiber (Modul)

- Modulparameter
  - Sind über modinfo einsehbar
  - Bei erfolgreich geladenem Modul auch unter /sys/module/modname/parameters

```
static char *param = "\0";           // leerer Beispielparameter
module_param(param, charp, S_IRUGO); // char ptr, S_IRUGO read only
MODULE_PARM_DESC(param, "Beschreibung des Parameters");
```

Für mögliche Datentypen, siehe include/linux/moduleparam.h
Berechtigungen, siehe include/linux/stat.h

Aufbau Treiber (Modul)

- Register driver

```
static struct file_operations fops = {
    .read = sample_read,
    .write = sample_write,
    .open = sample_open,
    .release = sample_release
};

#define DEVICE_NAME „sample_chrdev“
static int major = 0;

/* Modul init function */
static int __init sample_init(void)
{
    major = register_chrdev(0, DEVICE_NAME, &fops);

    if (Major < 0) {
      printk(KERN_ALERT "Registering simple device failed with %d\n", major);
      return major;
    }
    ...
}
```

Aufbau Treiber (Modul)

- Unregister driver

```
...

/* Modul cleanup function */
static void __exit sample_exit(void){
   int ret = unregister_chrdev(Major, DEVICE_NAME);
   if (ret < 0)
      printk(KERN_ALERT "Error in unregister_chrdev: %d\n", ret);
}

/* Macros for modul startup and cleanup */
module_init(sample_init);
module_exit(sample_exit);
```

Aufbau Treiber (Modul)

- Fileoperationen

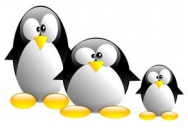
```
//struct file_operations {  // siehe include/linux/fs.h
//  ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
//  ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
//  int (*open) (struct inode *, struct file *);
//  int (*release) (struct inode *, struct file *);
//};

static struct file_operations fops = {
    .read = sample_read,
    .write = sample_write,
    .open = sample_open,
    .release = sample_release
};

static ssize_t sample_read(struct file *, char __user *, size_t, loff_t*)
{
    ...
}
```

Aufbau Treiber (Modul)

- Device wird nicht in /dev/ erstellt, was tun?
  - Major und Minor des Treibers herausfinden (Ausgaben im Treiber)
  - $ mknod /dev/sample-chrdev c 254 0
  - Alternativ im Treibercode



Aufbau Treiber (Modul)

- Device in /dev/ automatisch erstellen

```
#define DEVICE_NAME „sample_chrdev“
#define CLASS_NAME „chrdev“

static struct class*  sampleClass  = NULL;
static struct device* sampleDevice = NULL;

static int __init sample_init(void)
{
    // register_chrdev
    major = register_chrdev(0, DEVICE_NAME, &fops);

    // register device class
    sampleClass = class_create(THIS_MODULE, CLASS_NAME);

    // register device driver
    sampleDevice = device_create(sampleClass, NULL, MKDEV(major, 0), NULL, \
                          DEVICE_NAME);
}
```

Aufbau Treiber (Modul)

- In /dev/ erstelltes device aufräumen

```
#define DEVICE_NAME „sample_chrdev“
#define CLASS_NAME „chrdev“

static struct class*  sampleClass  = NULL;
static struct device* sampleDevice = NULL;

static int __exit sample_exit(void)
{
    device_destroy(sampleClass, MKDEV(major, 0));
    class_unregister(sampleClass);
    class_destroy(sampleClass);
    unregister_chrdev(major, DEVICE_NAME);
}
```