

Boot-up-Grafik unter Linux

Thomas Zimmermann

SUSE Software Solutions Germany GmbH

März 2026

Wenn der Grafiktreiber streikt, dann

nomodeset

Ich bin

- ▶ Thomas Zimmermann
- ▶ Software-Entwickler bei SUSE
- ▶ Co-Maintainer im DRM-Subsystems des Kernels

Von der Firmware zum Login

Thema des Vortrags ist der Start des Grafikstacks vom Einschalten des Computers bis zum Login-Prompt des Linux-Desktops.

Der Grafikstack in aller Kürze

- ▶ Anwendung
 - ▶ zeichnet nur eigene Ausgabe
 - ▶ Rendering übernimmt Mesa (OpenGL, Vulkan; Hardware oder Software)
 - ▶ sendet Frame per Wayland an Compositor
- ▶ Compositor
 - ▶ Systemdienst oder Linux-Desktop
 - ▶ generiert Gesamtausgabe aus Anwendungs-Ausgabe (auch per Mesa)
 - ▶ sendet Display-Modus und Gesamtausgabe an Kernel-Treiber
- ▶ Kernel
 - ▶ DRM-Treiber programmieren Hardware
- ▶ IPC
 - ▶ Wayland als Compositor-Protokoll
 - ▶ *dma-buf*-Schnittstelle für effizienten Datenaustausch zwischen Hardware, Treibern, Compositor, Anwendungen

Was passiert eigentlich vorher?

1. Einschalten
2. POST (Firmware)
3. Bootloader laden
4. Linux-Kernel und erste Treiber laden
5. Userspace starten
 - ▶ Linux-Treiber laden und Geräte initialisieren
 - ▶ Systemdienste starten

Unser Ziel:

- ▶ Einfache Möglichkeit Grafikmodus einzustellen
- ▶ Einfacher Zugriff auf Grafikspeicher
- ▶ **Ohne** die Details der Hardware zu kennen

Firmware und Bootloader

Firmware

- ▶ CPU startet System-Firmware an vordefinierter Speicher-Adresse
- ▶ UEFI, BIOS, SoC-Firmware, Coreboot
- ▶ System-Firmware startet Geräte-Firmware
- ▶ Firmware bietet Grafikschnittstellen *unabhängig* vom Grafikchip

VESA BIOS Extension (VBE)

- ▶ VESA BIOS Extension 3.0 (Sep 1998) für 16/32-bit PC
- ▶ Immer noch in Qemu verwendet
- ▶ 8/15/16/24/32-bit Grafik, linearer Framebuffer, Page mapping
- ▶ Grafikkarte kann per SW-Interrupt INT 10h oder PMI programmiert werden

INT 10h: BIOS Video Interrupt

```
1 struct vesa_mode_info vminfo;
2 struct biosregs ireg, oreg;
3
4 ireg.ax = 0x4f01; // Mode-Info lesen
5 ireg.cx = 0x116; // 1024x768, XRGB1555
6 ireg.di = (size_t)&vminfo;
7 intcall(0x10, &ireg, &oreg); // Interrupt ausfuehren
8
9 if (oreg.ax == 0x004f) {
10     // vminfo enthaelt Mode-infos
11 } else {
12     // Fehler
13 }
14
15 // Linux-Kernel, boot-vesa.c
```

struct vesa_mode_info

```
1 struct vesa_mode_info {
2     ...
3     u16 logical_scan;    /* Zeilenlaenge in Bytes */
4     u16 h_res;           /* Pixel pro Zeile */
5     u16 v_res;           /* Zeilen */
6     ...
7     u8 bpp;              /* Bits pro Pixel */
8     ...
9     u8 rmask;            /* Anzahl der Rot-Bits */
10    u8 rpos;              /* Rot-Position */
11    u8 gmask;
12    u8 gpos;
13    u8 bmask;
14    u8 bpos;
15    u8 resv_mask;
16    u8 resv_pos;
17    ...
18    u32 lfb_ptr;          /* LFB-Adresse */
19    ...
20 } __attribute__((packed));
21
22 // Linux-Kernel, vesa.h
```

Funktionen des VBE

0x4f00 VBE-Controller abfragen

0x4f01 Mode-Info abfragen

0x4f02 Modus programmieren

0x4f03 Aktuellen Modus abfragen

0x4f09 Farbpalette abfragen/einstellen

Aber...

- ▶ VBE funktioniert nur im 16/32-bit Modus; kein 64-bit
- ▶ daher nur im BIOS und Bootloader (GRUB) nutzbar
- ▶ Videomodus muss eingestellt werden, bevor Kernel den Protected Mode startet
- ▶ programmierter Displaymodus bleibt auch im PM, Framebuffer schreibbar
- ▶ Bootloader übergibt dem Kernel die Mode-Info des Framebuffers

struct boot_params

```
1 struct boot_params {  
2     struct screen_info screen_info; /* Offset 0x000 */  
3     ...  
4     struct edid_info edid_info;      /* Offset 0x140 */  
5     ...  
6 } __attribute__((packed));  
7  
8 // Linux-Kernel, bootparam.h
```

- ▶ Kernel-Binary enthält globale Variable boot_params
- ▶ Bootloader schreibt the Werte direkt in boot_params und startet den Kernel
- ▶ Kernel liest boot_params während des Bootens

struct screen_info

```
1 struct screen_info {
2     ...
3     __u8 orig_video_isVGA; /* Offset 0x0f */
4     ...
5     __u16 lfb_width; /* Offset 0x12 */
6     __u16 lfb_height; /* Offset 0x14 */
7     __u16 lfb_depth; /* Offset 0x16 */
8     __u32 lfb_base; /* Offset 0x18 */
9     __u32 lfb_size; /* Offset 0x1c */
10    ...
11    __u16 lfb_linelength; /* Offset 0x24 */
12    __u8 red_size; /* Offset 0x26 */
13    __u8 red_pos; /* Offset 0x27 */
14    __u8 green_size; /* Offset 0x28 */
15    __u8 green_pos; /* Offset 0x29 */
16    __u8 blue_size; /* Offset 0x2a */
17    __u8 blue_pos; /* Offset 0x2b */
18    __u8 rsvd_size; /* Offset 0x2c */
19    __u8 rsvd_pos; /* Offset 0x2d */
20    ...
21 } __attribute__((packed));
22
23 #define VIDEO_TYPE_VLFB 0x23 /* VESA VGA in graphic mode */
24 #define VIDEO_TYPE_EFI 0x70 /* EFI graphic mode */
25
26 // Linux-Kernel, screen_info.h
```

Unified Extensible Firmware Interface (UEFI) (x86)

- ▶ Unified Extensible Firmware Interface 2.11 final (März 2019)
- ▶ BIOS-Nachfolger
- ▶ ähnliche Grafikfunktionen wie VBE, aber anderes Interface
- ▶ Zugriff auf Geräte über *UEFI Handle*
- ▶ Jedes Handle bietet *UEFI Protocols*
- ▶ Jedes Protocol bietet *Interfaces*
- ▶ z.B. *Graphics Output Protocol* mit den Interfaces
 - QueryMode* Fragt den aktuellen Modus ab
 - SetMode* Stellt einen Display Modus ein

EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE

```
1  typedef struct {
2      UINT32                                     MaxMode;
3      UINT32                                     Mode;
4      EFI_GRAPHICS_OUTPUT_MODE_INFORMATION      **Info;
5      UINTN                                     SizeOfInfo;
6      EFI_PHYSICAL_ADDRESS                      FrameBufferBase;
7      UINTN                                     FrameBufferSize;
8  } EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE;
9
10 typedef struct {
11     UINT32                                     Version;
12     UINT32                                     HorizontalResolution;
13     UINT32                                     VerticalResolution;
14     EFI_GRAPHICS_PIXEL_FORMAT                 PixelFormat;
15     EFI_PIXEL_BITMASK                         PixelInformation;
16     UINT32                                     PixelsPerScanLine;
17 } EFI_GRAPHICS_OUTPUT_MODE_INFORMATION;
18
19 typedef struct {
20     UINT32 RedMask;
21     UINT32 GreenMask;
22     UINT32 BlueMask;
23     UINT32 ReservedMask;
24 } EFI_PIXEL_BITMASK;
25
26 // UEFI 2.8, Sec 12.9
```

Aber...

- ▶ UEFI unterscheidet zwischen Boottime- und Runtime-Services
- ▶ GOP ist ein Boottime-Service und nur im Bootloader verfügbar
- ▶ Selbe Lösung wie für VESA: Bootloader setzt Grafikmodus und schreibt Daten in `boot_params`

Unified Extensible Firmware Interface (UEFI) (non-x86)

- ▶ UEFI lädt Kernel-Binary direkt
- ▶ EFI Boot Stub des Kernels initialisiert Framebuffer und screen_info

Devicetree (non-x86)

- ▶ SoC-Firmware initialisiert Hardware und Display
- ▶ Devicetree-Dateiformat beschreibt Hardware auf SoC; Kernel benutzt Hardware entsprechend der Beschreibung
- ▶ Baum-Struktur mit Knoten für Geräte und deren Ressourcen
- ▶ Knoten unter */chosen* beschreiben aktuelle Konfiguration
 - ▶ *simple-framebuffer*
 - ▶ *display* (PowerPC)

simple-framebuffer

```
1 chosen {
2     #address-cells = <1>;
3     #size-cells = <1>;
4     framebuffer0: framebuffer@1d385000 {
5         compatible = "allwinner,simple-framebuffer", "simple-framebuffer";
6         allwinner,pipeline = "de_be0-lcd0";
7         reg = <0x1d385000 3840000>;
8         width = <1600>;
9         height = <1200>;
10        stride = <3200>;
11        format = "r5g6b5";
12        clocks = <&ahb_gates 36>, <&ahb_gates 43>, <&ahb_gates 44>;
13        lcd-supply = <&reg_dc1sw>;
14        display = <&lcdc0>;
15    };
16 };
17
18 // Linux-Kernel, simple-framebuffer.yaml
```

Coreboot

- ▶ Freie Alternative zu UEFI, BIOS, usw.
- ▶ Erstellt Tabelle mit Konfiguration, z.B. Framebuffer

```
1  struct cb_framebuffer {
2      u32 tag;
3      u32 size;
4
5      u64 physical_address;
6      u32 x_resolution;
7      u32 y_resolution;
8      u32 bytes_per_line;
9      u8 bits_per_pixel;
10     u8 red_mask_pos;
11     u8 red_mask_size;
12     u8 green_mask_pos;
13     u8 green_mask_size;
14     u8 blue_mask_pos;
15     u8 blue_mask_size;
16     u8 reserved_mask_pos;
17     u8 reserved_mask_size;
18     u8 orientation;
19     struct cb_framebuffer_flags flags;
20     u8 pad;
21 };
22
23 // Coreboot, coreboot_tables.h
```

Linux

Geräte und Treiber

- ▶ Kernel erstellt ein Gerät pro Framebuffer
- ▶ Verschiedene DRM-Treiber, je nach Schnittstelle
- ▶ Klein genug für das Kernel-Binary

Gerät	DRM-Treiber	Schnittstelle
coreboot-framebuffer	corebootdrm	Coreboot
efi-framebuffer	efidrm	UEFI
of-display	ofdrm	DT via <i>display</i>
simple-framebuffer	simplifiedrm	DT via <i>simple-framebuffer</i>
vesa-framebuffer	vesadrm	VESA

Währenddessen im Userspace (z.B. Qemu mit SeaBIOS)

- ▶ /sys/bus/platform/devices/vesa-framebuffer.0/
- ▶ /dev/dri/card0

```
1 | localhost:~ # sudo udevadm info /dev/dri/card0
2 | P: /devices/pci0000:00/0000:00:01.0/vesa-framebuffer.0/drm/card0
3 | M: card0
4 | R: 0
5 | J: c226:0
6 | U: drm
7 | T: drm_minor
8 | D: c 226:0
9 | N: dri/card0
10 | L: 0
11 | S: dri/by-path/pci-0000:00:01.0-platform-vesa-framebuffer.0-card
12 | E: DEVPATH=/devices/pci0000:00/0000:00:01.0/vesa-framebuffer.0/drm/card0
13 | E: DEVNAME=/dev/dri/card0
14 | E: DEVTYPE=drm_minor
15 | E: MAJOR=226
16 | E: MINOR=0
17 | E: SUBSYSTEM=drm
18 | E: USEC_INITIALIZED=31694248
19 | E: ID_PATH=pci-0000:00:01.0-platform-vesa-framebuffer.0
20 | E: ID_PATH_TAG=pci-0000_00_01_0-platform-vesa-framebuffer_0
21 | E: NVME_HOST_IFACE=none
22 | E: ID_FOR_SEAT=drm-pci-0000_00_01_0-platform-vesa-framebuffer_0
23 | E: DEVLINKS=/dev/dri/by-path/pci-0000:00:01.0-platform-vesa-framebuffer.0-card
24 | E: TAGS=:master-of-seat:uaccess:seat:
```

Console, Plymouth, Login, Desktop

- ▶ Reguläre Text- und Grafikausgabe läuft über den DRM-Treiber
 - ▶ Framebuffer-Console
 - ▶ Plymouth
 - ▶ Display Manager (gdm, usw.)
 - ▶ Desktop
- ▶ Software-Rendering ist meistens *gut genug*
- ▶ DRM-Treiber halten alle Grafik-Puffer im Systemspeicher
 - ▶ Beim Pageflip wird der neue Grafik-Puffer in den Framebuffer kopiert
 - ▶ Farbformat wird automatisch konvertiert
 - ▶ Puffer können als *dma-buf* mit anderen Geräten geteilt werden

Nov 22 15:30



openSUSE - Linux OS. Th x



https://www.opensuse.org



HOME

TOOLS

NEWS

CONTRIBUTE

CONFERENCES

openSUSE Communication Channels

chat.opensuse.org

Meet up with openSUSE at conferences.

View upcoming conferences

Get
software

News
Forums

Shop
Blogs



Up

DELL

Laden des nativen Treibers

- ▶ Bevor der native Treiber geladen wird, muss der generische Treiber entfernt werden
- ▶ Der native Treiber entfernt zunächst das Gerät des System-Framebuffers per Hot-Unplug
- ▶ Dadurch wird auch der DRM-Treiber entfernt
- ▶ Passiert normalerweise bevor der Display Manager startet
- ▶ Userspace schaltet auf den neuen DRM-Treiber um
- ▶ Grafik-Stack ist nun vollständig geladen

```
1 | int devm_aperture_acquire_for_platform_device(struct platform_device *pdev,  
2 |                                             resource_size_t base,  
3 |                                             resource_size_t size);  
4 |  
5 | int aperture_remove_conflicting_devices(resource_size_t base,  
6 |                                       resource_size_t size,  
7 |                                       const char *name);  
8 |  
9 | // Linux-Kernel, aperture.h
```

Features

Früher...

- ▶ Nur Fbdev-Treiber
 - ▶ Framebuffer wird direkt an Userspace exportiert
 - ▶ Keine Unterstützung für dma-buf
 - ▶ Keine Konvertierung des Farbformats möglich
 - ▶ Handover nicht trivial
 - ▶ Kein EDID
 - ▶ Nur Xorg-Userspace
 - ▶ Linux-Desktop muss X11 unterstützen
- ▶ Keine Verbindung zum PCI-Gerät
 - ▶ PCI-Hardware konnte einfach entfernt werden
 - ▶ SIGBUS

Hardware-Support

- ▶ Neue Hardware kann ohne nativen Treiber verwendet werden
- ▶ Alte Hardware kann ohne nativen Treiber verwendet werden
 - ▶ Funktioniert auch mit geringer Farbtiefe
 - ▶ DRM-Treiber emulieren XRGB8888 für Userspace

Verbesserte Fehlerbehandlung

- ▶ Native Grafiktreiber brechen sofort ab, wenn *nomodeset* gesetzt wurde
- ▶ Generischer Treiber läuft weiter
- ▶ Dient als Fallback im Fehlerfall

Live-Demo