Embedded Linux Workshop

Chosen topics of Embedded Linux

Marek Novak Lukas Janik

March 6, 2020

Acrios systems s.r.o

Schedule

- **9:00 - 10:30 Lecture**
- 10:30 - 10:50 Coffee break
- 10:50 - 12:15 Lecture
- 12:15 - 13:00 Lunch
- 13:00 - 14:30 Hands-on
- 14:30 - 14:45 Coffee break
- 14:45 - 16:15 Hands-on
- 16:15 - 17:00 - Q&A

Materials

https://cutt.ly/1teDK9V

Outline

Morning - Theoretical part

- Device Tree Essentials
- Kernel modules, device drivers
- GPIO In Linux
- Initrd
- Accessing physical memory
- IPC In Linux
- Package managers in embedded Linux

Afternoon - Practice

- Working with device tree
- First kernel module
- Customizing GPIO subsystem
- Character device driver development

Quote

Intelligence is the ability to avoid doing work, yet getting the work done.
– Linus Torvalds

Device Tree Essentials

Device Tree

Device Tree is a separate data structure for describing hardware.

Device Tree - Motivation #1

- On systems without device tree, hardware structure is “hard coded” and compiled as a part of the Linux kernel (board files)
- Each board has its own *board file* that defines and creates devices. It is a part of the kernel though.
- Re-compilation of kernel when hardware description changes. Kernel won’t boot on another platform
- Bootloader loads single image (+initrd)

Device Tree - Motivation #2

- Device tree is compiled separately, the resulting binary is called **device tree blob** or **flattened device tree** (FDT)
- One kernel can run on multiple platforms within same architecture
- In Linux kernel tree at `arch/<arch>/boot/dts`
- Custom device tree implementation also in non-GPL OS (e.g. GreenHills Integrity)

Boot with Device tree

- The device tree blob and kernel is loaded from file (SD, tftp, ...) into RAM

“‘sh fatload mmc 0 0x8800000 devtree.dtb fatload mmc 0 0x0800000 uImage

“‘ * Bootloader may modify the blob (memory fixups, chosen node) * Extract the kernel from uImage and boot it:

```
sh   bootm 0x0800000 - 0x8800000
```

- Kernel expands the blob to its internal representation called Expanded Device Tree (EDT)

Device Tree - Structure and syntax

- Tree structure with named nodes
- C-like syntax (can be preprocessed by GCC)
- Each node can have an arbitrary number of named properties
- `compatible` property is a special property that links a node to a kernel driver
- Nodes can be linked to each other by `phandles`
- Nodes can have labels that will be expanded to node's phandle or path when referenced
- OS-specific or vendor-specific properties, nodes and compatible strings contain the os/vendor name as a prefix
- Formed by *Open Firmware* project, currently maintained by *devicetree.org* community

Device Tree - Vendor prefix

```
linux,cma {
          compatible = "shared-dma-pool";
          linux,cma-default;
          ...
      };

dma-channel@80400030 {
              compatible = "xlnx,axi-dma-s2mm-channel";
              dma-channels = <0x01>;
              interrupts = <0x00 0x1d 0x04>;
              xlnx,datawidth = <0x20>;
              xlnx,device-id = <0x00>;
          };
```

Device Tree - Property types

Type	Example
empty value	`interrupt-controller;`
u32 integer	`value = 0x11223344;`
u64 integer	`value = <0x11223344 0x55667788>;`
array	`reg = <0xC1000 0x1000 0xA7000 0x1000>;`
string	`compatible = "prefix,the-string";`
string list	`clk-names = "clk_per", "clk_phy";`
phandle	`parent = <&another_node>;`

Device Tree - Example #1

```
pcf8575: gpio@20 {
    compatible = "nxp,pcf8575";
    reg = <0x20>;
    interrupt-parent = <&irqpin2>;
    interrupts = <3 0>;
    gpio-controller;
    interrupt-controller;
};
```

Device Tree - Example #2

```
/ {
    compatible = "accton,wr6202", "ralink,rt3052-soc";

    chosen {
        bootargs = "console=ttyS0,115200";
    };

    gpio-leds {
        compatible = "gpio-leds";

        wps {
            gpios = <&gpio0 14 GPIO_ACTIVE_LOW>;
        };
    };
...
```

Device Tree - Build

- Device tree compiler - **dtc**
- Preprocessing (if required) done by any GCC-like preprocessor
- The blob can be converted back to its source form (debugging)
- **device tree blob is platform independent**

Create a device tree blob from dts

```
dtc -I dts -O dtb -o mx6ulp.dtb mx6ulp.dts
```

Create a device tree source from the blob

```
dtc -I dtb -O dts -o mx6ulp.dts mx6ulp.dtb
```

Device Tree in kernel module #1

```c
// drivers/gpio/gpio-dwapb.c

static const struct of_device_id dwapb_of_match[] = {
    { .compatible = "snps,dw-apb-gpio",
      .data = (void *)0},
    { .compatible = "apm,xgene-gpio-v2",
      .data = (void *)GPIO_REG_OFFSET_V2},
    { /* Sentinel */ }
};
MODULE_DEVICE_TABLE(of, dwapb_of_match);
```

Device Tree in kernel module #2

```c
// drivers/gpio/gpio-dwapb.c

static struct platform_driver dwapb_gpio_driver = {
    .driver     = {
        .name   = "gpio-dwapb",
        .pm = &dwapb_gpio_pm_ops,
        .of_match_table = of_match_ptr(dwapb_of_match),
        .acpi_match_table = ACPI_PTR(dwapb_acpi_match),
    },
    .probe      = dwapb_gpio_probe,
    .remove     = dwapb_gpio_remove,
};

module_platform_driver(dwapb_gpio_driver);
```

Device Tree - tips

- A long include chain is a common source of errors
- Inspect changes during the `DT Lifecycle`
- preprocessing
- build
- dtb $\rightarrow$ FDT
- FDT $\rightarrow$ EDT
- Convert *dtb* back to *dts*
- Add structure-checking functionality to `probe()` function of your driver

Device Tree in /proc

- EDT structure exported to filesystem
- make sure that `CONFIG_PROC_FS` and `CONFIG_OF` is enabled
- Each curly brace in device tree is a folder in `/proc/device-tree`
- accessible by dtc: `dtc -I fs ...`

Device Tree in /proc

```
root@rp-f057cd:/proc/device-tree# ls -l
total 0
-r--r--r--  1 root root  4 Feb 28 13:07 #address-cells
drwxr-xr-x  2 root root  0 Feb 28 13:07 aliases
drwxr-xr-x 33 root root  0 Feb 28 13:07 amba
drwxr-xr-x  8 root root  0 Feb 28 13:07 amba_pl
drwxr-xr-x  2 root root  0 Feb 28 13:07 chosen
-r--r--r--  1 root root 15 Feb 28 13:07 compatible
drwxr-xr-x  4 root root  0 Feb 28 13:07 cpus
drwxr-xr-x  2 root root  0 Feb 28 13:07 fixedregulator
drwxr-xr-x  2 root root  0 Feb 28 13:07 fpga-full
drwxr-xr-x  4 root root  0 Feb 28 13:07 led-system
drwxr-xr-x  2 root root  0 Feb 28 13:07 memory
-r--r--r--  1 root root  1 Feb 28 13:07 name
drwxr-xr-x  2 root root  0 Feb 28 13:07 phy0
drwxr-xr-x  2 root root  0 Feb 28 13:07 pmu@f8891000
drwxr-xr-x  4 root root  0 Feb 28 13:07 reserved-memory
-r--r--r--  1 root root  4 Feb 28 13:07 #size-cells
drwxr-xr-x  2 root root  0 Feb 28 13:07 __symbols__
root@rp-f057cd:/proc/device-tree# cat compatible
xlnx,zynq-7000
```

Device Tree tools

dt_to_config
`/scripts/dtc/dt_to_config`

- Check device tree source against kernel configuration
- Find nodes that do not have drivers present or set for build
- `dt_to_config <path_to_dts_or_dtb>`

dtdiff
`/scripts/dtc/dtxdiff`

- Compare two versions of DeviceTree (any format)
- Use `dtxdiff <dts file> <dtb file>`

Runtime debugging

`Documentation/dynamic-debug-howto.txt`

- Enable debug for a specific file/line/function/module
- Enable kernel config `CONFIG_DYNAMIC_DEBUG`
- At boot-time - Add a query to kernel cmdline

```
dyndbg="func bus_add_driver +p"   dyndbg="func
really_probe +p"
```

- At run-time - via debugfs

```
bash   echo "func bus_add_driver +p" >
/sys/kernel/debug/dynamic_debug/control   echo "func
really_probe +p" >
/sys/kernel/debug/dynamic_debug/control
```

Runtime debugging

```
$ dmesg
bus: 'usb': really_probe: probing driver usb with device 4-
bus: 'usb': really_probe: bound device 4-2 to driver usb
bus: 'usb': add driver r8152
bus: 'usb': really_probe: probing driver r8152 with device
r8152: probe of 4-2:2.0 rejects match -19
usbcore: registered new interface driver r8152
bus: 'usb': really_probe: probing driver r8152 with device
bus: 'usb': add driver cdc_ether
usbcore: registered new interface driver cdc_ether
usb 4-2: reset SuperSpeed Gen 1 USB device number 7 using x
r8152 4-2:1.0 eth0: v1.10.10
bus: 'usb': really_probe: bound device 4-2:1.0 to driver r8
r8152 4-2:1.0 enp0s20u2: renamed from eth0
```

Device Tree - library

- Device Tree for dummies
- Device Tree Reference
- devicetree.org
- Debugging devtree #1
- Debugging devtree #2

Kernel modules

What is a kernel module

Each piece of code that can be added to the kernel at runtime is called a module.

What is a kernel module

- Object code that is not linked into complete executable
- Allows extending the kernel functionality without a need to reboot the system - easy driver development.
- Same code can be built into the kernel (available from early boot), or as a module.
- Modules are stored as separate files in a filesystem. It needs to be mounted before modules can be loaded.
- Thanks to loadable modules, Linux kernel binary can be very small yet universal and multi-platform.

Module vs applications

Applications

- Event-driven or procedural
- Linked against external libraries
- Running in non-privileged mode
- Error *may not* cause system crash
- May be reentrant

Modules

- Strictly event-driven
- Can use only functions exported from kernel.
- Running in privileged mode
- Error may cause system crash
- Must be reentrant
- Can export symbols to be used by other loadable module

Simple kernel module

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init first_module_init(void)
{
    printk(KERN_INFO "Our first module loaded!\n");
    return 0;
}

static void __exit first_module_cleanup(void)
{
    printk(KERN_INFO "Our first module is cleaning up\n");
}

/* Module entry and exit points */
module_init(first_module_init);
module_exit(first_module_cleanup);

/* Optional: Specify Meta information */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Lukas Janik");
MODULE_DESCRIPTION("Our first module");
```

Kernel module licensing

- Kernel provided under terms of GPL-2.0
- Licensing interface between userspace and kernel are syscalls
- Modules should be tagged by `MODULE_LICENSE` macro which specifies whether the module shall be linked with other modules
- Userspace headers are exception since they have to be included both to GPL kernel and (possibly) non-GPL user programs

```
/* SPDX-License-Identifier: GPL-2.0 WITH
   Linux-syscall-note */
```

Userspace vs kernelspace

- **Kernelspace** - Linux kernel monolith and modules. Runs in privileged mode.
- **Userspace** - Other applications, running in non-privileged mode.
- Userspace applications perform privileged operations indirectly via kernel (syscalls)[1]
- Stable interface (only new syscalls added)
- Part of std. C library

`open, read, write, close, fsync, access, bind, chown, chroot, ...`

[1] To see all syscalls, run man syscalls

Kernel composition

Essential parts of Linux kernel

- Device drivers
- Filesystem drivers
- Networking drivers
- Process management

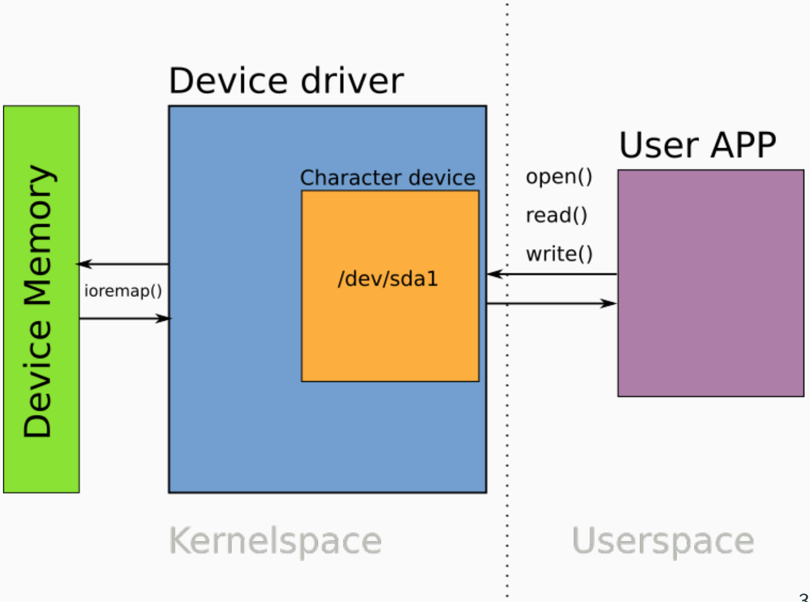
Device driver classes

- Character devices
- Block devices
- Network interfaces

Device model

- **Device** is an "object". It provides some properties (platform data) and resources (IRQs, registers).
- **Driver** is a set of methods. It defines how the kernel should interact with the device.
- **Bus** is a common parent of *devices* and *drivers*. It implements device operations, either itself or by binding devices to drivers
- All devices are connected to some bus
- Bus can be physical (USB, PCI, etc.) or virtual (platform)
- Devices can be connected to more than one bus (e.g. USB controller)

Device model



Device discovery

Discoverable bus devices

- USB, PCI, FireWire, . . .
- Created during discovery process by the bus driver

Non-discoverable devices

- I2C, SPI, . . .
- Created from device tree or during machine init

Platform devices

- Platform device is a device that is inherently not discoverable - e.g. I^2C devices, SoC controllers, ...
- Platform devices are bound to platform driver by matching names
- Instantiated by code or from *Device tree*
- Should be registered early

Case study: imx31 Lite #1

```
// arch/arm/mach-imx/mach-mx31lite.c
static unsigned int mx31lite_pins[] = {
    /* UART1 */
    MX31_PIN_CTS1__CTS1,
    MX31_PIN_RTS1__RTS1,
    MX31_PIN_TXD1__TXD1,
    MX31_PIN_RXD1__RXD1,
    /* SPI 0 */
    MX31_PIN_CSPI1_SCLK__SCLK,
    MX31_PIN_CSPI1_MOSI__MOSI,
    MX31_PIN_CSPI1_MISO__MISO,
    MX31_PIN_CSPI1_SPI_RDY__SPI_RDY,
    MX31_PIN_CSPI1_SS0__SS0,
    MX31_PIN_CSPI1_SS1__SS1,
    ...
};
```

Case study: imx31 Lite #2

```
/* UART */
static const struct imxuart_platform_data
    uart_pdata __initconst = {
        .flags = IMXUART_HAVE_RTSCTS,
};
/* SPI */
static const struct spi_imx_master
    spi0_pdata __initconst = {
        .chipselect = spi0_internal_chipselect,
        .num_chipselect = ARRAY_SIZE(spi0_internal_chipselect),
};
/* NAND */
static const struct mxc_nand_platform_data
    mx31lite_nand_board_info __initconst  = {
    .width = 1,
    .hw_ecc = 1,
};
```

Case study: imx31 Lite #3

```
static struct smsc911x_platform_config smsc911x_config = {
    .irq_polarity   = SMSC911X_IRQ_POLARITY_ACTIVE_LOW,
    .irq_type    = SMSC911X_IRQ_TYPE_PUSH_PULL,
    .flags       = SMSC911X_USE_16BIT,
};

static struct resource smsc911x_resources[] = {
    {
        .start      = MX31_CS4_BASE_ADDR,
        .end        = MX31_CS4_BASE_ADDR + 0x100,
        .flags      = IORESOURCE_MEM,
    }, {
        /* irq number is run-time assigned */
        .flags      = IORESOURCE_IRQ,
    },
};
```

Case study: imx31 Lite #4

```
static struct platform_device smsc911x_device = {
    .name       = "smsc911x",
    .id     = -1,
    .num_resources  = ARRAY_SIZE(smsc911x_resources),
    .resource   = smsc911x_resources,
    .dev        = {
        .platform_data = &smsc911x_config,
    },
};

static struct platform_device physmap_flash_device = {
    .name   = "physmap-flash",
    .id     = 0,
    .dev    = {
        .platform_data  = &nor_flash_data,
    },
    .resource = &nor_flash_resource,
    .num_resources = 1,
};
```

Case study: imx31 Lite #5

```
// arch/arm/mach-imx/mach-mx31lite.c
static void __init mx31lite_init(void)
{
    imx31_soc_init();
    mxc_iomux_setup_multiple_pins(mx31lite_pins,
        ARRAY_SIZE(mx31lite_pins), "mx31lite");
    imx31_add_imx_uart0(&uart_pdata);
    imx31_add_spi_imx0(&spi0_pdata);
    /* NOR and NAND flash */
    platform_device_register(&physmap_flash_device);
    imx31_add_mxc_nand(&mx31lite_nand_board_info);
    imx31_add_spi_imx1(&spi1_pdata);

    regulator_register_fixed(0, dummy_supplies,
        ARRAY_SIZE(dummy_supplies));
}
```

Case study: imx31 Lite #6

```
// drivers/net/ethernet/smsc/smsc911x.c
static int smsc911x_drv_probe(struct platform_device *pdev)
{
    ...
    struct smsc911x_platform_config *config = dev_get_platdata(&pdev->dev);

    res = platform_get_resource_byname(pdev, IORESOURCE_MEM,
                                       "smsc911x-memory");
    if (!res)
        res = platform_get_resource(pdev, IORESOURCE_MEM, 0);

    irq = platform_get_irq(pdev, 0);
    ...
```

Binding device to a driver

Binding a device to a driver is done automatically by the driver core when:

- Driver is registered and the device already exists - `driver_attach()`
- Device is created and the driver is already registered - `device_attach()`

Manual unbinding

- rmmod'ing the platform driver module will unbind all its devices
- Using sysfs

Unbinding driver using sysfs

```
$ ls -l /sys/bus/usb/drivers/usb
total 0
lrwxrwxrwx 1 root root    0 Feb  2 09:23 1-1 ->
    ../../../../devices/pci0000:00/0000:00:1a.0/usb1/1-1
lrwxrwxrwx 1 root root    0 Feb  2 09:23 1-1.2 ->
    ../../../../devices/pci0000:00/0000:00:1a.0/usb1/1-1/1-
lrwxrwxrwx 1 root root    0 Feb  2 09:23 1-1.6 ->
    ../../../../devices/pci0000:00/0000:00:1a.0/usb1/1-1/1-
--w------- 1 root root 4096 Feb  2 09:23 bind
--w------- 1 root root 4096 Feb  2 09:23 uevent
--w------- 1 root root 4096 Feb  2 09:23 unbind

$echo -n "1-1.6" > /sys/bus/usb/drivers/usb/unbind
```

Device files

- Each device may expose itself to userspace via special device files, stored in /dev
- Each device file is bound to driver via **major** and **minor** number
- Major number - unique number, determines device type
- Minor number - instances of same device between each other or sub-type
- “b” vs “c” in first column of ls -l for block devices and character devices, respectively.
  Major and minor numbers at place where file size normally appears
- For a full list of static device number allocation, see `Documentation/devices.txt`

ls -l /dev

```
ls -l /dev
drwxr-xr-x  2 root  root            0 Jan 25 06:45 pts
crw-rw-rw-  1 root  root       1,   8 Jan 25 06:45 random
lrwxrwxrwx  1 root  root            4 Jan 25 06:45 rtc -> rtc0
crw-------  1 root  root     250,   0 Jan 25 06:45 rtc0
brw-rw----  1 root  disk       8,   0 Jan 25 06:45 sda
brw-rw----  1 root  disk       8,   1 Jan 25 06:50 sda1
crw-rw-rw-  1 root  tty        5,   0 Jan 28 20:34 tty
crw--w----  1 root  tty        4,   0 Jan 25 06:45 tty0
crw--w----  1 root  tty        4,   1 Jan 28 19:47 tty1
crw--w----  1 root  tty        4,  10 Jan 25 06:45 tty10
```

```
Memory devices
  1 = /dev/mem        Physical memory access
  2 = /dev/kmem       Kernel virtual memory access
  3 = /dev/null       Null device
  4 = /dev/port       I/O port access
  5 = /dev/zero       Null byte source
  6 = /dev/core       OBSOLETE - replaced by /proc/kcore
  7 = /dev/full       Returns ENOSPC on write
  8 = /dev/random     Nondeterministic random number gen.
  9 = /dev/urandom    Faster, less secure random number gen.
 10 = /dev/aio        Asynchronous I/O notification interface
 11 = /dev/kmsg       Writes to this come out as printk's, re
            export the buffered printk records.
 12 = /dev/oldmem     OBSOLETE - replaced by /proc/vmcore
```

The story of /dev

- At the beginning, device files were created statically ~ <2.4
- Lot of files, nodes also for devices that are not present (thousands)
- Running out of device numbers
- Devfs ~ >2.4
- Dev files only for devices that are present
- Non-standard, not persistent device names
- udev ~ >2.5
- userspace utility
- dev files created upon request from kernel
- persistent dev file naming configurable by user
- Devtmpfs ~ 2.6 - now
- Early tmpfs populated with device nodes
- No userspace required to have working /dev -> Faster boot
- udev can run on top of it as soon as userspace starts, utilizing

Character devices

- Most common type of device
- Device acts as a "Stream of characters"
- Examples: `serial port, I2C, SPI, /dev/random, ...`
- Must instantiate a `cdev` structure
- Must implement `file_operations`

struct file_operations

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *,...
    ssize_t (*write) (struct file *, const char __user *,...
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, ...
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, ...
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ...
};
```

Block devices

- Device accessed by blocks
- Most often storage devices
- Examples: `mmc, hard disks...`
- Must instantiate a `gendisk` structure
- Must implement `block_device_operations`

struct block_device_operations

```
struct block_device_operations {
    int (*open) (struct block_device *, fmode_t);
    void (*release) (struct gendisk *, fmode_t);
    int (*rw_page)(struct block_device *, sector_t, ...
    int (*ioctl) (struct block_device *, fmode_t, ...
    int (*compat_ioctl) (struct block_device *, ...
    long (*direct_access)(struct block_device *, ...
            long);
    unsigned int (*check_events) (struct gendisk *disk,
                  unsigned int clearing);
    int (*media_changed) (struct gendisk *);
    void (*unlock_native_capacity) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    int (*getgeo)(struct block_device *, struct hd_geometry *);
    void (*swap_slot_free_notify) ...
};
```

ioctl()

- Functions defined in `file_operations` are not sufficient for all purposes
- ioctl() allows to pass custom data from userspace to kernel
- Used e.g. in serial driver to set line parameters (baud rate, etc.)
- Command (`cmd` parameter) should be unique, vid. Documentation/ioctl-number.txt

Kernel:

```
int (*ioctl) (struct inode *inode, struct file *fl,
    unsigned int cmd, unsigned long data);
```

Userspace:

```
int ioctl(int fd, int cmd, ...);
```

Procfs, sysfs

procfs - /proc

- pseudo FS that provides information about kernel processes and system information.
- Older one
- Does not have a strictly defined structure
- Allows all functions from `file_operations`

sysfs - /sys

- Another pseudo FS
- Since 2.6.
- Structured, uniform way to expose system information
- Current way of exposing driver information and/or setting points
- Restricted file operations

procfs examples

- `/proc/modules` - list of loaded modules (lsmod)
- `/proc/uptime` - system uptime (uptime)
- `/proc/version` - kernel version (uname)
- `/proc/cpuinfo` - CPU information
- `/proc/meminfo` - memory information
- `/proc/config.gz` - kernel .config used to build running kernel
- `/proc/<num>` - information about process with PID <num>

sysfs examples

- `/sys/dev` - system devices (character/block)
- `/sys/bus` - system buses
- `/sys/class` - system device classes registered to kernel
- `/sys/module` - system modules (also builtin)
- `/sys/firmware` - system firmware objects
- **sysctl** - utility to manipulate `/sys` files

Kernel build system - Kbuild

Four essential building blocks:

1. `config symbols` - for conditional build of the code or to decide `y/m/n`
2. `Kconfig files` - define meta information of `config symbols` and available options, used by UI tools (menuconfig, xconfig, gconfig)
3. `.config file` - a database of selected `config symbols`
4. `makefiles` - a common GNU makefiles defining the build process itself

Example - linux-xlnx/net/Kconfig

```
menu "Character devices"

source "drivers/tty/Kconfig"

config DEVMEM
    bool "/dev/mem virtual device support"
    default y
    help
      Say Y here if you want to support the /dev/mem device
      The /dev/mem device is used to access areas of physic
      memory.
      When in doubt, say "Y".

config DEVKMEM
    bool "/dev/kmem virtual device support"
```

Makefile - linux-xlnx/net/Makefile

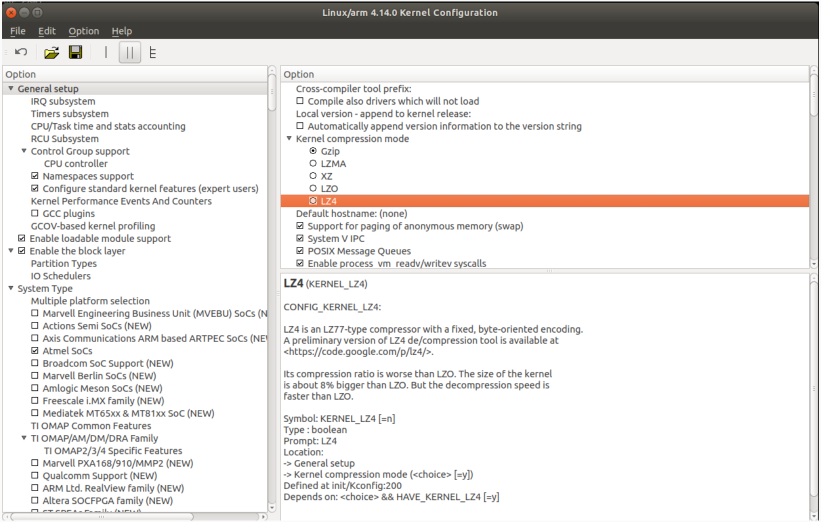
```
#
# Makefile for the kernel character device drivers.
#
obj-y                           += mem.o random.o
obj-$(CONFIG_TTY_PRINTK)        += ttyprintk.o
obj-y                           += misc.o
obj-$(CONFIG_ATARI_DSP56K)      += dsp56k.o
obj-$(CONFIG_VIRTIO_CONSOLE)    += virtio_console.o
obj-$(CONFIG_RAW_DRIVER)        += raw.o
obj-$(CONFIG_SGI_SNSC)          += snsc.o snsc_event.o
obj-$(CONFIG_MSPEC)             += mspec.o
obj-$(CONFIG_MMTIMER)           += mmtimer.o
obj-$(CONFIG_UV_MMTIMER)        += uv_mmtimer.o
obj-$(CONFIG_IBM_BSR)           += bsr.o
```

Menuconfig

```
.config - Linux/arm 4.9.0 Kernel Configuration
> Device Drivers > Character devices ─────────────────────────────────
                              Character devices
 Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters are hotkeys.
 Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.
 Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                          [*] Enable TTY
                          [*]   Virtual terminal
                          [*]     Enable character translations in console
                          [*]     Support for console on virtual terminal
                          -*-     Support for binding and unbinding console drivers
                          [*]   Unix98 PTY support
                          [ ]   Legacy (BSD) PTY support
                          [ ]   Non-standard serial port support
                          < >   HSDPA Broadband Wireless Data Card - Globe Trotter
                          < >   GSM MUX line discipline support (EXPERIMENTAL)
                          < >   Trace data sink for MIPI P1149.7 cJTAG standard
                          [*] /dev/mem virtual device support
                          [ ] /dev/kmem virtual device support
                              Serial drivers  --->
                          < > TTY driver to output user messages via printk
                          [ ] ARM JTAG DCC console
                          < > Virtio console
                          < > IPMI top-level message handler  ----
                          < > Hardware Random Number Generator Core support  ----
                          <*> Xilinx Device Configuration
                          < > Siemens R3964 line discipline
                          < > Applicom intelligent fieldbus card support
                          < > RAW driver (/dev/raw/rawN)
                          < > TPM Hardware Support  ----
                          < > Xillybus generic FPGA interface

                <Select>    < Exit >    < Help >    < Save >    < Load >
```

Menuconfig - Device drivers/Character devices

xconfig


Linux/arm 4.14.0 Kernel Configuration
File Edit Option Help
Option
General setup
IRQ subsystem
Timers subsystem
CPU/Task time and stats accounting
RCU Subsystem
Control Group support
CPU controller
Namespaces support
Configure standard kernel features (expert users)
Kernel Performance Events And Counters
GCC plugins
GCOV-based kernel profiling
Enable loadable module support
Enable the block layer
Partition Types
IO Schedulers
System Type
Multiple platform selection
Marvell Engineering Business Unit (MVEBU) SoCs (N
Actions Semi SoCs (NEW)
Axis Communications ARM based ARTPEC SoCs (NE
Atmel SoCs
Broadcom SoC Support (NEW)
Marvell Berlin SoCs (NEW)
Amlogic Meson SoCs (NEW)
Freescale i.MX family (NEW)
Mediatek MT65xx & MT81xx SoC (NEW)
TI OMAP Common Features
TI OMAP/AM/DM/DRA Family
TI OMAP2/3/4 Specific Features
Marvell PXA168/910/MMP2 (NEW)
Qualcomm Support (NEW)
ARM Ltd. RealView family (NEW)
Altera SOCFPGA family (NEW)
Option
Cross-compiler tool prefix:
Compile also drivers which will not load
Local version - append to kernel release:
Automatically append version information to the version string
Kernel compression mode
Gzip
LZMA
XZ
LZO
LZ4
Default hostname: (none)
Support for paging of anonymous memory (swap)
System V IPC
POSIX Message Queues
Enable process_vm_readv/writev syscalls
LZ4 (KERNEL_LZ4)
CONFIG_KERNEL_LZ4:
LZ4 is an LZ77-type compressor with a fixed, byte-oriented encoding.
A preliminary version of LZ4 de/compression tool is available at
<https://code.google.com/p/lz4/>.
Its compression ratio is worse than LZO. The size of the kernel
is about 8% bigger than LZO. But the decompression speed is
faster than LZO.
Symbol: KERNEL_LZ4 [=n]
Type : boolean
Prompt: LZ4
Location:
-> General setup
-> Kernel compression mode (<choice> [=y])
Defined at init/Kconfig:200
Depends on: <choice> && HAVE_KERNEL_LZ4 [=y]

Building the kernel

- One needs to select the target architecture via env. variable ARCH. It will be used for configuration and build.
- When left unset, kernel will be built for host's architecture
- To see all possible architectures, check `boot/arch` folder
- When cross-compiling, set the CROSS_COMPILE variable with the prefix of the toolchain
- It is advised to export these variables to the shell. Otherwise you would need to set them for each command separately
- When compiling for host machine, leave both variables unset

```
export ARCH=arm
export CROSS_COMPILE=arm-none-eabihf-
```

Building the kernel - select configuration

Use pre-defined configuration for your setup

- Default configs for most scenarios
- Check `arch/<YourArch>/configs` for configurations available for your architecture
- Load selected configuration

```sh
make xilinx_zynq_defconfig
```

Create custom configuration

- Use one of available tools

```sh
make menuconfig # or
make xconfig # or
make gconfig
```

Building the kernel - build

- Use the `-j<corenum>` parameter to run on multiple threads to speedup the process
- Run the kernel build `sh make -j4`
- The build will produce:
- `vmlinux` - the uncompressed ELF kernel image
- `arch/<YourArch>/boot/?Image` - the compressed kernel image that can boot (bzImage, zImage)
- `arch/<YourArch>/boot/dts/*.dtb` - the device tree binaries

Building the kernel - installation

- run `make install` to install the kernel to host system
- usually not used in embedded development
- requires root privileges
- copies the kernel image, used configuration and System.map
- run `make modules_install` to install built modules to `/lib/modules`
- set `INSTALL_MOD_PATH` variable to specify path where modules will be copied

Building modules out-of-tree

- You can build kernel modules at any location
- The only requirement is the presence of kernel headers
- Desktop OS Distributions often provide a package `sh    apt install linux-headers-$(uname -r)`
- Minimal makefile to compile kernel module `first_module.c`:
  “‘makefile obj-m += first_module.o
  KERNEL_TREE=/lib/modules/$(shell uname -r)/build

all: make -C $(KERNEL_{T}REE)M =$(PWD) modules “‘

Manual loading and unloading of a module

- **insmod** - Loads a module from a file. Does not resolve dependencies
  `insmod first_module.ko [args]`
- **modprobe** - Loads a module from `/lib/modules/<x.y.z-arch>`. Loads dependencies first.
  `modprobe spi-gpio`
- **rmmod** or **modprobe -r** - Unloads a module from kernel.
  `rmmod first_module`

Module parameters

`linux/moduleparam.h`

- Module can take arbitrary number of named parameters
- Each parameter has assigned permissions
- S_IRUSR, S_IRUGO, ...
- Many supported types: `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp`, `bool`, `invbool`
- Use `modinfo` utility to show available module parameters
- Check `/usr/modules/<module>/parameters` to see all parameters, and get/set their value

```
static int loops = 0;
module_param(loops, int, S_IRUGO);
MODULE_PARM_DESC(loops, "A number of loops");

static char *text = NULL;
module_param(text, charp, S_IRUGO);
MODULE_PARM_DESC(text, "This is a char pointer (string)");
```

Automatic loading of a module

- When a device is inserted - *udev*
- Kernel sends *uevent* to udev, udev runs appropriate actions (modprobe, create /dev node)
- "udev rules" in /etc/udev/rules.d (allow access to device to normal users)
- `udevadm monitor`
- During boot by specifying in `/etc/modprobe.conf`

How to find out that there is a missing driver

- No special mechanism
- Check `dmesg`
- If applicable, check `lsusb` or `lspci`
- Find out whether required module is loaded
  `lsmod`
- Check if the module is built into kernel
  `cat /lib/modules/$(uname -r)/modules.builtin`

Library i

Linux source - Elixir
Kernel device drivers
Linux build system
Platform device API
A fresh look at the kernel's device model

GPIO in Linux

GPIO Kernel API

- Legacy integer-based API [`gpio`]
  `linux/gpio.h`
- Current descriptor-based API [`gpiod`]
  `linux/gpio/consumer.h`
- GPIO should be obtained (reserved) before used
- GPIO can be exported to userspace

Kernel example

```
// From Documentation/gpio/board.txt
foo_device {
        compatible = "acme,foo";
        ...
        led-gpios = <&gpio 15 GPIO_ACTIVE_HIGH>, /* red */
                <&gpio 16 GPIO_ACTIVE_HIGH>, /* green */
                <&gpio 17 GPIO_ACTIVE_HIGH>; /* blue */

        power-gpios = <&gpio 1 GPIO_ACTIVE_LOW>;
    };
```

Kernel example

```
// From Documentation/gpio/board.txt
struct gpio_desc *red, *green, *blue, *power;

red = gpiod_get_index(dev, "led", 0, GPIOD_OUT_HIGH);
green = gpiod_get_index(dev, "led", 1, GPIOD_OUT_HIGH);
blue = gpiod_get_index(dev, "led", 2, GPIOD_OUT_HIGH);
power = gpiod_get(dev, "power", GPIOD_OUT_HIGH);
```

Access functions

```
int  gpiod_direction_input(struct gpio_desc *desc)
int  gpiod_direction_output(struct gpio_desc *desc, int value)
```

- Two variants of set/get functions:
- Functions that are spinlock-safe (controller is memory mapped)
- Functions that can sleep (controller connected via external bus - I2C, etc.)

```
int  gpiod_get_value(const sruct gpio_desc *desc);
void gpiod_set_value(struct gpio_desc *desc, int value);
int  gpiod_get_value_cansleep(const struct gpio_desc *desc);
void gpiod_set_value_cansleep(struct gpio_desc *desc, int value);
```

Userspace - Sysfs API

- Legacy userspace API in `/sys/class/gpio`
- Currently deprecated
- Remains exported when application crashes
- Multiple file descriptors, multiple syscalls

```
# Export GPIO pin 15
echo 15 > /sys/class/gpio/export
# Set as output
echo out > /sys/class/gpio/gpio15/direction
# Set GPIO 15 to "1"
echo 1 > /sys/class/gpio/gpio15/value
```

Userspace - Character device API

- Merged in 4.8
- One device per gpiochip
- Access via `/dev/gpiochip0` etc.
- Allows multiple operations at single syscall
- API defined in `include/linux/gpio.h`
- **libgpiod** - C library for handling new GPIO userspace API
- Userspace tools for GPIO handling provided
- https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git

libgpiod - example

```c
struct gpiod_chip *chip;
struct gpiod_line *line;
// Open GPIO chip
chip = gpiod_chip_open("/dev/gpiochip0");
// Get line (pin) from the GPIO chip
line = gpiod_chip_get_line(chip, offset);
// Request (reserve) the line and set as output
gpiod_line_request_output(line, "consumer", 0);
// Set value to the line
gpiod_line_set_value(line, 0);
// Release the line
gpiod_line_release(line);
```

libgpiod - Tools

- **gpiodetect** - list all gpiochips present on the system, their names, labels and number of GPIO lines
- **gpioinfo** - list all lines of specified gpiochips, their names, consumers, direction, active state and additional flags
- **gpioget** - read values of specified GPIO lines
- **gpioset** - set values of specified GPIO lines, potentially keep the lines exported and wait until timeout, user input or signal
- **gpiofind** - find the gpiochip name and line offset given the line name
- **gpiomon** - wait for events on GPIO lines, specify which events to watch, how many events to process before exiting or if the events should be reported to the console

Library

GPIO in the kernel: an introduction

GPIO Sysfs Interface for Userspace

New GPIO interface for Userspace

Documentation/gpio/board.txt

Initrd

System boot

- Bootloader loads the kernel image (and devtree) to RAM
- Kernel is self-extracted and run
- Kernel mounts temporary root file system to /
- Modules required for mounting the final rootfs are loaded
- Actions required for mounting the final are performed (user is asked for password to LUKS)
- The / is switched to new location and initrd is dropped
- Kernel starts init (sysv, systemd) with PID 1
- Init starts all system services

What is Initrd?

- Initial RAM Disk provided during boot
- Contents temporarily mounted to /
- Purpose: Load modules and utilities required to mount the rootfs
- FS modules (LVM, btrfs, ...)
- Encryption utilities (dm-crypt)
- network utilities for NFS (dhclient)
- After mounting the rootfs, whole set of modules is available. The initrd's memory is released.

Initrd vs Initramfs

Initrd

- The older one
- Regular `ramdev` block device
- Requires underlying filesystem → has to be compiled in kernel (e.g. ext2)
- dentry, inode for each opened file has to be allocated also in kernel → a bit higher memory consumption, complexity → a bit slower than initramfs

Initramfs

- Since 2.5.x
- Uses `tmpfs`
- Does not need underlying filesystem
- `tmpfs` support is in kernel, no need for additional modules
- Often called just `initrd`

Generating initrd

- Usually each OS distribution provides own script
  - Ubuntu: `update-initramfs`, `mkinitramfs`
  - Archlinux: `mkinitcpio`
    - `mkinitcpio -c /etc/mkinitcpio-custom.conf -g /boot/linux-custom.img`
- Image itself is usally generated based on a config (receipt)

/etc/mkinitcpio.conf

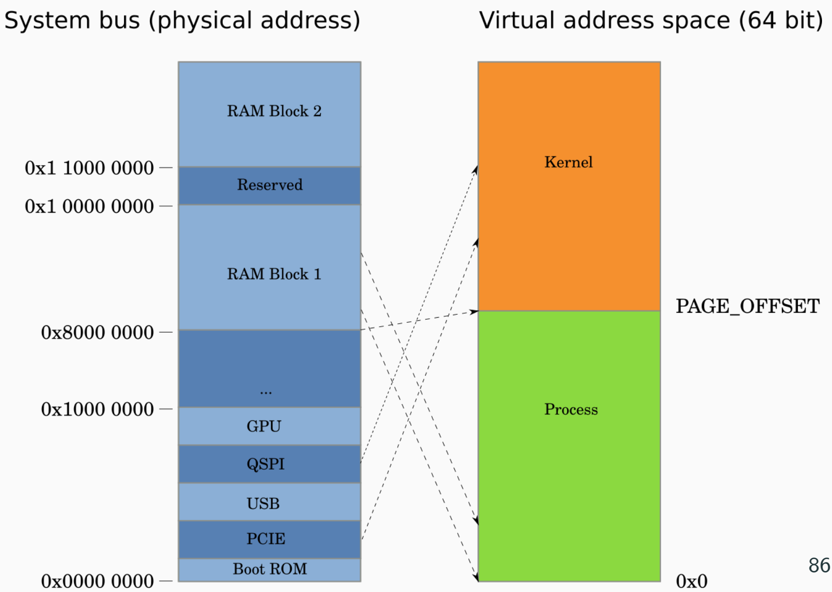
```
MODULES=()
BINARIES=()
FILES=()

# HOOKS
# This is the most important setting in this file.  The H..
# modules and scripts added to the image, and what happen..
# Order is important, and it is recommended that you do n..
# order in which HOOKS are added.  Run 'mkinitcpio -H <ho..
# help on a given hook.
HOOKS=(base udev autodetect modconf block filesystems key..

COMPRESSION="gzip"
```

Accessing physical memory

Physical vs virtual memory

- **Physical memory** - defined by hardware, can be different for each device on the memory bus
- **Virtual memory** - as seen from software / behind MMU
- Translation (mapping) done by memory management unit (MMU)
- Smallest unit - **Page**, usually 4 kB
- Single **page frame** (physical) may be mapped multiple times (to multiple virtual pages)
- The relation between physical and virtual addresses is stored in a hierarchy of **Page tables**

Memory mapping



Motivation

- Each user process resides in its own address space
- One process can't corrupt kernel memory
- One process can't corrupt another process's memory
- Each process has different virtual - physical mapping
- More processes can map same chunk of RAM (e.g. for RPC)
- User process' memory is assigned by kernel and controlled by MMU
- Kernel memory is permanently mapped (at PAGE_OFFSET)
- Performance
- Handling interrupts, exceptions, syscalls, . . .

Kernel virtual memory

- `void *kmalloc(size_t size, gfp_t flags);`
  allocate normal kernel contiguous memory
- `void *vmalloc(unsigned long size);`
  allocate non-contiguous memory (in separate addres space). Usually for large allocations. Allocate entire pages.
- `void __iomem *ioremap(resource_size_t res_cookie, size_t size);`
  Map device memory to kernel
- `void *kmap(struct page *page);`
  Permanently map arbitrary physical page to kernel. Use `kunmap()` as soon as not required.

ioremap() caching on ARM

```
// arch/arm/include/asm/io.h
/*
 * Function     Memory type Cacheability    Cache hint
 * ioremap()          Device        n/a     n/a
 * ioremap_nocache()  Device        n/a     n/a
 * ioremap_cache()    Normal        Writeback   Read allocate
 * ioremap_wc()       Normal        Non-cacheable   n/a
 * ioremap_wt()       Normal        Non-cacheable   n/a
 *
 * All device mappings have the following properties:
 * - no access speculation
 * - no repetition (eg, on return from an exception)
 * - number, order and size of accesses are maintained
 * - unaligned accesses are "unpredictable"
 * - writes may be delayed before they hit the endpoint device
 */
```

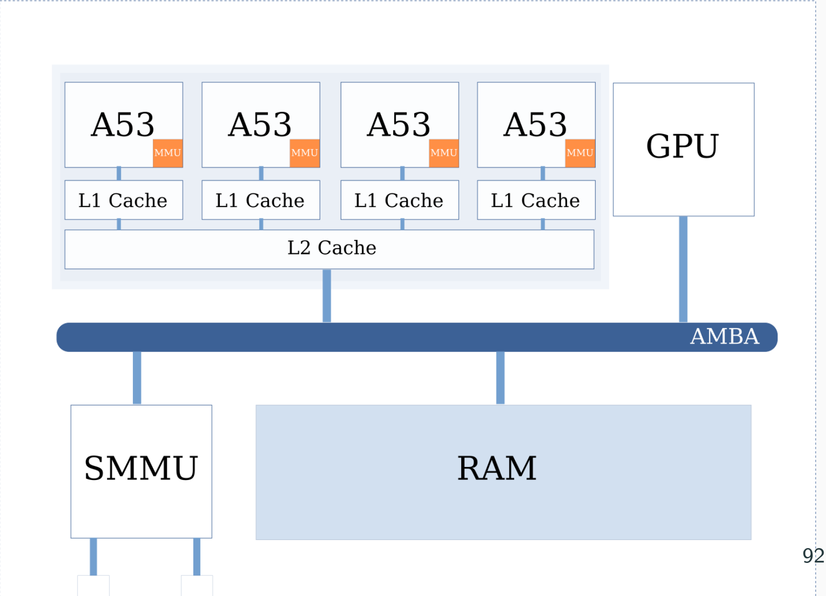
/dev/mem

- Driver in `drivers/char/mem.c`
- Character device for interfacing userspace applications with physical memory
- Available operations:
- read() - read from physical memory
- write() - write to physical memory
- mmap() - map physical range to userspace
- Some limitations may apply (CONFIG_STRICT_DEVMEM)

Memory-mapped IO

```c
// drivers/gpio/gpio-zynq.c
/* Fetch the memory resource from the device */
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
/* Map registers to kernel */
gpio->base_addr = devm_ioremap_resource(&pdev->dev, res);
if (IS_ERR(gpio->base_addr))
  return PTR_ERR(gpio->base_addr);
// ...
/* set the GPIO pin as output */
reg = readl_relaxed(gpio->base_addr + ZYNQ_GPIO_DIRM_OFFSET(bank_num));
reg |= BIT(bank_pin_num);
writel_relaxed(reg, gpio->base_addr + ZYNQ_GPIO_DIRM_OFFSET(bank_num));
```

Memory caching on ARM



Process virtual memory

- Show memory mapping of process with PID `<pid>`
  `/proc/<pid>/maps`

Library

Memory management in Linux
kvmalloc()
KAISER: hiding the kernel from user space

Quote

The memory management on the PowerPC can be used to frighten small children.
– Linus Torvalds

Inter-Process-Communication (IPC) In Linux

IPC - Overview

- Allows to create more complex systems
- Multiple processes handling a portion of the system communicating with each other
- List of available mechanisms: **Signals**, Anonymous Pipes, **Named Pipes or FIFOs**, SysV Message Queues, POSIX Message Queues, SysV Shared memory, POSIX Shared memory, SysV semaphores, POSIX semaphores, FUTEX locks, **File-backed and anonymous shared memory using mmap**, UNIX Domain Sockets, Netlink Sockets, **Network Sockets**, Inotify mechanisms, FUSE subsystem, D-Bus subsystem, micro bus (OpenWRT)
- Availability depends on actual distribution
- Link: Linux IPC Mechanisms

IPC - Signals

- One way asynchronous notifications
- Sent by kernel or a process to another process
- Typically alerts on an event - CTRL+C pressing, Stack fault, User defined signal...
- Signals can be: raised, caught, acted upon, ignored
- Handles signals cause execution of a signal handler function
- Acts like an interrupt - once execution ends, main context continues
- Link: Linux Process and Signals

IPC - Signals : Example

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void my_signal_interrupt(int sig)
{
  printf("I got signal %d\n", sig);
  (void) signal(SIGINT, SIG_DFL);
}

int main()
{
  (void) signal(SIGINT,my_signal_interrupt);

  while(1) {
      printf("Waiting for interruption...\n");
      sleep(1);
  }
}
```

IPC - Named Pipes

- Acts as a FIFO
- Created by `mkfifo test`, where ‘test’ is the name of the pipe
- One process can write to the file
- Other process opens the file and reads data until EOF is reached

IPC - Named Pipes: Writer

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
void main(void)
{
    int f;

    printf("Wait until a process opens FIFO for reading...\n");
    f = open("test", O_WRONLY); //open FIFO called "test"
    printf("Write the message...\n");
    write(f,"hello",5); //write 5 bytes to FIFO
    close(f);
    printf("Message Delivered!\n");
    return;
}
```

IPC - Named Pipes: Reader

```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
void main(void)
{
    char buffer[32]; //some buffer
    int count;
    int f;

    printf("Wait for a process to open FIFO for writing...\n");
    f = open("test", O_RDONLY); //open FIFO called "test"
    printf("Read the message...\n");
    count = read(f, buffer, 32); //read up to 32 bytes
    close(f);
    buffer[count] = '\0';
    printf("Received: '%s'\n", buffer);
    return;
}
```

IPC - Shared Memory

- Allows for multiple writers and multiple readers
- Explained on a practical example - logger process
- Link: Interprocess Communication Using Posix Shared Memory In Linux
- Link: Memory Mapped Files and Shared Memory

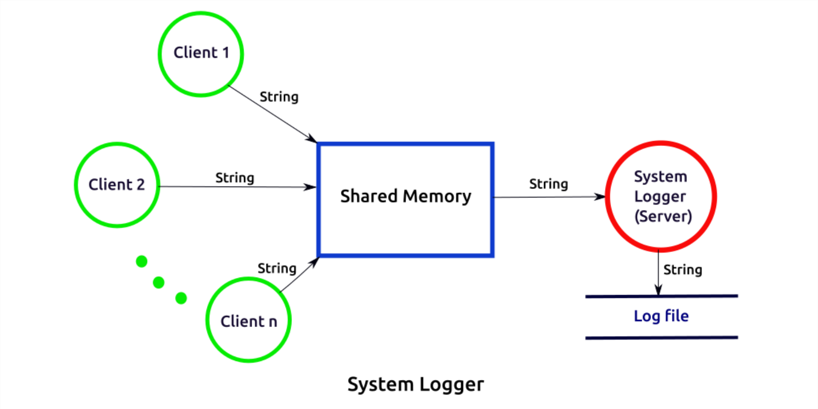
IPC - Shared Memory: Principle

- Multiple processes map the same physical memory to their respective shared memories
- Care must be taken to respect caching
- Care must be taken to provide semaphores/mutexes to avoid race conditions
- A fast replacement for read and write operations
- After the mapping, data are sent to the other process by directly writing a variable

IPC - Shared Memory: Important C Functions

- Using shm_open() function a shared memory object is made available
- Using shm_unlink(), the object can be "closed"
- Using ftruncate(), the initial size of the shared memory is set
- Using mmap(), the shared memory is mapped to the process virtual memory
- Using munmap(), the mapping is removed

IPC - Shared Memory: Logger



IPC - Network Sockets

- Using TCP or UDP communication on 127.0.0.1 / localhost
- Very common - easily portable to networked scenario
- Slower than UNIX Domain Sockets, but cross platform (Windows)
- Not file-backed - data written to a socket is lost on reboot (unlike named pipes)

Package Managers in embedded linux

Embedded or non-embedded?

- It is hard to make categorize
- Package managers used commonly today selected
- Principle is always the same:
  - Setup installation feeds
  - Install, remove, update, list
- Selected: APT, OPKG, SMART

Advanced Packaging Tool (APT)

- Popular in Debian, Ubuntu and related
- Configuration in /etc/apt
- Feeds in /etc/apt/sources.list and/or in /etc/apt/sources.list.d/ folder
- A lot of packages available

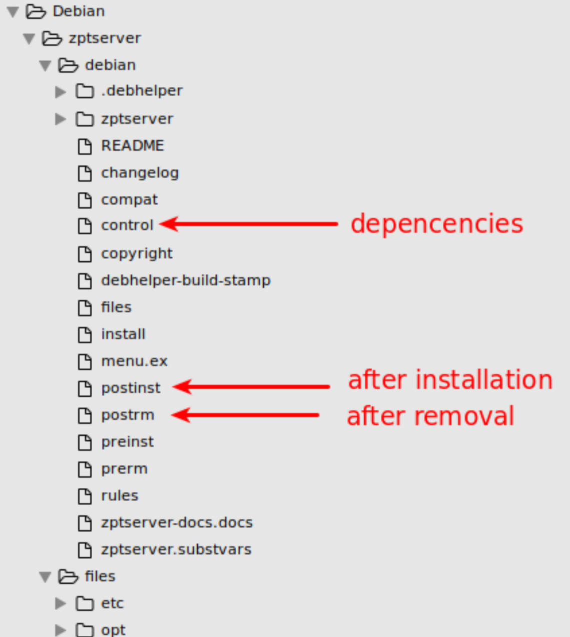
APT - feeds/repositories

- Contain .deb files called packages
- URLs listed in format: “deb ” (e.g.: deb http://deb.debian.org/debian buster)
- Possible to install from file: apt install ./mypackage.deb
- Every package:
    - Contains pre/post install and pre/post remove scripts
    - Names of dependencies/prerequisites
    - Version, maintainer, etc.

APT - operations

- apt install : installation
- apt purge : uninstall and remove
- apt search : search for package
- apt list : list installed packages
- apt update : get updated list of available packages
- can be a wildcard

APT - example package inside



*

Open Package Management (OPKG)

- Used in OpenWRT, Buildroot and Yocto
- Simpler than APT, however less packages available
- Fork and successor of IPKG (dead/dying)
- Command line interface *very* similar to APT
- Uses .ipk packages

OPKG - feeds

```
root@Omega-1773:~# ls /etc/opkg
customfeeds.conf    distfeeds.conf
root@Omega-1773:~# cat /etc/opkg/customfeeds.conf
# add your custom package feeds here
#
# src/gz someName http://www.example.com/path/
```

OPKG - package structure

```
packages/serialnumber/
|-- ipkbuild
|   `-- example_package
|       |-- control.tar.gz
|       |   |-- control   - prerequisites, maintainer name,
|       |   |-- postinst  - run after installation
|       |   |-- preinst   - run before installation
|       |   `-- prerm     - run before removal
|       |-- data.tar.gz   - actual package files
|       |   |-- usr
|       |   |   `-- bin
|       |   |       `-- my_binary
|       `-- debian-binary - version of the packaging used
`-- example_package_1.3.3.7_varam335x.ipk
```

SMART

- Modern, version 1.0 in 2008
- Merged in Yocto project
- Supports RPM, APT, Slackwate “channels”
- Has a command line GUI interface
- Future of packaging in Linux world?
- Link: https://labix.org/smart

Practical part

Outline

- Preparing the workspace - Docker
- First steps with Device tree
- Building first kernel module
- Setting default state of a GPIO pins - two approaches
- Writing character device driver

Docker

- Opensource project for application isolation and deployment
- Building blocks: **images** and **containers**
- **Image** - isolated environment (filesystem) in which an application will run
- **Container** - a process running within an image under Docker engine

Docker - Motivation

- Application isolation on many levels
- Filesystem separation (can be easily connected to host FS with bind mounts)
- Process separation (own set of PIDs, . . .)
- Network isolation
- Applications run natively on host's kernel
- No performance drop
- No memory overhead
- IBM Research on Docker vs VM performance link
- Fast deployment
- Images for most frequently used tools/services available on Dockerhub

Docker - example

- Run a container with ubuntu image `sh    docker run -it ubuntu /bin/bash`
- -i . . . interactive
- -t . . . attach to stdin/stdout
- Access host filesystem from container `sh    docker run -it -v /home/user/wd:/data ubuntu /bin/bash`
- -v<host>:<guest> . . . bind mount <host> directory to <guest> within the container

Preparing the workspace

1. Download the archive to your working directory
   https://files.acrios.com/index.php/s/GAL98g32gdH7HQP
2. Extract the archive sh `tar -xvzf embedded-linux.tar.gz` `cd 01-embedded-linux`
3. Run following command to see all available make targets sh `make help`
4. Build the sandbox (may take few minutes to download all stuff) run `make sandbox`. This step will download and install all required tools to the docker container.

Next time you run ‘make sandbox‘, only changed files will be re-downloaded. Use this command to enter the sandbox.

Building a kernel for RedPitaya

1. Make sure you are in docker container; working directory `/pitaya`.
2. Run `make kernel-download` to only download the kernel sources, or:
3. Run `make kernel-build` to download and build kernel sources with `xilinx_zynq_defconfig`

Important kernel headers

```
/* Error checking/converting macros IS_ERR(), PTR_ERR etc. */
#include <linux/err.h>
/* Common kernel macros. KERN_INFO, ALIGN(), ARRAY_SIZE(), abs() */
#include <linux/kernel.h>
/* module_init, module_exit, meta information macros */
#include <linux/module.h>
/* __init and __exit macros */
#include <linux/init.h>
/* string operations */
#include <linux/string.h>
/* Legacy gpio() functions */
#include <linux/gpio.h>
/* gpiod() functions */
#include <linux/gpio/consumer.h>
```

Important kernel headers

```
/* Kernel device model */
#include <linux/device.h>
/* Kernel platform device model */
#include <linux/platform_device.h>
/* kmalloc() */
#include <linux/slab.h>
/* Working with device nodes and properties */
#include <linux/of.h>
/* Unified device property interface */
#include <linux/property.h>
/* Bit operations */
#include <linux/bitops.h>
/* struct file_operations, chardev */
#include <linux/fs.h>
/* Userspace access - copy from user, copy to user ... */
#include <linux/uaccess.h>
```

Example 1 - Device tree

`src/01_device_tree`

1. Inspect attached `Makefile`
2. Inspect and build the prepared dts `socfpga_stratix10_socdk.dts` using the `make`
3. Check the whole build process, try to decompile
4. What are the differences between the original source and the decompiled one? Why?

```
sh  cd 01_device_tree  make dtb  make dts
```

Example 2 - First kernel module

`src/02_first_module`

1. Check the source code
2. Build the module using prepared Makefile
3. Load the module to kernel

1. Built natively –> load in your host OS
2. Built in sandbox –> transfer to RedPitaya and load there

4. Load and unload the module, check results in **dmesg**

Example 2 - Kernel module build

```
cd 02_first_module
make
```

```
cd <working directory>
insmod first_module.ko
dmesg | tail
# Check output
rmmod first_module
dmesg | tail
# Check output
```

Example 3 - Kernel module parameters

`03_parameters`

1. Check the source code
2. Build the module using prepared Makefile
3. Load the module to kernel
    1. Built natively –> load in your host OS
    2. Built in sandbox –> transfer to RedPitaya and load there
4. Check how passing module parameters works
5. Add description to each parameter. Check by `modinfo` tool.

Example 3 - Kernel module parameters

docker

```
cd 03_parameters
make
```

redpitaya

```
cd <working directory>
insmod module_parameters.ko
dmesg | tail
# Check output
rmmod module_parameters
insmod module_parameters.ko <param1=x> <param2=y>
dmesg | tail
# Check output
rmmod module_parameters
```

Example 4 - GPIO on RedPitaya

Update the existing GPIO controller driver so it supports "default-on" property.

1. Inspect the device tree used on the running board
2. Check the module hierarchy to handle onboard LEDs
3. Add a `default-on` property to GPIO controller node. The property is an array of pin numbers that will become "ON" after driver startup. Use pins of user LED 3 and 4.
4. Update the driver of used GPIO controller so that it loads the array of pins from the DeviceTree and sets them to log. "1"
5. Rebuild the kernel and replace the default kernel on RedPitaya
6. Boot the board with new kernel

Example 4 - Inspecting the onboard devtree

- The device tree blob is stored in `/boot/devicetree.dtb` on the target
- Copy the blob to `src/04_gpio_zynq` folder
- Convert the blob to source and inspect in text editor `sh dtc -I dtb -O dts <source> -o <output>`
- Find out the compatible string of GPIO controller
- Find the GPIO controller driver within the Linux kernel tree `sh cd src/linux-xlnx grep -Hnr --color "the_compatible_string" *`

Example 4 - Making changes

- Add the `default-on` property to the GPIO controller node, containing numbers of pins used for user LED3 and LED4
- Modify the `probe` function of the GPIO controller driver so it:
- loads an array of integers from `default-on` property
- sets appropriate pin directions to "OUT"
- sets appropriate pin states to "1"

Example 4 - Build and load

- Build the device tree
- Rebuild the kernel with updated module

- Remount the `/boot` partition as RW
- Replace the `uImage` and `devicetree.dtb`

Example 5 - GPIO on RedPitaya, the right way

Create a custom driver for setting default GPIO states.

1. Create a new module for setting default states of chosen GPIO pins. Get inspired by the `leds-gpio.c`
2. Use device tree source from last step, remove user leds 0 and 1 and use those pins for node “gpio-default”
3. Build and load the module and device tree binary to the RedPitaya
4. Boot the board and check results

Example 5 - DevTree structure

```
gpio-default {
        compatible = "gpio-defaults";

        pin1 {
            gpios = <&gpio0 58 0>;
            default-state = "off";
        };

        pin2 {
            gpios = <&gpio0 59 0>;
            default-state = "off";
        };
    };
```

Example 6 - Character device driver

Create a new module implementing character device functionality.

1. Create a new module implementing character device functionality
2. Implement file operations so that:

- Each open() will be counted and printed to kernel debug buffer
- Writing to device will store a message into a buffer. Number of bytes written will be printed to kernel debug buffer.
- Reading from device will print last stored message

3. Build and load the module to your OS or to RedPitaya.
4. Check functionality by writing/reading the /dev node and `dmesg`
5. Inspect related `/sys/class` nodes

Questions?

Backup slides

Booting into read-only filesystem

- Devices that do not need to write any data to filesystem (FS) often mount the filesystem as read-only
  `mount -t ext4 /dev/sda1 / -odefaults,ro`
- Useful to prevent from undesired writes to flash memory
- Tmpfs (ramdisk) used as a volatile writable FS
- To prevent filesystem corruption when errors are detected
  `mount -t ext4 /dev/sda1 / -odefaults,errors=remount-ro`
- seamlessly remount as read-write
  `sudo mount -o remount,rw /`

Links