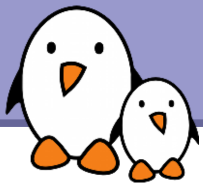
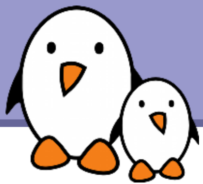


Accessing hardware from userspace

Michael Opdenacker
Thomas Petazzoni
Bootlin



Rights to copy

© Copyright 2009, Bootlin
feedback@free-electrons.com

Document sources, updates and translations:
https://bootlin.com/doc/legacy/accessing-hardware

Corrections, suggestions, contributions and translations are welcome!

Latest update: Jul 25, 2018

Attribution – ShareAlike 3.0

You are free

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions

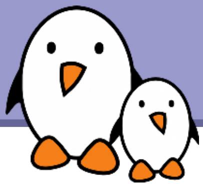
Attribution. You must give the original author credit.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

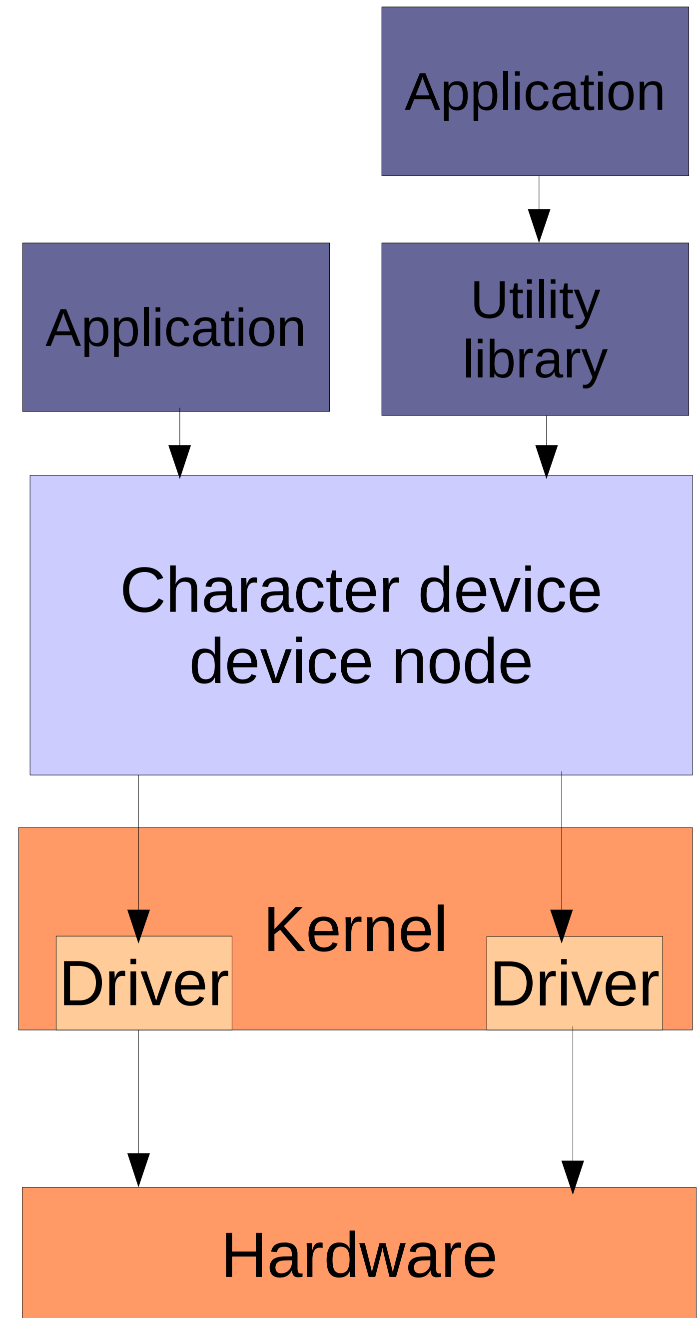
Your fair use and other rights are in no way affected by the above.

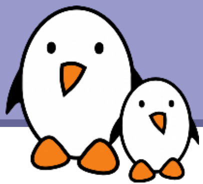
License text: http://creativecommons.org/licenses/by-sa/3.0/legalcode



Kernel vs. userspace

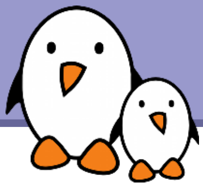
- For most devices, the driver
  - Is inside the kernel
  - Provides an interface for userspace application to communicate with the hardware
- The interface is usually
  - A character device
  - A character device preferably accessed through an utility library
- There are some exceptions: block devices, network interfaces, printers or graphics with X.org.





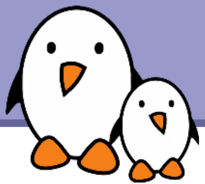
Character device API

- Character devices are seen by userspace applications as files, so the traditional Unix file API is available
  - `open()` and `close()` on the device file
  - `read()` to get data from the device
  - `write()` to send data through the device
  - `ioctl()` to perform special operations on the device
  - `poll()` and `select()` to wait for events
  - `mmap()` to remap the device memory into the process address space
- The kernel driver is responsible for implementing this API, so that from the perspective of the userspace application, communicating with the hardware is very simple.



ioctl()

- `ioctl()` is a function of the C library, a system call, and an operation of character device driver
- It is used to implement operations specific to the device or device type, such as setting the serial port speed, changing the screen resolution, adjusting the video capture format, etc.
- Prototype of the function in userspace
  `int ioctl(int d, int request, ...)`
  - d is the file descriptor
  - `request` is a number identifying the operation. This number is device or device-type specific.
  - `...` is an unlimited number of arguments. The number of arguments, their type and semantic depend on the ioctl operation
- See `man ioctl`



Example with the serial port

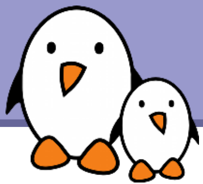
```
#include <termios.h>
#include <fcntl.h>
#include <sys/ioctl.h>

int main(void)
{
   int fd, serial;
   fd = open("/dev/ttyS0", O_RDWR);
   write(fd, “Hello”, 5);
   ioctl(fd, TIOCMGET, &serial);
   if (serial & TIOCM_DTR)
      printf("TIOCM_DTR is not set");
   else
      printf("TIOCM_DTR is set");
   close(fd);
}
```

Open the device

Write to it

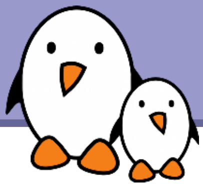
Perform a special operation on it : get the status bits

Warning: error checking omitted!



ioctl operations for serial ports

- There are many ioctl operations for serial ports, as defined by the `tty_ioctl` manual page. For some of them, POSIX also specifies functions to wrap ioctl operations
  - `TCGETS` operation is similar to the `tcgetattr()` function, it gets a termios structure
  - `TCSETS` operation is similar to the `tcsetattr()` function, it sets a termios structure
  - `TCSBRK` similar to `tcsendbreak()`, it sends a break
  - `TCXONC` similar to `tcflow()`, to control the software flow control
  - `TCFLSH` similar to `tcflush()`, to flush the input or output buffers
  - `TIOCMGET` to get the status of the modem bits
  - `TIOCMSET` to set the status of the modem bits
- See the `termios` manual page for more details



Termios example

```
struct termios options;

fd = open(“/dev/ttyS0”, O_RDWR);

tcgetattr(fd, &options);

cfsetispeed(&options, B19200);
cfsetospeed(&options, B19200);
options.c_cflag |= (CLOCAL | CREAD);

tcsetattr(fd, TCSANOW, &options);
```

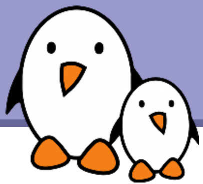
Get current value of the termios structure

Modify the settings

Set the new termios structure

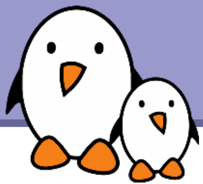
Warning: error checking omitted!

I2C



I2C from userspace

- Some I2C devices have directly a driver in the kernel
  - In this case, the driver is tied to the appropriate kernel infrastructure, depending on the device type
  - It is made available to userspace through this infrastructure
- The `i2c-dev` driver allows an userspace application to directly interact on the I2C bus
  - Character devices are created in userspace for each I2C adapter
  - Major is 89, the minor is the adapter number
  - Conventional name is `/dev/i2c-0`, `/dev/i2c-1`, etc.
  - See `/sys/class/i2c-dev/` or run `i2cdetect -l` for a list
  - `i2cdetect` is part of the `i2c-tools` package in Ubuntu/Debian distributions



i2c-dev

- Open the i2c-dev device

```
fd = open(“/dev/i2c-0”, O_RDWR);
```

- Specify the device with which you want to communicate

```
ioctl(fd, I2C_SLAVE, 0x40);
```

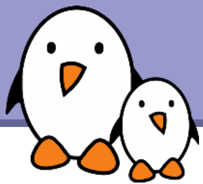
- Write to the bus

```
buf[0] = register;
buf[1] = data1;
buf[2] = data2;
write(fd, buf, 3);
```

- Read from the bus

```
read(fd, buf, 1);
```

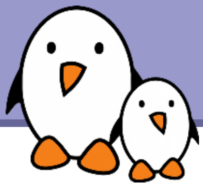
- See `Documentation/i2c/dev-interface` for details



Accessing hardware directly (1)

- The `/dev/mem` character device allows to access directly to the physical memory, including I/O memory
  - read() or write() operations are possible
  - mmap() operation is also possible, to remap specific parts of the physical memory to the address space of the application.
  - Obviously, access rights to this device must be properly set, as it allows to do anything with the system
    `crw-r----- 1 root kmem 1, 1 2009-04-28 10:37 /dev/mem`
- Exemple from the X.org server

```
fd = open ("/dev/mem", O_RDWR);
a = mmap (0, size, PROT_READ|PROT_WRITE,
          MAP_SHARED, fd, addr);
close(fd);
```

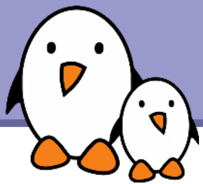


Accessing hardware directly (2)

- Other character device drivers can also provide a mmap() operation which can be used by userspace applications
- For example, in DirectFB, a library that relies on the kernel framebuffer driver :

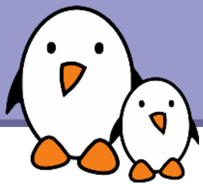
```
int fd;
char *fbbase;
fd = open(“/dev/fb0”, O_RDWR);
fbbase = mmap(NULL, shared->fix.smem_len,
            PROT_READ | PROT_WRITE,
            MAP_SHARED, fd, 0);
```

- Then the fbbase pointer can be used to directly read and write to the framebuffer.



GPIOs

- GPIOs can be directly accessed through `/dev/mem` or a specific character driver implementing the `mmap()` operation
- If the board code supports the `gpiolib` kernel framework, GPIOs are made available to userspace through sysfs
  - `/sys/class/gpio/gpioN/` directory for each GPIO
    - `direction` file to configure the direction (either in and out)
    - `value` file to configure the value (0 or 1)
  - `/sys/class/gpio/`
    - `export` allows to export GPIO to userspace that haven't been explicitly exported by the kernel. Writing the GPIO number is sufficient
    - `unexport` allows to unexport GPIOs that have previously been exported
  - See `Documentation/gpio.txt` in kernel sources for details.



Utility libraries

- Most of the device drivers in the kernel fit inside a framework, that unifies the set of operations that can be performed on a device of a given type
- Some of these device types must be used directly as character devices by userspace applications
- For other device types, an utility library is available to ease usage of the device
  - ALSA sound devices, represented as character devices in userspace, are better used through `libasound`
  - Video 4 Linux devices through libv4l
  - Framebuffer devices through DirectFB
- Don't forget to check if an utility library exists for your case, or write your own !