Digital electronics

J.-M Friedt

FEMTO-ST/time & frequency department

jmfriedt@femto-st.fr

slides at jmfriedt.free.fr

March 17, 2021

Plan

7 lessons/lab sessions (4-hour long schedules):

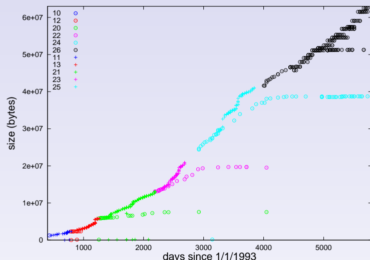
1. Executive environments: principles and introduction, getting started with FreeRTOS
2. FreeRTOS, RTEMS, Nuttx ... multitasking and associated methods to make sure shared data and resources are kept in known states (mutex & semaphore)
3. Using the scheduler, mutex and semaphores to solve the "philosopher problem"
4. Embedded systems with GNU/Linux – POSIX compatible operating system
   Architecture of an operating system, kernel v.s userspace
   Internet connectivity and networking
5. Accessing hardware resources from userspace – memory translation from physical to virtual address space (Memory Management Unit) – `/dev/mem`
6. Accessing hardware resources from a web server – internet connected instrument
7. From userspace to kernel space: character device (*char device*) for communicating between users and the kernel

Introduction: kernel & kernel modules

- Developper ⇒ focus on the software-hardware interface
- Only the kernel can access all resources of the processor (e.g. interrupts, DMA)
- The kernel shares resources and make sure hardware access is consistent (kernel = supervisor)
- Adding new functionalities to the kernel: modules ("plugins")

Porting GNU/Linux to a new architecture

- a (cross-)compilation toolchain
- define the memory map of the target (`-T` option of `ld`)
- bootloader development (peripheral initialization and loading the kernel to memory)[1]
- developing kernel modules for handling hardware dedicated to the new architecture [2].

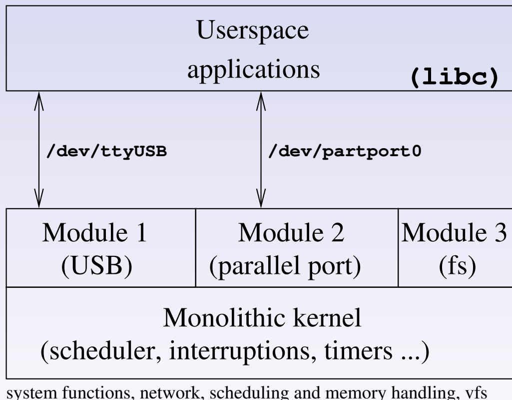


Evolution of the Linux kernel `.tar.gz` archive size (`www.kernel.org`)

[1] S. Guinot, J.-M Friedt, *GNU/Linux sur Playstation Portable*, GNU/Linux Magazine France **114**, Mars 2009, pp.30-40

[2] J. Corbet, A. Rubini & G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*, O'Reilly (2005), available at à `http://lwn.net/Kernel/LDD3/`

Operating System (OS) architecture



The /dev directory

- Pseudo-files linking userspace and kernelspace through system calls implemented in `libc` (`man syscalls`)
- each *device* is defined with a *major number* and, within each class, a *minor number*
- the kernel is not aware of filesystems: on the kernel side, a driver is associated with a given major number and handles each syscall:
  - `open()`
  - `close()`
  - `write()`
  - `read()`
  - `ioctl()`

```
static struct file_operations qadc_fops =
{
        owner:          THIS_MODULE,
        read:           qadc_read,
        open:           qadc_open,
        ioctl:          qadc_ioctl,
        release:        qadc_release,
};

static int __init qadc_init (void)
{...
 register_chrdev (qadc_major, "ppp", &qadc_fops);
 ...}

module_init (qadc_init);
module_exit (qadc_exit);
```

Basic structure

Kernel module ⇒ no `main()` starting function:

- one entry and one exit function:
  `module_init (my_init_function);` and
  `module_exit (my_exit_function);`
- initializing the communication link with userspace:
  `register_chrdev (90 , "jmf" ,&fops );`
- information displayed through `/var/log/syslog` using
  `printk (KERN_ALERT "formatted message");` (*no coma !*)
- ability to dynamically create dev entries using *misc. devices*

```
int hello_start()  // init_module(void)
{jmfdev.name = "jmf";  // /dev/jmf: major = classe misc
 jmfdev.minor = MISC_DYNAMIC_MINOR;
 jmfdev.fops = &fops;
 misc_register(&jmfdev); // dynamic creation of /dev/jmf
}
```

Mondule handling commands

Modules (*kernel object*) with `.ko` extension

- are linked to the kernel with `insmod toto.ko`,
- are removed from the kernel with `rmmod toto`.
- list of attached modules with `lsmod`
- Messages logged by modules using `printk` are displayed with `dmesg`
- "Official" modules are located in `/lib/modules`
- "Official" modules with dependencies are loaded with `modprobe` [3]

[3] see `/lib/modules/version/modules.dep`

Directories needed for compiling modules

The sources of the kernel running on the PC are located in `/usr/src`:

- Debian/GNU Linux separates generic headers from parts specific to a given architecture
- `/usr/src/linux-headers-*-common` does *not* includes parts specific to the current architecture
- `/usr/src/linux-headers-*-*arch` is the directory we use to find the specific configuration of the kernel running on the host.

New `Makefile` architecture for compiling modules relying on the kernel `Makefile` by adding the new kernel target to the `obj-m` variable with

```
obj-m +=0mymod.o 1mymod.o 2mymod.o
all:
	make -C /usr/src/linux-headers-4.8.0-1-amd64 M=$(PWD) modules
```

For cross-compiling to the target architecture: `output/build/linux-*` in Buildroot provides the Linux kernel

```
obj-m +=0mymod.o 1mymod.o 2mymod.o
all:
	make ARCH=arm CROSS_COMPILE=arm-linux- -C \
    ../buildroot-2020.11.1/output/build/linux-custom/ M=$(PWD) modules
```

Real v.s virtual addresses

Same problem as /dev/mem and mmap in userspace: here too the MMU must provide the virtual block address translated to the physical address:

```
request_mem_region (IO_BASE, 36*9, "GPIO test" );// PG9
jmf_gpio = ( u32 )ioremap(IO_BASE, 36*9);      // 36 bytes/port, ports A-I
writel (0x10, jmf_gpio+36*6+0x04 );             // GPIO PG9 en sortie
```

30.2. Port Register List

Module Name	Base Address
PIO	0x01C20800

Register Name	Offset	Description
Pn_CFG0	n*0x24+0x00	Port n Configure Register 0 (n from 0 to 9)
Pn_CFG1	n*0x24+0x04	Port n Configure Register 1 (n from 0 to 9)
Pn_CFG2	n*0x24+0x08	Port n Configure Register 2 (n from 0 to 9)
Pn_CFG3	n*0x24+0x0C	Port n Configure Register 3 (n from 0 to 9)
Pn_DAT	n*0x24+0x10	Port n Data Register (n from 0 to 9)
Pn_DRV0	n*0x24+0x14	Port n Multi-Driving Register 0 (n from 0 to 9)
Pn_DRV1	n*0x24+0x18	Port n Multi-Driving Register 1 (n from 0 to 9)
Pn_PUL0	n*0x24+0x1C	Port n Pull Register 0 (n from 0 to

Addresses given in the microprocessor datasheet (here Allwinner A10)

Methods provided by the kernel

Many functionalities provided by the kernel for handling tasks (scheduler), timer or communicating with userspace.

- concepts close to microcontroller programming

```
init_timer_on_stack (&exp_timer );
exp_timer.expires = jiffies+delay*HZ ;
exp_timer.data = 0;
exp_timer.function = do_something ;
add_timer (&exp_timer);
```

- *tasklet*: software interrupts

```
char my_tasklet_data[]="my_tasklet_function was called";
void my_tasklet_function( unsigned long data )
{ printk( "%s\n", (char *)data ); return; }

DECLARE_TASKLET( my_tasklet, my_tasklet_function,
 (unsigned long) &my_tasklet_data );

int init_module( void )
{ tasklet_schedule( &my_tasklet ); return 0; }

void cleanup_module( void )
{ tasklet_kill( &my_tasklet ); return; }
```

$\rightarrow$ ISR should be **short** and **schedule** a tasklet

Kernel module: interrupt handling

Only a kernel module can handle interrupts (e.g. interrupt 7 is associated with pin 10 of the parallel port on a PC)

```
jmfriedt@(none):~$ cat /proc/interrupts | grep parp
  7:          0    IO-APIC-edge  parport0
```

handled with [4]

```
#define PARALLEL_PORT_INTERRUPT 7
#define BASEPORT                0x378
int s=0;
static int int_handler(void) {printk(">>> PARALLEL PORT INT\n");s=1;}
static int __init jmfpport_init_module (void) {
  ret = request_irq(PARALLEL_PORT_INTERRUPT, &int_handler,
                    SA_INTERRUPT, "parallelport", NULL);
  enable_irq(7);
  outb_p(0x10, BASEPORT + 2);
}
static ssize_t skeleton_read (struct file *file, char *buf,
                size_t count, loff_t *ppos) {
 [...]
 do {} while (sortir==0);   // interruptible_sleep_on(&skeleton_wait);
 err = copy_to_user(buf,string,counter);
 ...
}
```

[4] www.captain.at/howto-linux-device-driver-template-skeleton.php

Practical demonstration

1. Kernel module for Linux
2. Converting from virtual to real memory in kernel space
3. Dynamic communication entry creation in `/dev`
4. Reading a datasheet (hardware addresses) remains mandatory !
5. GNU License to be allowed to access the functionalities provided by the kernel or other modules (hardware abstraction)

→ convert the userspace C blinking LED example to a kernel module