Embedded systems 7/7

J.-M Friedt

FEMTO-ST/time & frequency department

jmfriedt@femto-st.fr

slides at jmfriedt.free.fr

October 11, 2020

Linux kernel: a toolbox

- standard interface to access peripherals from userspace (*timer*, `gpio`)
- access to scheduler to add tasks (*tasklet*)
- consistent access to resources: semaphore and mutex

Handling events (e.g. **interrupts**) and propagate a signal to userspace applications

Kernel API

Core functions provided by the kernel to handle userspace applications: scheduler, timer, communication with userspace

- tasks (*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; }
```

→ ISR must be short and create a tasklet scheduled when time allows

Hardware access

Solution 1: `ioremap` to identify virtual address and set registers

Solution 2: use existing drivers provided by the kernel (`make linux-menuconfig` in Buildroot)

```
CONFIG_GPIO_XILINX:
Say yes here to support the Xilinx FPGA GPIO device
Symbol: GPIO_XILINX [=m]
Type  : tristate
Prompt: Xilinx GPIO support
 Location:
   -> Device Drivers
     -> GPIO Support (GPIOLIB [=y])
       -> Memory mapped GPIO drivers
 Defined at drivers/gpio/Kconfig:573
 Depends on: GPIOLIB [=y] && HAS_IOMEM [=y] && OF_GPIO [=y]
```

or for other platforms:

```
CONFIG_GPIO_SUNXI:
This option enables support for GPIOs on the Allwinner SOCs
(sun4i/sun5i/sun7i). The GPIOs must be defined in [gpio_para]
section of sysconfig.fex file (gpio_used/gpio_num/gpio_pin_x variables)
Symbol: GPIO_SUNXI [=y]
Type  : tristate
Prompt: GPIO Support for sunxi platform
  Depends on: GPIOLIB [=y] && (ARCH_SUN4I [=n] || ARCH_SUN5I [=y] || ARCH_S
```

Hardware access

Solution 1: ioremap to identify virtual address and set registers
Solution 2: use existing drivers provided by the kernel (make linux-menuconfig in Buildroot)
Using GPIO: define pins in *devicetree* [1]

```
gpio-leds {
  compatible = "gpio-leds";
  led-8-yellow {
    label = "led8";
    gpios = <&gpio0 0 0>;
    default-state = "off";
    linux,default-trigger = "mmc0";
  };
  led-9-red {
    label = "led9";
    gpios = <&gpio0 7 0>;
    default-state = "off";
    linux,default-trigger = "heartbeat";
  };
};
```

or (pin)+906

[1] redpitaya/board/redpitaya/zynq-red_pitaya.dts in the BR2_EXTERNAL since the Redpitaya is not officially supported by Buildroot

Kernel module: using gpiolib & interrupt handling

Test if GPIO is available (22 = EINVAL, 16 = EBUSY) [2]

```
static irqreturn_t irq_handler(int irq, void *dev_id)
{...
  return IRQ_HANDLED;
}

gpio =906+10;   // MIO10
err=gpio_is_valid(gpio);
err=gpio_request_one(gpio, GPIOF_IN, "jmf_irq");
if (err!=-22)
  {irq = gpio_to_irq(gpio);
   irq_set_irq_type(irq, IRQ_TYPE_EDGE_BOTH);
   err = request_irq(irq, irq_handler, IRQF_SHARED, "GPIO jmf", &dummy);
   dummy=0;
  }
```

[2] /usr/include/asm-generic/errno-base.h

Signals: event handler in userspace

Userspace equivalent to software interrupt: signals

```
#include <signal.h>

void my_handler (int sig) {
    printf ("I got SIGINT, number %d\n", sig);}

int main ( void ) {
    signal (SIGINT, my_handler);
    while (1) {}
}
```

leads to, every time CTRL-C is hit (kill with `kill -9 PID`)

```
jmfriedt@(none):~$ ./sigint
I got SIGINT, number 2
I got SIGINT, number 2
```

Interrupt signal distribution

Sharing interrupt information through signals

1. a unique hardware interrupt handling module (kernel)
2. multiple userspace programs might react to an interrupt
3. each program registers with the module
4. the module reacts quickly to the interrupt signal ...
5. ... and informs the programs identified through their PID when time allows.

Interrupt signal distribution

Alternate solution (*thread*)

1. a program wants to be notified of an interrupt as it is running
2. creates a thread blocked when reading a communication interface with the kernel in `/sys/class` or `/dev`
3. when the interrupt is triggered, the module unblocks the read syscall handling function ... [3] [4]
4. ... and the thread notifies the program of the event.

[3] `http://www.makelinux.net/ldd3/chp-6-sect-2`
[4] `http://www.makelinux.net/ldd3/chp-5-sect-3` for kernel semaphores

Shared data protection

Multiple tasks use the same data ⇒ consistency issues
Three methods to protect shared data handling:

- semaphore (counter)
- mutex (binary with the task removed from the scheduler)
- spin-lock (binary with active testing of the lock)

Producer-consumer:

1. one task produces data
2. a user requests these data
3. reading is **blocked** as long as the data have not been produced

- Requesting a semaphore blocks the execution if already null
- Raising the semaphore unblocks the waiting process/task
- The counter can rise multiple time to unblock multiple processes

```
#include <linux/semaphore.h>
struct semaphore mysem;

sema_init(&mysem, 0);       // init the semaphore as empty

down (&mysem); // blocks consumer
...
up (&mysem);   // producer unblocks
```

Shared data protection: mutex

Blocks-unblocks parts of the software using a common resource:
only one part of a software using a mutex can be executed at any given time

```
#include <linux/mutex.h>
struct mutex mymutex;

mutex_init(&mymutex);

mutex_lock(&mymutex);   // blocks
...
mutex_unlock(&mymutex); // unblocks
```

As many mutex as shared resources

Shared data protection: spinlock

A mutex removes the task from the scheduler $\Rightarrow$ slow
For fast operations (interrupts), actively probe the lock: `spinlock`

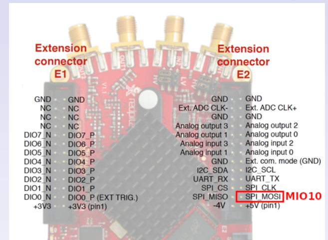
```
#include <linux/spinlock.h>
static DEFINE_SPINLOCK(myspin);

spin_lock_init(&myspin);

spin_lock(&myspin);
...
spin_unlock(&myspin);

MODULE_LICENSE("GPL");
```

deadlock issue: two processes calling simultaneously blocking conditions (e.g. SMP architecture with two cores locking mutex depending on each other).

Demonstration with GPIO

- The two GPIO available are connected to LEDs ⇒ output only
- Other MIO are set for communication (I2C, SPI in `ps7_init.c`) ⇒ reconfigure as GPIO
- MIO10 is pin 3 of E2 connector
- 3 bits L0_SEL, L1_SEL and L2_SEL select the GPIO function

```
> devmem 0xF800012C # GPIO clock active
0x00500444
> devmem 0xF8000728 # A = SPI
0x000016A0
> devmem 0xF8000728 32 0x00001600
```



Field Name	Bits	Type	Reset Value	Description
L3_SEL	7:5	rw	0x0	Level 3 Mux Select 000: GPIO 10 (bank 0), Input/Output 001: CAN 0 Rx, Input 010: I2C 0 Serial Clock, Input/Ouput 011: PJTAG TDI, Input 100: SDIO 1 IO Bit 0, Input/Output 101: SPI 1 MOSI, Input/Output 110: reserved 111: UART 0 RxD, Input
L2_SEL	4:3	rw	0x0	Level 2 Mux Select 00: Level 3 Mux 01: SRAM/NOR Data Bit 7, Input/Output 10: NAND Flash IO Bit 5, Input/Output 11: SDIO 0 Power Control, Output
L1_SEL	2	rw	0x0	Level 1 Mux Select 0: Level 2 Mux 1: Trace Port Data Bit 2, Output
L0_SEL	1	rw	0x0	Level 0 Mux Select 0: Level 1 Mux 1: Quad SPI 1 IO Bit 0, Input/Output
TRI_ENABLE	0	rw	0x1	Operates the same as MIO_PIN_00[TRI_ENABLE]

Register (slcr) MIO_PIN_10

Name	MIO_PIN_10
Relative Address	0x00000728
Absolute Address	0xF8000728
Width	32 bits
Access Type	rw
Reset Value	0x00001601
Description	MIO Pin 10 Control

Register MIO_PIN_10 Details

Field Name	Bits	Type	Reset Value	Description
reserved	31:14	rw	0x0	reserved
DisableRcvr	13	rw	0x0	Operates the same as MIO_PIN_00[DisableRcvr]
PULLUP	12	rw	0x1	Operates the same as MIO_PIN_00[PULLUP]
IO_Type	11:9	rw	0x3	Operates the same as MIO_PIN_00[IO_Type]
Speed	8	rw	0x0	Operates the same as MIO_PIN_00[Speed]

Accessing the GPIO using gpiolib: interrupt handling

```
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */
#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/gpio.h>

static int dummy, irq, dev_id, jmf_gpio=906+10;

static irqreturn_t irq_handler(int irq, void *dev_id)
{dummy++; printk(KERN_INFO "plip %d",dummy);
 return IRQ_HANDLED;
}

int hello_start()
{int err;
 err=gpio_is_valid(jmf_gpio);
 err=gpio_request_one(jmf_gpio, GPIOF_IN, "jmf_irq");
 if (err!=-22)
    {irq = gpio_to_irq(jmf_gpio);
     irq_set_irq_type(irq, IRQ_TYPE_EDGE_BOTH);
     err = request_irq(irq, irq_handler, IRQF_SHARED, "GPIO jmf", &dev_id);
     dummy=0;
    }
 return 0;
}

void hello_end()
{free_irq(irq,&dev_id);
 gpio_free(jmf_gpio);
}

module_init(hello_start);
module_exit(hello_end);

MODULE_LICENSE("GPL");  // needed to use Linux kernel functions
```

Accessing the GPIO using gpiolib: sending a signal

```
#include <linux/sched/signal.h>       // kill_pid
#ifdef __ARMEL__                      // on Redpitaya
#include <linux/gpio.h>

static irqreturn_t irq_handler(int irq, void *dev_id)
#else                                 // on PC
struct timer_list exp_timer;

static void irq_handler(struct timer_list *t)
#endif
{struct pid *mypid;
 if (pid!=0)
    {mypid= find_vpid(pid);
     if (mypid == NULL)
        pr_info("Cannot find PID from user program\r\n");
     else // do_send_sig_info(SIGUSR1, SEND_SIG_FORCED, task, PIDTYPE_MAX);
        kill_pid(mypid, SIGUSR1, 1);
    }
#ifdef __ARMEL__
 return IRQ_HANDLED;
#else
 mod_timer(t, jiffies + HZ);
#endif
}

static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off)
{int mylen;
 char buf[15];
 if (len>14) mylen=14; else mylen=len;
 if (copy_from_user(buf, buff, mylen) == 0) sscanf(buf, "%d", &pid);
 printk("PID %d registered",pid);
 return len;
}
```

Accessing the GPIO using gpiolib: creating a task to handle the interrupt

```
static struct work_struct irq_work;

static irqreturn_t irq_handler(int irq, void *dev_id)
{schedule_work(&irq_work);  // long task handled when time allows
 return IRQ_HANDLED;
}

void do_something(struct work_struct *data)
{struct pid *mypid;
 if (pid!=0)
    {mypid= find_vpid(pid);
     if (mypid == NULL)
        pr_info("Cannot find PID from user program\r\n");
     else // do_send_sig_info(SIGUSR1, SEND_SIG_FORCED, task, PIDTYPE_MAX);
          kill_pid(mypid, SIGUSR1, 1);
    }
#ifdef __ARMEL__
 return IRQ_HANDLED;
#else
 mod_timer(t, jiffies + HZ);
#endif
}

int hello_start()  // init_module(void)
{...
 INIT_WORK(&irq_work, do_something);
 ...
 err = request_irq(irq, irq_handler, IRQF_SHARED, "GPIO jmf", &irq_id);
}

void hello_end() // cleanup_module(void)
{...
 free_irq(irq,&irq_id);
}
```

Accessing the GPIO using gpiolib: userspace

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

void signal_handler(int signum)
{printf("Signal received\r\n");}

int main()
{FILE *f;
 signal(SIGUSR1, signal_handler);
 printf("My PID is %d.\n", getpid());
 f=fopen("/dev/jmf","w");  // tune to communication interface with module
 fprintf(f,"%d\n", getpid()); fflush(f);
 fclose(f);
 while (1) {};
 return 0;
}
```

Demonstrate `read()` blocking from a separate thread, unblocked by interrupt trigger

```
#include <pthread.h>                                   // compile with -lpthread

void *function(void *dummy) {[...]}    // blocking read and react to interrupt

int main()
{pthread_t my_thread;
 pthread_create(&my_thread, NULL, function, NULL); // last NULL = dummy param
 pthread_join(my_thread, NULL);
}
```