Embedded Linux

Communication Between Kernel and User Space

© 2017 Arm Limited

Goal

To delve into more details about Linux Kernel modules and to illustrate the communication mechanism between Kernel and user memory spaces

© 2017 Arm Limited

Summary

Introduction

The reference use case

The module-level point of view

The user-level point of view

© 2017 Arm Limited

Summary

Introduction

The reference use case

The module-level point of view

The user-level point of view

© 2017 Arm Limited

Introduction

When designing an application that communicates with custom hardware, two levels shall be considered:

- User level, where the behavior of the application is defined using virtual file system (VFS) calls to communicate with the hardware
- Module level, where the behavior of each VFS function is implemented based upon the functionalities the hardware implements

Different hardware may require different implementations of the same VFS function.

At the user-level, the programmer shall be aware of the functionalities the hardware provides, and shall use the VFS calls accordingly.

© 2017 Arm Limited

Summary

Introduction

The reference use case

The module-level point of view

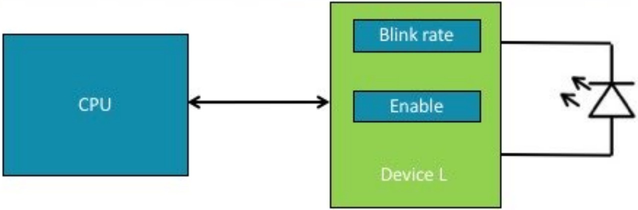
The user-level point of view

© 2017 Arm Limited

The Reference Use Case

To illustrate the concept let's consider this example:

- The custom hardware device L is attached to the CPU, and it controls a led.
- When enabled, L turns led on/off according to a user-defined blink rate.
- Blink rate register: 32 bits with the blink rate in Hz
- Enable register: 1 bit, when set to 0, the device L disabled; when set to 1, the device L enabled.



 © 2017 Arm Limited

arm

The CPU/Device Interface

CPU/Device L connection can be either:

- Memory mapped: each register is associated to an address, as in the example below.

Blink rate register	0xf0080000
Enable register	0xf0080004

- Through GPIO, where each register is associated to a set of GPIOs, as in the example below.

Blink rate register	gpio(0-31)
Enable register	gpio(32)

- Serial Communication (SPI, I2C, etc.)

The module-level implementation will be affected by the adopted CPU/Device Interface.

The user-level implementation will abstract these details.

© 2017 Arm Limited

Summary

Introduction

The reference use case

The module-level point of view

The user-level point of view

© 2017 Arm Limited

The Module Level

(แจกแจง)

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- Reset: L is disabled, and the blink rate register is set to zero.
- Program: The blink rate register is set to a user-defined value.
- Enable: L is enabled.
- Disable: L is disabled.
- Poll rate: L returns the content of the blink rate register.
- Poll state: L returns the content of the enable register.
- Power-off: L terminates its operation and starts waiting for the next reset.

© 2017 Arm Limited

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- Reset: L is disabled, and the blink rate register is set to zero.
- Program: The blink rate register is set to a user-defined value.
- Enable: L is enabled.
- Disable: L is disabled.
- Poll rate: L returns the content of the blink rate register.
- Poll state: L returns the content of the enable register.
- Power-off: L terminates its operation and starts waiting for the next reset.

Both the blink rate and enable registers are set to zero.
This operation is done once, when starting using the device.

 © 2017 Arm Limited

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- Reset: L is disabled, and the blink rate register is set to zero.
- Program: The blink rate register is set to a user-defined value.
- Enable: L is enabled.
- Disable: L is disabled.
- Poll rate: L returns the content of the blink
- Poll state: L returns the content of the enable register.
- Power-off: L terminates its operation and starts waiting for the next reset.

The blink rate register is set to a user defined value.
This operation can be done multiple times, during the device usage.

© 2017 Arm Limited

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- Reset: L is disabled, and the blink rate register is set to zero.
- Program: The blink rate register is set to a user-defined value.
- Enable: L is enabled.
- Disable: L is disabled.
- Poll rate: L returns the content of the
- Poll state: L returns the content of th
- Power-off: L terminates its operation and starts waiting for the next reset.

The enable register is set to one.
This operation can be done multiple times, during the device usage.

 © 2017 Arm Limited

arm

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- Reset: L is disabled, and the blink rate register is set to zero.
- Program: The blink rate register is set to a user-defined value.
- Enable: L is enabled.
- Disable: L is disabled.
- Poll rate: L returns the content of the blink
- Poll state: L returns the content of the
- Power-off: L terminates its operation and starts waiting for the next reset.

The enable register is set to zero.
This operation can be done multiple times, during the device usage.

 © 2017 Arm Limited

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- Reset: L is disabled, and the blink rate register is set to zero
- Program: the blink rate register is set to a user-defined value.
- Enable: L is enabled.
- Disable: L is disabled.
- Poll rate: L returns the content of the blink rate register.
- Poll state: L returns the co … f the enable register.
- Power-off: L terminates its opera …

The blink rate register content is provided to the user level.
This operation can be done multiple times, during the device usage.

 © 2017 Arm Limited

arm

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- Reset: L is disabled, and the blink rate register is set to zero.
- Program: The blink rate register is set to a user-defined value.
- Enable: L is enabled.
- Disable: L is disabled.
- Poll rate: L returns the content of the blink rate register.
- Poll state: L returns the content of the enable register.
- Power-off: L terminates operation and starts waiting for the next reset.

The enable register content is provided to the user level.
This operation can be done multiple times, during the device usage.

 © 2017 Arm Limited

arm

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

In our example, the device L supports the following functionalities:

- Reset: L is disabled, and the blink rate register is set to zero.
- Program: The blink rate register is set to a user-defined value.
- Enable: L is enabled.
- Disable: L is disabled.
- Poll rate: L returns the content of the blink rate register.
- Poll state: L returns the content of the enable register.
- Power-off: L terminates its operation and starts waiting for the next reset.

The enable register content is set to zero.
This operation is done once, after the last usage of the device.

 © 2017 Arm Limited

The Module Level

At this level, the different functionalities of the device shall be enumerated, and an association shall be established with the VFS functionalities.

The virtual file system functions that are typically used are:

- open, which initiates the operations with the device
- release, which terminates the operation with the device
- write, which sends data coming from the user space to the device
- read, which reads from the device and send them to the user space
- ioctl, which performs custom operations

© 2017 Arm Limited

The Module Level

For the considered example, the following association between device L functionalities and the VFS is proposed.

Device functionality	Virtual file system function	Notes
Reset associated with	open	open() is used once, to establish the connection with the device.
Program	write	write() is used to send data to the device. The blink rate register shall be selected as target for the write operation using the ioctl() function.
Enable	write	write() is used to send data to the device. The enable register shall be selected as target for the write operation using the ioctl() function.
Disable	write	write() is used to send data to the device. The enable register shall be selected as target for the write operation using the ioctl() function.

 © 2017 Arm Limited

arm

The Module Level

For the considered example, the following association between device L functionalities and the VFS is proposed.

Device functionality	Virtual file system function	Notes
Poll rate	read	read() is used to send data to the application. The blink rate register shall be selected as target for the read operation using the ioctl() function.
Poll state	read	read() is used to send data to the application. The enable register shall be selected as target for the read operation using the ioctl() function.
Power-off	release	release() is used to terminate the connection with the device.
None	ioctl	ioctl() is used to select the target for read/write operations.

 © 2017 Arm Limited

The Module Level: File Operations

```
static dev_t L_dev;

struct cdev L_cdev;

struct file_operations L_fops = {
    .owner      = THIS_MODULE,
    .open       = L_open_close,
    .release    = L_open_close,
    .write      = L_write,
    .read       = L_read,
    .ioctl      = L_ioctl,
};
```

Device-specific functions

VFS functions

© 2017 Arm Limited
arm

The Module Level: ioctl() Implementation

When `cmd` is set to `BLINK_RATE/ENABLE`, the blink rate/enable register is selected.

```
static ssize_t L_ioctl(struct inode *inode, struct file *filep, unsigned int cmd, unsigned long arg)
{
  switch( cmd )
  {
    case BLINK_RATE:                               // global symbol previously defined
      selected_register = BLINK_RATE;              // global variable previously defined
      break;
    case ENABLE:                                   // global symbol previously defined
      selected_register = ENABLE;
      break;
  }

  return 0;
}
```


© 2017 Arm Limited

The Module Level: open()/release() Implementation

It disables the device and sets the blink rate to zero. The same operations are valid for open() and release() functions.

```
static int L_open_close(struct inode *inode, struct file *file)
{
  selected_register = ENABLE;                    // it selects the target for read/write operation.
  WRITE_DATA_TO_THE_HW( 0 );                     // it sends 0 to the enable register.
                                                 // it abstracts the low-level CPU/device L interface.

  selected_register = BLINK_RATE;                // it selects the target for read/write operation.
  WRITE_DATA_TO_THE_HW( 0 );                     // it sends 0 to the blink register.

  return 0;
}
```

 © 2017 Arm Limited

The Module Level: read() Implementation

It reads from the ioctl() selected register and pass the data to the user.

```
static ssize_t L_read(struct file *filp, char *buffer, size_t length, loff_t * offset)
{
  int data;

  READ_DATA_FROM_THE_HW( &data );                       // it abstracts the low-level CPU/device L interface.
  copy_to_user(buffer, &data, 4 );                      // see next slide

  return 4;                                             // it returns the number of bytes read.
}
```

 © 2017 Arm Limited

Passing Data to/from the Kernel

Kernel and application are running in two different memory spaces.

Specific functions are needed to move data between them.

```
copy_to_user(void __user *to, const void *from, unsigned long n)
```

Move data from kernel space to user space.

 © 2017 Arm Limited

The Module Level: write() Implementation

It writes to the ioctl() selected register the data coming from the user.

```
static ssize_t L_write(struct file *filp, char *buffer, size_t length, loff_t * offset)
{
  WRITE_DATA_TO_THE_HW( buffer );

  return 1;
}
```


© 2017 Arm Limited

The Module Level: Communication with the Device

Hidden in

hardware

- `READ_DATA_FROM_THE_HW()`
- `WRITE_DATA_TO_HW()`

Implementation depends on the CPU/Device L connection

Memory mapped example:

- Blink rate register: 0xf0080000
- Enable register: 0xf0080004

GPIO example:

- Blink rate register: GPIO(0-31) (MSB first) → memory map
- Enable register: GPIO(32)

 © 2017 Arm Limited

arm

Memory Mapped I/O

Memory areas can be used if:

- Available
- Reserved

```
int check_region( unsigned long first, unsigned long n)
```

- It checks whether the desired addresses are available

```
int request_region( unsigned long first, unsigned long n,
const char *name)
```

- It reserves the desired addresses

```
int release_region( unsigned long first, unsigned long n)
```

- It sets the desired addresses free

 © 2017 Arm Limited

Memory Mapped I/O: Initialization

```
static int __init L_module_init(void)
{
  int res;
  alloc_chrdev_region(&L_dev, 0, 1, "L_dev");
  printk(KERN_INFO "%s\n", format_dev_t(buffer, L_dev));
  cdev_init(&L_cdev, &L_fops);
  L_cdev.owner = THIS_MODULE;
  cdev_add(&L_cdev, L_dev, 1);

  r = check_region( ioremap(0xf0080000, 4 ), 8 );
  if(r) {
   printk( KERN_ALERT "Unable to reserve I/O memory\n");
   return -EINVAL;
  }
  request_region(ioremap(0xf0080000, 4), 8, "DevL");
  return 0;
}
```

Translates the physical address of the device (as defined by the memory map) into the corresponding virtual address.

© 2017 Arm Limited

Memory Mapped I/O: Clean-up

```
static void __exit L_module_cleanup(void)
{
  cdev_del(&L_cdev);
  unregister_chrdev_region(L_dev, 1);

  release_region(ioremap(0xf0080000, 4 ), 8);
}
```

free the occupy addresses

 © 2017 Arm Limited

arm

Memory Mapped I/O: read

```
int      READ_DATA_FROM_THE_HW( int *data )
{
  int    tmp;

  switch( selected_register )
  {
    case BLINK_RATE:
      tmp = inl( ioremap(0xf0080000, 4) );
      break;
    case ENABLE:
      tmp = inl( ioremap(0xf0080000, 4)+4 );
      break;
    }
    *data = tmp;

    return 4;
}
```

Functions for accessing I/O memory:

`inb()`: it reads 8-bit words.

`inw()`: it reads 16-bit words.

`inl()`: it reads 32-bit words.

 © 2017 Arm Limited

Memory Mapped I/O: write

```
int      WRITE_DATA_TO_THE_HW( char *data )
{
  switch( selected_register )
  {
    case BLINK_RATE:
      outl( (int)*data, ioremap(0xf0080000, 4) );
      break;
    case ENABLE:
      outl( (int)*data, ioremap(0xf0080000, 4)+4 );
      break;
    }

    return 4;
}
```

Functions for accessing I/O memory:

`outb()`: it writes 8-bit words.

`outw()`: it writes 16-bit words.

`outl()`: it writes 32-bit words.

 © 2017 Arm Limited

GPIO-based I/O

Prior to GPIO use, it shall be reserved for the module.

```
int gpio_request(unsigned gpio, const char *label)
```

- It checks whether the desired GPIO is available, and if yes, reserves it.

```
void gpio_free(unsigned gpio)
```

- It sets the desired GPIO free.

 © 2017 Arm Limited

GPIO-based I/O – initialization

```
static int __init L_module_init(void)
{
  int i, r;
  alloc_chrdev_region(&L_dev, 0, 1, "L_dev");
  printk(KERN_INFO "%s\n", format_dev_t(buffer, L_dev));
  cdev_init(&L_cdev, &L_fops);
  L_cdev.owner = THIS_MODULE;
  cdev_add(&L_cdev, L_dev, 1);

  for( i = 0; i < 32; i++ ) {
    r = gpio_request( i );
    if (r) {
      printk( KERN_ALERT "Unable to reserve GPIO\n");
      return -EINVAL;
    }
  }
  return 0;
}
```

GPIO shall be checked and reserved one by one.

 © 2017 Arm Limited

arm

GPIO-based I/O: Clean-up

```
static void __exit L_module_cleanup(void)
{
  int i;

  cdev_del(&L_cdev);
  unregister_chrdev_region(L_dev, 1);

  for( i = 0; i < 32; i++ )
    gpio_free( i );
}
```

GPIO shall be freed one by one.

 © 2017 Arm Limited

arm

GPIO-based I/O: read

```
int     READ_DATA_FROM_THE_HW( int *data )
{
  int   tmp = 0, i;

  switch( selected_register )
  {
    case BLINK_RATE:
      for( i = 0; i < 31; i++ ) {
        gpio_direction_input( i );
        tmp = (tmp << 1) | gpio_get_value( i );
      }
      break;
    case ENABLE:
      gpio_direction_input( 32 );
      tmp |= gpio_get_value( 32 )
      break;
  }
  *data = tmp;

  return 4;
}
```

GPIO direction shall be set to input.

GPIO value shall be read one by one.
The resulting word is built MSB first.

most significant bit

 © 2017 Arm Limited

arm

GPIO-based I/O: write

```
int       WRITE_DATA_TO_THE_HW( int data )
{
  int    i;
  switch( selected_register ) {
    case BLINK_RATE:
      for( i = 0; i < 31; i++ ) {
        gpio_direction_output( i, 0 );
        gpio_set_value( i, (data & (1 << i)) );
      }
      break;
    case ENABLE:
      gpio_direction_output( 32, 0 );
      gpio_set_value( 32, data & 0x00000001 );
      break;
  }

  return 4;
}
```

GPIO direction shall be set to output, with default value on the GPIO set to 0.

GPIO value shall be written one by one.

© 2017 Arm Limited

Interrupts

Often, kernel modules have to react to interrupts coming from the hardware.

To handle interrupts, a Kernel module shall:

- Request an interrupt line
- Associate an interrupt handler to an interrupt line
- Implement the interrupt handler

When finished with the device and unregistering the driver, the Kernel module shall free the interrupt line.

 © 2017 Arm Limited

Requesting the Interrupt Line

```
int request_irq(
  unsigned int irq,
  irqreturn_t (*hanlder)(),
  unsigned long flags,
  const char *dev_name,
  void *dev_id
);
```

Interrupt line to manage

Function pointer to the interrupt handler

Options to be used when associating the interrupt handler to the interrupt line

Name of the module requesting the interrupt

Pointer to a user-defined structure containing device-specific data. It can be NULL.

© 2017 Arm Limited

arm

Freeing the Interrupt Line

```
int free_irq(
  unsigned int irq,
  void *dev_id
);
```

Interrupt line to manage

Pointer to a user-defined structure containing device-specific data. It can be NULL. and free the interrupt line

 © 2017 Arm Limited

arm

The Interrupt Handler

```
static irqreturn_t hlr( int irq, void *dev_id )
{
      /*
    * Do something to handle the interrupt *
    */

  return IRQ_RETVAL(1);
}
```

 © 2017 Arm Limited

Interrupt Handling

Interrupt handlers may introduce long latencies.

- Lower-priority interrupts have to wait.
- If interrupts are disabled, everyone has to wait.

Rule of thumb: Keep interrupt handlers as short as possible.

 © 2017 Arm Limited

Top-half and Bottom-half

Split interrupt handling in two parts

Top-half

- Manages the interaction with real hw
- Does the minimum amount of work
- Keep the other events pending for the least amount of time

Bottom-half

- Processes the data coming from hw
- It can be interrupted.

 © 2017 Arm Limited

Needed Support

The idea

- Create a “process” that waits for incoming data.
- The “process” sleeps until a new data is ready.
- The interrupt handler prepares the new data and dispatches it to the waiting “process”.

Benefits

- The interrupt handler is very short → low latency for process
- The “process” can be interrupted → low latency

© 2017 Arm Limited

Work Queue

General structure in the Linux Kernel

A work queue is a list of activities to be executed.

Each activity is defined by

- Work: data to be processed
- Callback: function to process the work

API to manage work queues

```
struct workqueue_struct *create_workqueue( static char * );

void destroy_workqueue( struct workqueue_struct * );

int flush_workqueue( struct workqueue_struct * );
```


© 2017 Arm Limited

Work Queue

The work is defined using the `work_struct` structure.

- Typically, the first element of a user-defined structure storing the actual data to be processed by the callback

The callback is a generic C function.

API for work management:

```
INIT_WORK( work, func )

int queue_work( struct workqueue_struct *, struct work_struct * )
```

 © 2017 Arm Limited

Summary

Introduction

The reference use case

The module-level point of view

The user-level point of view

© 2017 Arm Limited

The User Level

At this level, the application invokes the VFS calls to implement the intended behavior.

The mapping between VFS functions and custom hardware functionalities is known and exploited to implement the desired behavior.

For the considered example, the application shall

- Open the connection with the device
- Set the desired blinking rate
- Enable the device
- Adjust the blinking rate (if needed)
- Disable/enable the device (if needed)
- Close the connection with the device

 © 2017 Arm Limited

The User Level: Application

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

Header files for the needed functions prototypes/data types

```
int main(int argc, char **argv){
  char *app_name = argv[0];
  char *dev_name = "/dev/devL";
  int fd = -1;
  int x, c;
```

actual code ที่ถูก set หลาย variable ไว้แล้ว

Variables needed for the operations of the application

 © 2017 Arm Limited

arm

The User Level: Application

It opens the device file using the `open()` system call.
File is opened as read/write.

```
/*
 * Open the sample device RD | WR
 */
if ((fd = open(dev_name, O_RDWR)) < 0) {
  fprintf(stderr, "%s: unable to open %s: %s\n", app_name, dev_name, strerror(errno));

  return( 1 );
}
```

As a result, the module initializes the device L, whose blink rate register is now zero, and disables it.

 © 2017 Arm Limited

arm

The User Level: Application

```
 x = ioctl(fd, BLINK_RATE, 0);          // it selects the blink rate register.
                                        // BLINK_RATE is a global symbol defined with the same value
                                        // used in the loadable kernel module.
  c = 25;
  x = write(fd, &c, 4 );                // it writes 25 in the blink rate register - 4 bytes

  x = ioctl(fd, ENABLE, 0);             // it selects the enable register.
  c = 1;
  x = write(fd, &c, 4 );                // it enables the device.

  if (fd >= 0) {                        // it closes the connection with the device.
    close(fd);
  }
  return( 0 );
}
```


© 2017 Arm Limited