arm

Embedded Linux

Building a Ranging Sensor Kernel Module

Goal

To discuss user-level communication with kernel modules and to present an example to support a hardware device connected to the UDOO-NEO board

Summary

Introduction

The sysfs file system

The HC-SR04 ultrasonic ranging sensor

Building Linux support for the HC-SR04 sensor

Summary

Introduction

The sysfs file system

The HC-SR04 ultrasonic ranging sensor

Building Linux support for the HC-SR04 sensor

Introduction

The communication between user level and module level can be implemented through

- The virtual file system (VFS) interface, where the user-level application invokes VFS API to access the device file
- Through a RAM-based filesystem known as `sysfs` that allows exporting kernel data structures to the user level

Summary

Introduction

The sysfs file system

The HC-SR04 ultrasonic ranging sensor

Building Linux support for the HC-SR04 sensor

The sysfs File System

RAM-based file system containing directories/files that are created by the Linux Kernel

Each directory/file contains information about portions of the Kernel that are set visible to the user.

- The content is not defined by any specific APIs.
- Kernel developers can export any information that is needed.

```
/sysfs
  /block
  /bus
  /class
  /dev
  /devices
  /firmware
  /fs
  /fsl_opt
  /kernel
    /config
    /debug
        gpio
        …
    /fscaps
    …
  /module
  /power
```

```
GPIOs 0-31, platform/209c000.gpio, 209c000.gpio:
gpio-9   (usb_otg1_vbus        ) out lo

GPIOs 32-63, platform/20a0000.gpio, 20a0000.gpio:
gpio-44  (wlan-en-regulator    ) out lo
gpio-49  (kim                  ) out lo

GPIOs 64-95, platform/20a4000.gpio, 20a4000.gpio:
gpio-66  (time_sync            ) out lo

GPIOs 96-127, platform/20a8000.gpio, 20a8000.gpio:
gpio-108 (usb_otg2_vbus        ) out lo

GPIOs 128-159, platform/20ac000.gpio, 20ac000.gpio:
gpio-132 (phy-reset            ) out lo

GPIOs 160-191, platform/20b0000.gpio, 20b0000.gpio:
gpio-160 (led0                 ) out lo
gpio-162 (2194000.usdhc cd     ) in  lo

GPIOs 192-223, platform/20b4000.gpio, 20b4000.gpio:
```

The sysfs File System

Users can read/write the Kernel objects exported through `sysfs`.

Depending on the specific purpose of the kernel object, read/write operations corresponds to object-specific behaviors.

Example `/sys/class/gpio`

- It provides user-level access to general purpose I/Os (GPIO).
- It contains
  - Two control files, `export` and `unexport`, to decide which GPIO is accessible to the user
  - One directory for each user-accessible GPIO, storing the `direction` file whose content defines whether the GPIO is an input or an output, and a `value` file whose content is the value to be written to the output GPIO or the value read from an input GPIO

The sysfs File System: Controlling GPIOs

The user can write to GPIOs by

- Exporting the GPIO, e.g., 105: `echo 105 > /sys/class/gpio/export`
- Setting the direction: `echo "out" > /sys/class/gpio/gpio105/direction`
- Writing the desired value: `echo 1 > /sys/class/gpio/gpio105/value`

The user can read from GPIOs by

- Exporting the GPIO, e.g., 105: `echo 105 > /sys/class/gpio/export`
- Setting the direction: `echo "in" > /sys/class/gpio/gpio105/direction`
- Reading the GPIO value: `cat /sys/class/gpio/gpio105/value`

Adding Entries to the sysfs File System

Entries that can be added to the sysfs file system are the directory and files.

To add a new directory to the sysfs file system:

- A new Kernel object shall be defined using the `kobject` data structure.

```
struct kobject {
  char                 *k_name;
  char                 name[KOBJ_NAME_LEN];
  struct kref          kref;
  struct list_head     entry;
  struct kobject       *parent;
  struct kset          *kset;
  struct kobj_type     *ktype;
  struct dentry        *dentry;
};
```

- The object shall be added to the file system using the `kobject_create_and_add()` function.
- When no longer needed, the above shall be destroyed using the `kobject_put()` function.

Adding Entries to the sysfs File System

To add a new file to the sysfs file system within a directory corresponding to a Kernel object

- A new Kernel object attribute shall be defined using the `kobj_attribute` data structure.

```
struct kobj_attribute {
  struct attribute attr;
  ssize_t (*show)(struct kobject *kobj,
                  struct kobj_attribute *attr,
                  char *buf);
  ssize_t (*store)(struct kobject *kobj,
                   struct kobj_attribute *attr,
                   const char *buf,
                   size_t count);
};
```

Function executed when the file is read

Function executed when the file is written

- The new file shall be added using the `sysfs_create_file()` function.

arm

Using sysfs and VFS API

The usage of sysfs and the VFS API depends on the designer intentions.

The sysfs file system can be used to provide device-specific/or subsystem-specific functionalities that cannot be mapped easily into the VFS API.

- For example, to select and program a desired GPIO or to select the clock frequency scaling behavior

The sysfs file system and the VFS API can be used concurrently to satisfy different purposes, for example

- The VFS API is used to communicate with a specific device modeled as a file.
- The sysfs file system is used to provide debug information.

In the following slides, the above concepts will be put to work to support a specific hardware device: the HC-SR04 ultrasonic ranging sensor.

Summary

Introduction

The sysfs file system

The HC-SR04 ultrasonic ranging sensor

Building Linux support for the HC-SR04 sensor

The HC-SR04 Ultrasonic Ranging Sensor

The HC-SR04 is a sensor able to measure the distance of objects using ultrasounds.

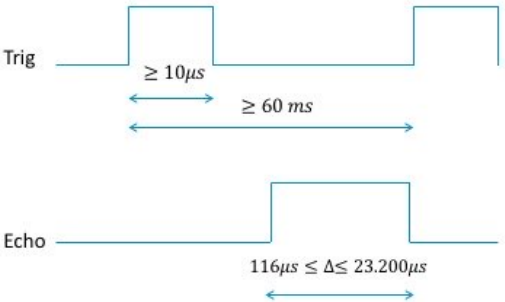
It provides the following connectors:

- Vcc, 5V power supply input
- GND, ground input
- Trig, TTL input to trigger the operation of the sensor
- Echo, TTL output with a pulse-modulated square waveform providing the distance readout

arm

The HC-SR04 Ultrasonic Ranging Sensor

The sensor operates according to the following time diagram.



- To enable the sensor, a pulse shall be generated on the Trig input;
- The pulse shall be at least 10 ms long;
- At least 60 ms shall separate two consecutive trigger pulses.

- The sensor provides the distance readout as Echo pulse with duration between 116 ms (2 cm) and 23,200 ms (400 cm).
- Assuming the pulse duration D is measured in ms, the corresponding distance D in cm is obtained as $D = \frac{\Delta}{58}$

Summary

Introduction

The sysfs file system

The HC-SR04 ultrasonic ranging sensor

Building Linux support for the HC-SR04 sensor

Building Linux Support for the HC-SR04 Sensor

We will develop Linux support for HC-SR04 sensor that makes use of both VFS API and sysfs file system.

A loadable Kernel module will use the VFS API as follows:

- The write() system call will trigger the sensor, and it will measure the echo pulse duration. คำนวณหาระยะต่างๆที่วัดจาก sensor
- The read() system call will provide to the user level the measured echo pulse duration in microseconds. รับค่าที่คำนวณเป็น ms ให้ user level

The same loadable Kernel module will use the sysfs to record and provide the user the last echo pulse duration read from the sensor.

Module Data Structure Definition

```
static dev_t hcsr04_dev;
struct cdev hcsr04_cdev;
```

Data structures for a new character-based device

```
static int hcsr04_lock = 0;
```

Module usage flag

```
static struct kobject *hcsr04_kobject;
```

Kernel object for adding a directory entry to sysfs

```
static ktime_t rising,
               falling;
```

Data structures to keep the time of the echo pulse rising edge/falling edge

```
static struct kobj_attribute hcsr04_attribute =
__ATTR(hcsr04, 0660, hcsr04_show, hcsr04_store);
```

Kernel object attribute

Name of the file

File access right

Functions invoked when the file is read/written

Module Data Structure Definition

```
struct file_operations hcsr04_fops = {
  .owner = THIS_MODULE,
  .read = hcsr04_read,
  .write = hcsr04_write,
  .open = hcsr04_open,
  .release = hcsr04_close,};
```

File operations for the kernel module corresponding to the VFS API

Module Initialization Function

```
static int __init hcsr04_module_init(void)
{
  char buffer[64];

  alloc_chrdev_region(&hcsr04_dev, 0, 1, "hcsr04_dev");
  printk(KERN_INFO "%s\n", format_dev_t(buffer, hcsr04_dev));
  cdev_init(&hcsr04_cdev, &hcsr04_fops);
  hcsr04_cdev.owner = THIS_MODULE;
  cdev_add(&hcsr04_cdev, hcsr04_dev, 1);
  gpio_request( GPIO_OUT, "hcsr04_dev" );
  gpio_request( GPIO_IN, "hcsr04_dev" );
  gpio_direction_output( GPIO_OUT, 0 );
  gpio_direction_input( GPIO_IN );
  hcsr04_kobject = kobject_create_and_add("hcsr04", kernel_kobj);
  sysfs_create_file(hcsr04_kobject, &hcsr04_attribute.attr);
  return 0;
}
```


Module Initialization Function

```
static int __init hcsr04_module_init(void)
{
  char buffer[64];

  alloc_chrdev_region(&hcsr04_dev, 0, 1, "hcsr04_dev");
  printk(KERN_INFO "%s\n", format_dev_t(buffer, hcsr04_dev));
  cdev_init(&hcsr04_cdev, &hcsr04_fops);
  hcsr04_cdev.owner = THIS_MODULE;
  cdev_add(&hcsr04_cdev, hcsr04_dev, 1);
  gpio_request( GPIO_OUT, "hcsr04_dev" );
  gpio_request( GPIO_IN, "hcsr04_dev" );
  gpio_direction_output( GPIO_OUT, 0 );
  gpio_direction_input( GPIO_IN );
  hcsr04_kobject = kobject_create_and_add("hcsr04", kernel_kobj);
  sysfs_create_file(hcsr04_kobject, &hcsr04_attribute.attr);
  return 0;
}
```

Insert the new character device in the Linux Kernel.

Module Initialization Function

```
static int __init hcsr04_module_init(void)
{
  char buffer[64];

  alloc_chrdev_region(&hcsr04_dev, 0, 1, "hcsr04_dev");
  printk(KERN_INFO "%s\n", format_dev_t(buffer, hcsr04_dev));
  cdev_init(&hcsr04_cdev, &hcsr04_fops);
  hcsr04_cdev.owner = THIS_MODULE;
  cdev_add(&hcsr04_cdev, hcsr04_dev, 1);
  gpio_request( GPIO_OUT, "hcsr04_dev" );
  gpio_request( GPIO_IN, "hcsr04_dev" );
  gpio_direction_output( GPIO_OUT, 0 );
  gpio_direction_input( GPIO_IN );
  hcsr04_kobject = kobject_create_and_add("hcsr04", kernel_kobj);
  sysfs_create_file(hcsr04_kobject, &hcsr04_attribute.attr);
  return 0;
}
```

Reserve two GPIOs for the character device, one as output for the Trig signal and one as input for the Echo signal.

Module Initialization Function

```
static int __init hcsr04_module_init(void)
{
  char buffer[64];

  alloc_chrdev_region(&hcsr04_dev, 0, 1, "hcsr04_dev");
  printk(KERN_INFO "%s\n", format_dev_t(buffer, hcsr04_dev));
  cdev_init(&hcsr04_cdev, &hcsr04_fops);
  hcsr04_cdev.owner = THIS_MODULE;
  cdev_add(&hcsr04_cdev, hcsr04_dev, 1);
  gpio_request( GPIO_OUT, "hcsr04_dev" );
  gpio_request( GPIO_IN, "hcsr04_dev" );
  gpio_direction_output( GPIO_OUT, 0 );
  gpio_direction_input( GPIO_IN );
  hcsr04_kobject = kobject_create_and_add("hcsr04", kernel_kobj);
  sysfs_create_file(hcsr04_kobject, &hcsr04_attribute.attr);
  return 0;
}
```

Add the hcsr04 directory in /sys/kernel.

Add the hcsr04 file in /sys/kernel/hcsr04.

ultrasonic

arm

Module Clean-up Function

```
static void __exit hcsr04_module_cleanup(void)
{
  gpio_free( GPIO_OUT );
  gpio_free( GPIO_IN );

  hcsr04_lock = 0;

  cdev_del(&hcsr04_cdev);
  unregister_chrdev_region( hcsr04_dev, 1 );

  kobject_put( hcsr04_kobject );
}
```

Release the used GPIOs.

Mark the module as free.

Remove the character device from the kernel.

Remove the hcsr04 directory from sysfs.

Module Open Function

```
int hcsr04_open(struct inode *inode, struct file *file)
{
  int ret = 0;

  if( hcsr04_lock > 0 )
    ret = -EBUSY;
  else
    hcsr04_lock++;

  return( ret );
}
```

Make sure that only one application at a time can use the device.

implement loch variable เพื่อ device ถูกใช้งานอยู่

arm

Module Close Function

```
int hcsr04_close(struct inode *inode, struct file *file)
{
  hcsr04_lock = 0;

  return( 0 );
}
```

Set the device free to be used.

arm

Module Write Function

```
ssize_t hcsr04_write(struct file *filp, const char *buffer, size_t length, loff_t * offset)
{
  gpio_set_value( GPIO_OUT, 0 );
  gpio_set_value( GPIO_OUT, 1 );
  udelay( 10 );
  gpio_set_value( GPIO_OUT, 0 );
```

Generate a pulse on the output connected to Trig. After setting the output to 1, we wait for 10 µs, and then we return the output to 0.

```
  while( gpio_get_value( GPIO_IN ) == 0 )
    ;
  rising = ktime_get();
```

We wait as long as the input connected to Echo is 0. We then record the kernel time when the rising edge occurred.

```
  while( gpio_get_value( GPIO_IN ) == 1 )
    ;
  falling = ktime_get();
```

We wait as long as the input connected to Echo is 1. We then record the kernel time when the falling edge happened.

```
  return( 1 );
}
```

Module Read Function

```
ssize_t hcsr04_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
  int ret;
  int pulse;

  pulse = (int)ktime_to_us( ktime_sub( falling, rising ) );

  ret = copy_to_user( buf, &pulse, 4 );

  return 4;
}
```

The pulse duration is computed subtracting rising time from falling time, and then translating the result in µs.
`ktime_sub()` and `ktime_to_us()` are used to handle the specific data structure used to store the Kernel time measured using `ktime_get()`.

We provide to the user space the four bytes storing the pulse duration represented as integer number.

Four bytes are read.

Module Show and Store Function

```
static ssize_t hcsr04_show(struct kobject *kobj,
                           struct kobj_attribute *attr,
                           char *buf)
{
  return sprintf(buf, "%d\n", ktime_to_us(ktime_sub(falling,rising)));
}
```

Function executed when the user reads /sys/kernel/hcsr04/hcsr04

```
static ssize_t hcsr04_store(struct kobject *kobj,
                            struct kobj_attribute *attr,
                            char *buf,
                            size_t count)
{
  return 1;
}
```

Function executed when the user writes /sys/kernel/hcsr04/hcsr04

Module Test Application

```
int main(int argc, char **argv)
{
  char *app_name = argv[0];
  char *dev_name = "/dev/hcsr04";
  int fd = -1;
  char c;
  int d;

  fd = open(dev_name, O_RDWR);
  c = 1;
  write( fd, &c, 1 );
  read( fd, &d, 4 );

  printf( "%d: %f\n", d, d/58.0 );
  close( fd );
  return 0;
}
```

We assume that the module is loaded and that the /dev/hcsr04 device file has been created.

We open the device file.

We trigger the sensor by executing the write system call. The written value is meaningless.

We read the four bytes storing the echo pulse duration.

We display the duration and the corresponding distance.

We close the device file.