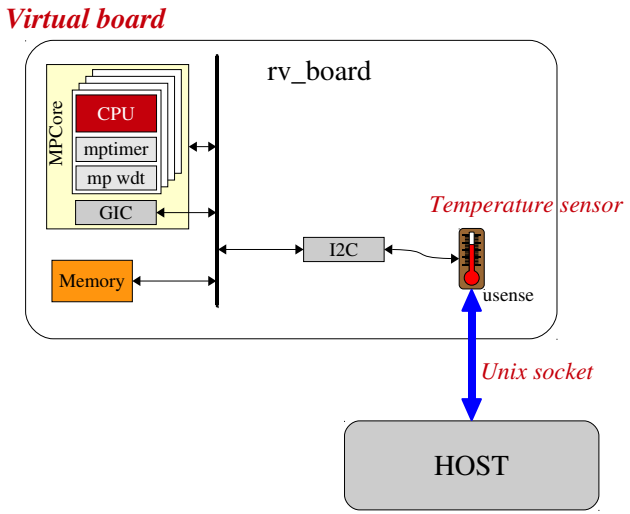
Calcolatori Elettronici e Sistemi Operativi

I2C driver for a temperature sensor

HW



Device

I2C temperature sensor (virtual)

- connected to the i2c adapter of the rv_board
- i2c address: **0x57**
- temperature resolution: 0.1 °C
- data is sent as a 2-byte signed value (LSB first)
- default temperature read: 25 °C
  - to change the "environment" temperature:
    - send the string "**TEMP:**<*value*>" to the Unix socket "usense-sckt"
    - value is a number in the [-100.0 , 100.0] range

Driver

Register the device

- *not performed in the board initialization code*

Register a new i2c driver: **usense**

On driver binding register a new char device

- *reading from char device returns the sensor temperature*
- sensor is read with the function provided by the i2c subsystem:
  - i2c_smbus_read_byte_data
- max 1 read operation / sec.
  - i2c peripherals are slow

usense.c: file structure

- *Headers and macros*
- *Function prototypes*
- *Device structure definition*
- *Globals and module parameters*
- *Chardev interface (functions and structure)*
- *I2c interface (funcions and structure)*
- *Module initialization and cleanup*
- *Module authorship and license*

usense.c: headers and macros

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/i2c.h>
#include <linux/string.h>

#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>

#define USENSE_I2C_ADDRESS      0x57

#define MODNAME                 "usense"
```

usense.c

usense.c: function prototypes

```
static
int usense_open(struct inode *inode, struct file *file);

static
int usense_release(struct inode *inode, struct file *file);

static
ssize_t usense_read(struct file *file, char __user *buffer,
                    size_t count, loff_t *offset);

static
int usense_read_temperature(struct i2c_client *client);
```

usense.c

usense.c: device structure definition

```
struct usense_device_descr {
    struct mutex        mutex;
    struct cdev         cdev;
    int                 major;
    struct i2c_client   *client;
    int                 last_temperature;
    unsigned long       last_read_time;
    int                 read;
    int                 first_read_req;
};

static struct usense_device_descr usense_device;
```

usense.c

usense.c: module initialization and cleanup

```
/* if HW info and registration is done in arch_initcall just use:
module_i2c_driver(usense_driver);
*/
static struct i2c_board_info usense_i2c_board_info = {
    I2C_BOARD_INFO("usense", USENSE_I2C_ADDRESS)
};
static struct i2c_client *usense_client;


    /* Module init */
static
int __init usense_init_module(void)
{
    struct i2c_adapter *adapter = i2c_get_adapter(0);
    int ret;

    if (!adapter) {
        pr_err(MODNAME ": Error getting i2c adapter\n");
        /* equivalent to: printk(KERN_ERR MODNAME ... */
        ret = -ENODEV;
        goto exit1;
    }

    usense_client = i2c_new_device(adapter, &usense_i2c_board_info);
```

usense.c

usense.c: module initialization and cleanup

```
    if (!usense_client) {
        pr_err(MODNAME ": Error registering i2c device\n");
        ret = -ENODEV;
        goto exit2;
    }

    ret = i2c_add_driver(&usense_driver);
    if (ret < 0) {
        goto exit3;
    }

    i2c_put_adapter(adapter);
    return 0;

exit3:
    i2c_unregister_device(usense_client);
exit2:
    i2c_put_adapter(adapter);
exit1:
    return ret;
}
```

usense.c

usense.c: module initialization and cleanup

```
    /* Module cleanup */
static
void __exit usense_cleanup_module(void)
{
    i2c_del_driver(&usense_driver);
    i2c_unregister_device(usense_client);
}


module_init(usense_init_module);
module_exit(usense_cleanup_module);
```

usense.c

usense.c: chardev interface

```
static struct file_operations usense_fops = {
     .owner    =   THIS_MODULE,
     .open     =   usense_open,
     .release =   usense_release,
     .read     =   usense_read,
};
```

usense.c

usense.c: chardev interface

```
static
int usense_open(struct inode *inode, struct file *file)
{
    struct usense_device_descr *dev;
    dev = container_of(inode->i_cdev, struct usense_device_descr, cdev);

    file->private_data = dev;
    dev->first_read_req = 1;

    return 0;
}


static
int usense_release(struct inode *inode, struct file *file)
{
    return 0;
}
```

usense.c

usense.c: chardev interface

```
    /* Read */
static
ssize_t usense_read(struct file *file, char __user *buffer,
                    size_t count, loff_t *offset)
{
    struct usense_device_descr *dev = file->private_data;
    ssize_t ret = 0;
    int temperature;
    static char buff[10];
    int datalen;

    if (mutex_lock_interruptible(&dev->mutex)) return -ERESTARTSYS;

    if (!dev->read || dev->last_read_time + HZ < jiffies) {
        /* update data */
        temperature = usense_read_temperature(dev->client);
        dev->last_temperature = temperature;
        dev->last_read_time = jiffies;
        dev->read = 1;
    }
```

usense.c

usense.c: chardev interface

```
    if (dev->first_read_req) {
        sprintf(buff, "%d.%d\n", dev->last_temperature / 10,
                                 dev->last_temperature % 10);
        datalen = strlen(buff);
        if (copy_to_user(buffer, buff, datalen)) {
            ret = -EFAULT;
        } else {
            *offset += datalen;
            ret = datalen;
            dev->first_read_req = 0;
        }
    } else {
        /* signal EOF */
        ret = 0;
        dev->first_read_req = 1;
    }

    mutex_unlock(&dev->mutex);
    return ret;
}
```

usense.c

usense.c: i2c interface

```
static const struct i2c_device_id usense_id[] = {
    { "usense",             /* name */
      USENSE_I2C_ADDRESS    /* driver_data: data private to the driver */
    },
     { } /* empty */
};
MODULE_DEVICE_TABLE(i2c, usense_id);


static struct i2c_driver usense_driver = {
     .driver = {
         .name   = "usense",
         .owner  = THIS_MODULE,
     },
     .probe      = usense_probe,
     .remove     = usense_remove,
     .id_table   = usense_id,
};
```

usense.c

usense.c: i2c interface

```
/* called by kernel: just register a char interface to query the usense device */
static
int usense_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
    dev_t devid = 0;
    int err, devno;
    struct usense_device_descr *dev;

    if (client->addr != id->driver_data) {
        pr_err(MODNAME ": wrong address (is %d)\n", client->addr);
        return -ENODEV;
    }

    memset(&usense_device, 0, sizeof(usense_device));
    dev = &usense_device;       /* or use dynamic allocation */
    i2c_set_clientdata(client, dev);

    dev->client = client;
```

usense.c

usense.c: i2c interface

```
    err = alloc_chrdev_region(&devid, 0 /* minor */, 1, MODNAME);
    dev->major = MAJOR(devid);
    if (err < 0) {
        pr_warning(KERN_WARNING MODNAME ": can't get major %d\n", dev->major);
        return err;
    }
    devno = MKDEV(dev->major, 0);

    mutex_init(&dev->mutex);
    cdev_init(&dev->cdev, &usense_fops);

    err = cdev_add(&dev->cdev, devno, 1);
    if (err) {
        /* registration failed */
        pr_err(MODNAME ": Error %d adding device\n", err);
        unregister_chrdev_region(devno, 1);
        return err;
    }

    return 0;
}
```

usense.c

usense.c: i2c interface

```
static
int usense_remove(struct i2c_client *client)
{
    struct usense_device_descr *dev = i2c_get_clientdata(client);
    int devno;

    if (dev) {
        i2c_set_clientdata(client, NULL);
        cdev_del(&dev->cdev);
        devno = MKDEV(dev->major, 0 /* minor */);
        unregister_chrdev_region(devno, 1);
    }
    return 0;
}
```

usense.c

usense.c: i2c interface

```
static
int usense_read_temperature(struct i2c_client *client)
{
    int ret;
    int temperature;
    int8_t t;

    ret = i2c_smbus_read_byte_data(client, 0);  /* temperature_lo */
    if (ret < 0) {
        pr_warn(MODNAME ": Error %d reading from device\n", ret);
        return -1;
    }
    temperature = (int8_t)ret;

    ret = i2c_smbus_read_byte_data(client, 1);  /* temperature_hi */
    if (ret < 0) {
        pr_warn(MODNAME ": Error %d reading from device\n", ret);
        return -1;
    }
    t = (int8_t)ret;
    temperature |= (t << 8);

    return temperature;
}
```

usense.c

usense.c: module authorship and license

```
MODULE_AUTHOR("Calcolatori Elettronici e Sistemi Operativi (uniud)");
MODULE_DESCRIPTION("Example for i2c");
MODULE_VERSION("1.0");
MODULE_LICENSE("Dual BSD/GPL");
```

usense.c

Makefile

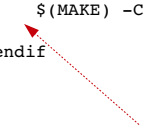
```
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)
     obj-m := usense.o

# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else

     KERNELDIR ?= /lib/modules/$(shell uname -r)/build
     PWD := $(shell pwd)

default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif
```



Tabulation (mandatory)

Makefile

changetemp.c

```c
#define SOCKETNAME          "usense-sckt"

#define COMMAND_TEMP        "TEMP:"
#define COMMAND_TEMP_LEN    sizeof(COMMAND_TEMP)

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```


changetemp.c

```c
#define TIMEOUT 10


int main(int argc, char **argv)
{
    char *socketname = SOCKETNAME;
    int main_socket;
    struct sockaddr_un address_un;
    char buf[512] = COMMAND_TEMP;
    int count;
    int i;
    int res;

    if (argc > 1) {
        strncat(buf, argv[1], sizeof(buf) - COMMAND_TEMP_LEN - 1);
    } else {
        return 1;
    }
    count = strlen(buf);

    main_socket = socket(PF_UNIX, SOCK_DGRAM, 0);
    if (main_socket < 0) return -1;
```


changetemp.c

```c
    address_un.sun_family = AF_UNIX;
    memset(address_un.sun_path, 0, sizeof(address_un.sun_path));

    /* use the unix abstract namespace */
    strncpy(address_un.sun_path+1, socketname, sizeof(address_un.sun_path)-1-1);


    for (i = 0 ; i < TIMEOUT ; i++) {
        if (connect(main_socket, (void*)&address_un, sizeof(address_un)) >= 0) {
            break;
        } else {
            sleep(1);
        }
    }
    if (i == TIMEOUT) {
        return -1;
    }
```


changetemp.c

```c
    for (i = 0 ; i < count ; i += res) {
        res = write(main_socket, ((char*)buf) + i, count - i);
        if (res <= 0) {
            if ((errno == EAGAIN) || (errno == ENOMEM) || (errno == EINTR)) {
                continue;
             } else {
                close(main_socket);
                return res;
            }
        }
    }

    close(main_socket);

    return 0;
}
```