Driver I2C

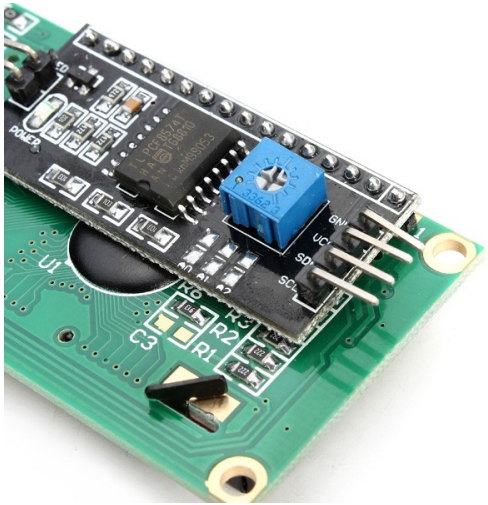
Alunos:
Igor dos Santos
Luis Fernando Segalla

Objetivos

- Fazer a comunicação entre um Raspberry Pi 3 e uma tela LCD usando o protocolo I2C.

Materiais

Display LCD



Porta expansora
pcf 8574

Raspberry Pi 3
Modelo b

i2cDriverSO_t

```
typedef struct i2cDriverSO
{
    struct i2c_client *meuCliente;
    struct mutex       meuMutex;
    struct cdev        meuCdev;

    u8                 regs_cntrl;
    u8                 linha;
    u8                 coluna;
}i2cDriverSO_t;
```

Macros

```
#define LCD_ADDRESS             0x27

#define LCD_CHR                 0x01
#define LCD_CMD                 0x00

#define LCD_LINE0               0x80
#define LCD_LINE1               0xC0

#define LCD_ENABLE              0x04
#define LCD_CLEAR               0x01
#define LCD_HOME                0x02
#define LCD_BACKLIGHT           0x08


// Valores para ioctl (switch-case)
#define INICIA_TELA             10
#define BACKLIGHT               11
#define CURSOR                  12
#define BLINK                   13

#define CURSOR_ADDR             1
#define BLINK_ADDR              0
```

Structs e definições

```
int meuMajor = 0;
int meuMinor = 0;

//Funções utilitárias
void write_command(struct i2c_client *client, u8 data, int instr);
void display_cursor(i2cDriverSO_t *display, u8 value);
void display_blink(i2cDriverSO_t *display, u8 value);
void setBacklight(i2cDriverSO_t *display, u8 value);

//Estruturas e funções referentes a parte i2c do problema

//struct necessária para o funcionamento do driver
static i2cDriverSO_t      *display;//representação interna do nosso driver

static struct i2c_device_id meuIdTable[] =
{
    {DRIVER_NAME,LCD_ADDRESS},
    {},
};
MODULE_DEVICE_TABLE(i2c, meuIdTable);
```

Structs e definições

```
static int meuProbe(struct i2c_client *client, const struct i2c_device_id *id);
static int meuRemove(struct i2c_client *client);


static struct i2c_driver meuDriveri2c =
{
    .probe      = meuProbe,
    .remove     = meuRemove,
    .id_table   = meuIdTable,
    .driver =
    {
        .owner      = THIS_MODULE,
        .name       = DRIVER_NAME,
    }
};

//Estruturas e funções referente a parte de device drivers

static ssize_t meuWrite(struct file *filp, const char __user *buff, size_t count, loff_t *offp);
static ssize_t meuRead(struct file *filp, char __user *buff, size_t count, loff_t *offp);
static int meuOpen(struct inode *inode, struct file *filp);
static int meuRelease(struct inode *inode, struct file *filp);
static long meuIoctl(struct file *f, unsigned int cmd, unsigned long arg);
```

Struct file_operations

```
struct file_operations meuFile_operations =
{
    .owner                  = THIS_MODULE,
    .write                  = meuWrite,
    .read                   = meuRead,
    .open                   = meuOpen,
    .release                = meuRelease,
    .unlocked_ioctl         = meuIoctl,
};
```

meuInit

```c
static int __init meuInit(void)
{
    int result;
    struct i2c_adapter *meuAdapter;//representa meu barramento
    struct i2c_client  *meuCliente;//representa o dispositivo slave
    dev_t dev;//serve para achar os minor e major numbers
    struct i2c_board_info meuBoardinfo = { I2C_BOARD_INFO(DRIVER_NAME, LCD_ADDRESS) };;//template para a criação do device
    printk(KERN_ALERT "INICIALIZANDO.\n");

    //inicializando a parte do device driver

    result = alloc_chrdev_region(&dev,meuMinor,1,DRIVER_NAME);
    meuMajor = MAJOR(dev);

    if(result < 0 )
    {
        printk(KERN_ALERT "ERRO AO ALOCAR O MMAJOR NUMBER.\n");
        return result;
    }

    //inicializando a parte de i2c

    meuAdapter = i2c_get_adapter(1);//tenta pegar o barramento 1
    if(!meuAdapter)
    {
        printk(KERN_ALERT "ERRO AO ALOCAR O meuAdapter.\n");
        unregister_chrdev_region(dev,1);
        return -EINVAL;
    }
```

meuInit

```
    meuCliente = i2c_new_device(meuAdapter, &meuBoardinfo);//cria dispositivo i2c
    printk(KERN_ALERT "LOGO APÓS A CHAMADA DE i2c_new_device\n");//
    if(!meuCliente)
    {
        printk(KERN_ALERT "ERRO AO ALOCAR O meuCliente.\n");
        unregister_chrdev_region(dev,1);
        return -EINVAL;
    }

    result = i2c_add_driver(&meuDriveri2c);
    printk(KERN_ALERT "LOGO APÓS A CHAMADA DE i2c_add_driver\n");//
    if(result < 0)
    {
        printk(KERN_ALERT "ERRO AO ADICIONAR o display.\n");
        unregister_chrdev_region(dev,1);
        i2c_unregister_device(meuCliente);
        return -EINVAL;
    }

    IniciaTela(display);

    return 0;
}
```

meuProbe

```
static int meuProbe(struct i2c_client *client, const struct i2c_device_id *id)
{
    int out;
    int number = MKDEV(meuMajor,meuMinor);

    printk(KERN_ALERT "DENTRO DA FUNÇÃO meuProbe.\n");

    if ( !i2c_check_functionality(client->adapter, I2C_FUNC_SMBUS_BYTE_DATA) )//checo se ex
    {
        printk(KERN_ALERT "FALHOU NO TESTE i2c_check_functionality.\n");
        return -EIO;
    }

    display = (i2cDriverSO_t *)devm_kzalloc(&client->dev,sizeof(i2cDriverSO_t),GFP_KERNEL);
    if(!display)
    {
        printk(KERN_ALERT "FALHOU NA HORA DE ALOCAR MEMÓRIA PARA O DRIVER.\n");
        return -ENOMEM;
    }
```

meuProbe

```
    mutex_lock(&display->meuMutex);

    //Inicializando o cdev igual ao exemplo do linux device drivers
    cdev_init(&display->meuCdev, &meuFile_operations);
    display->meuCdev.owner = THIS_MODULE;
    display->meuCdev.ops   = &meuFile_operations;
    out = cdev_add(&display->meuCdev, number,1);

    if(out < 0)
    {
        printk(KERN_ALERT "FALHOU NA HORA DE ADICIONAR O DRIVER COM cdev_add.\n");
        return -ENOMEM;
    }

    display->meuCliente     = client;
    display->regs_cntrl     = 0x0C;
    display->linha          = 0;
    display->coluna         = 0;

    i2c_set_clientdata(client,display);

    mutex_unlock(&display->meuMutex);

    printk(KERN_ALERT "PASSOU POR TODOS OS PASSOS SEM PROBLEMAS. RETORNANDO. \n");
    return 0;
}
```

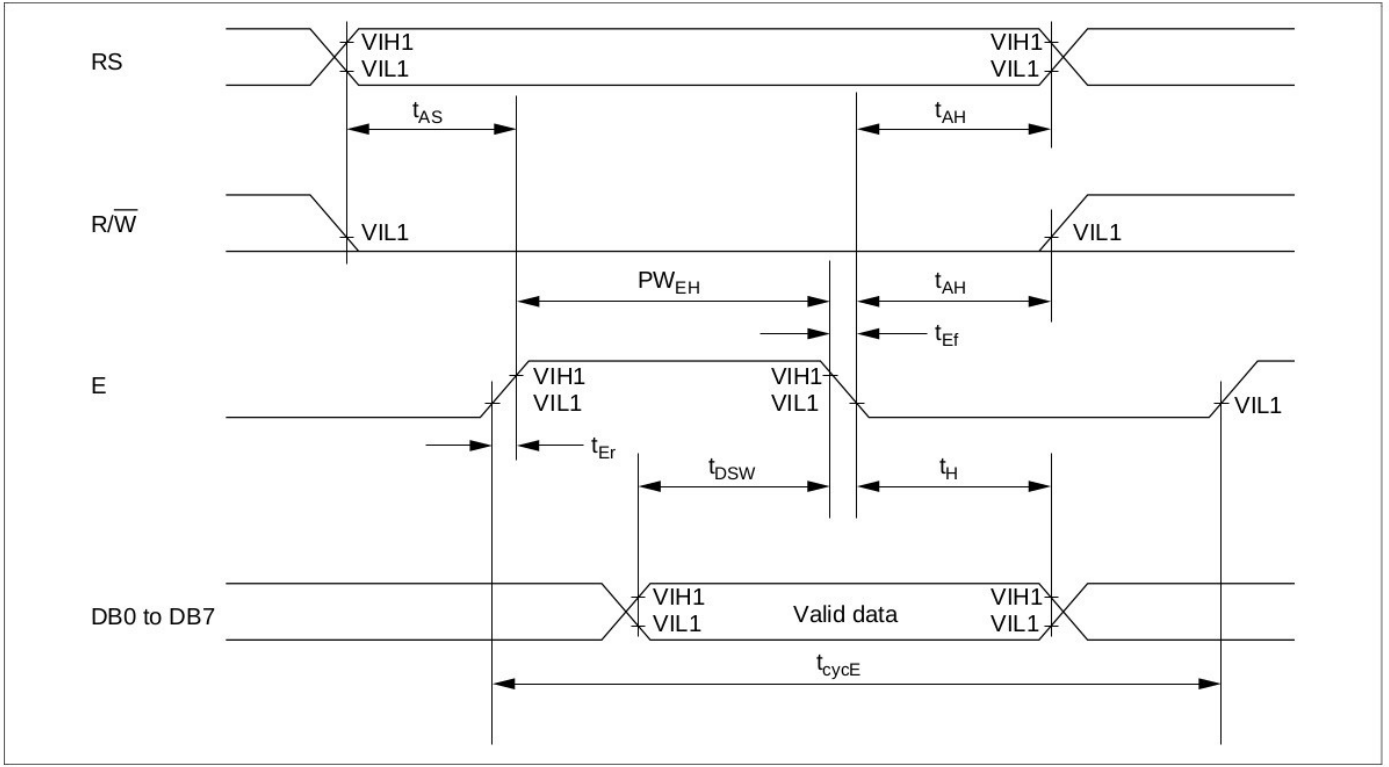
MeuWrite

```c
static ssize_t meuWrite(struct file *filp, const char __user *buff, size_t count, loff_t *offp)
{
    ssize_t tamSaida = 0;
    int i;
    printk(KERN_ALERT "DENTRO DA FUNÇÃO meuWrite.\n");

    if(!buff)//confere se o buffer não está vazio
    {
        printk(KERN_ALERT "Buffer vazio.\n");
        return -ERESTARTSYS;
    }

    mutex_lock(&display->meuMutex);

    for(i = 0; i < strlen(buff); i++)
    {
        write_command(display->meuCliente,buff[i],1); // 1 no terceiro argumento para dados

        display->coluna++; //incremento minha contagem das colunas
        if(display->coluna > 15) //confiro se estou na ultima coluna
        {
            display->coluna = 0;

            if(display->linha == 0) //confiro se estou na ultima linha
            {
                display->linha = 1;
                write_command(display->meuCliente,LCD_LINE1,0); // Para instruções -> Segundo bi

            }
            else
            {
                display->linha = 0;
                write_command(display->meuCliente,LCD_LINE0,0); // Para instruções -> Segundo bi

            }
        }
    }

    mutex_unlock(&display->meuMutex);
    tamSaida = strlen(buff);

    return tamSaida;
}
```

write_command

```
void write_command(struct i2c_client *client, u8 data, int instr)
{
    u8 up,low;

    up  = LCD_DATA_MSB(data); // separando 8 bits em duas partes de 4, pel
    low = LCD_DATA_LSB(data);

    if(instr == 0){ // se for instrução, não caracteres
        up  = LCD_RS_INSTR(up); // Avisando que é instrução
        low = LCD_RS_INSTR(low);
    }else{
        up  = LCD_RS_DATA(up); // Avisando que é dados(não instrução)
        low =LCD_RS_DATA(low);
    }

    up = up | LCD_BACKLIGHT; // display ativo
    low = low | LCD_BACKLIGHT;

    // Parte upper bits
    i2c_smbus_write_byte(client,up); // Mandando a parte upper dos 8 bits
    udelay(500);

    i2c_smbus_write_byte(client,LCD_E_HI(up)); // Enable high
    udelay(500);
    i2c_smbus_write_byte(client,LCD_E_LOW(up)); // Enable low -> Para pode
    udelay(500);

    // Parte lower bits

    i2c_smbus_write_byte(client,low); // Mandando a parte lower dos 8 bits
    udelay(500);

    i2c_smbus_write_byte(client,LCD_E_HI(low)); // Enable high
    udelay(500);
    i2c_smbus_write_byte(client,LCD_E_LOW(low)); // Enable low -> Para pod
    udelay(500);

    return;
}
```

Write Operation



Exemplo de como funciona a operação de escrita no display

write_macros

```
u8 retLCD_E_HI(u8 ret)
{
    return (ret | (LCD_ENABLE));
}

u8 retLCD_E_LOW(u8 ret)
{
    return (ret & ~(LCD_ENABLE));
}

u8 retLCD_RS_DATA(u8 ret)
{
    return (ret | LCD_CHR);
}

u8 retLCD_RS_INSTR(u8 ret)
{
    return (ret | LCD_CMD);
}

u8 retLCD_DATA_MSB(u8 ret)
{
    return (ret & 0xF0);
}

u8 retLCD_DATA_LSB(u8 ret)
{
    return ((ret << 4) & 0xF0);
}
```

write_macros

```
#define LCD_E_HI(ret)        retLCD_E_HI(ret)
#define LCD_E_LOW(ret)       retLCD_E_LOW(ret)

// Macro para definição valor pino RS -> 0 - instr

#define LCD_RS_DATA(data)    retLCD_RS_DATA(data)
#define LCD_RS_INSTR(data)   retLCD_RS_INSTR(data)

// Separa o byte em duas partes, uma MSB e outra L

#define LCD_DATA_MSB(x)    retLCD_DATA_MSB(x)
#define LCD_DATA_LSB(x)    retLCD_DATA_LSB(x)
```

meuIoctl

```
static long meuIoctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    printk(KERN_ALERT "CHAMANDO A FUNÇÃO meuIoctl.\n");
    mutex_lock(&display->meuMutex);

    switch(cmd)
    {
        case LIMPA_TELA://MODIFIQUEI AQUI
            write_command(display->meuCliente,LCD_CLEAR,0);
            display->linha = display->coluna = 0;
            break;

        case BACKLIGHT:
            setBacklight(display,arg);
            break;

        case CURSOR:»
            display_cursor(display,arg);
            break;

        case BLINK:»
            display_blink(display,arg);
            break;

        default:
            printk(KERN_ALERT "Comando inválido.\n");

    }

    mutex_unlock(&display->meuMutex);
    return 0;
}
```

display_cursor

```
void display_cursor(i2cDriverSO_t *display, u8 value){

    u8 addr = 1 << CURSOR ADDR;

    if(value == 1){
        display->regs cntrl |= addr;
    }else{
        display->regs_cntrl &= ~(addr);
    }

    write_command(display->meuCliente,display->regs_cntrl,0);
        udelay(200);

    return;
}
```

display_blink

```
void display_blink(i2cDriverSO_t *display, u8 value){

    u8 addr;
    addr = 1  << BLINK_ADDR;

    if(value == 1){
        display->regs_cntrl |= addr;
    }else{
        display->regs cntrl &= ~(addr);
    }

    write command(display->meuCliente,display->regs cntrl,0);
        udelay(200);

    return;
}
```

setBacklight

```
void setBacklight(i2cDriverSO t *display, u8 value){

    u8 addr;

    if(value == 1){
        addr = 0x08; // valor que aciona o backlight do display
    }else{
        addr = 0;
    }

    i2c_smbus_write_byte(display->meuCliente,addr);
        udelay(100);

    return;
}
```

userLevelFunctions

```
void Display()
{
    int in;
    int fd = OpenDisplay();
    char *text;
    if(fd < 0)
    {
        printf("Erro ao abrir o arquivo.\n");
        return;
    }
```

userLevelFunctions

```
while( in != 0 )
{
    printf("Digite seu comando mestre: \n0 - Sair
    scanf("%d", &in);

    switch(in)
    {
        case 0:
            write("Saindo");
            close(fd);
            break;

        case 1:
            scanf("%s",text);
            write(fd,text,0);
            break;

        case 2:
            ioctl(fd,BACKLIGHT,0);
            break;

        case 3:
            ioctl(fd,BACKLIGHT,1);
            break;

        case 4:
            ioctl(fd,LIMPA_TELA,0);
            break;

        case 5:
            ioctl(fd,CURSOR,1);
            break;

        case 6:
            ioctl(fd,CURSOR,0);
            break;

        case 7:
            ioctl(fd,BLINK,1);
            break;

        case 8:
            ioctl(fd,BLINK,0);
            break;

        default:
            printf("COMANDO INVALIDO.\n");
            break;
    }
}
```