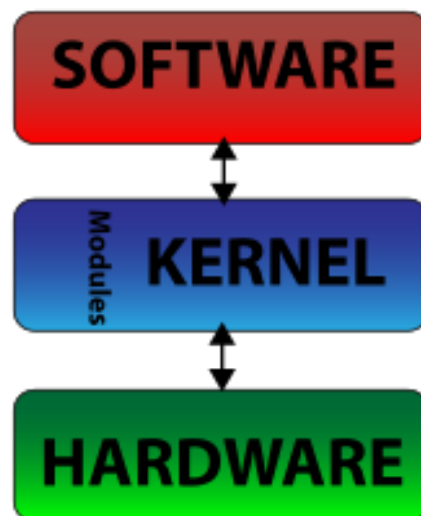


# OS Kernel

Rapport des TPs

EI-SE4

**KO Roqyun**  
**OLIVIER Raphaël**



<b>Introduction</b>	<b>2</b>
<b>TP 1</b>	<b>3</b>
Port Parallèle	3
Codage	4
Ecran LCD	5
Pinout	5
Data pins	5
Control pins	6
Chronogramme	7
Codage	8
DS1620 (Thermomètre)	10
Pinout	10
Status pins	10
Control pins	10
Chronogramme	12
Codage	13
<b>TP 2</b>	<b>14</b>
Module de noyau	15
Gestion de fichier	15
File Descriptor	15
Gestion de périphérique	17
ioctl (input / output control)	17
<b>TP 3</b>	<b>19</b>
Module de Noyau	19
Périphérique	20
Usage	20
Exemple :	20
<b>Conclusion</b>	<b>22</b>

# Introduction

L'objectif de ces TP est de manipuler le port parallèle par la programmation d'un module de noyau. Ainsi, on commande un écran LCD et un thermomètre relié au port parallèle. La manipulation du port parallèle peut être vérifiée par les fonctionnements correctes des modules branchés au port. L'ensemble des TP s'apparente à la réalisation d'un driver

Pendant le TP1, étudie les chronogrammes des modules (écran LCD et thermomètre) pour les commander. De même, on trouve l'adresse du port à manipuler. On utilise enfin la bibliothèque de C qui nous permet de manipuler le module de noyau sans y accéder.

Pendant le TP2, on accède directement à un module de noyau (/dev/parport0) pour manipuler le port parallèle au lieu utiliser la bibliothèque pour lire et écrire le port.

Pendant le TP3, on crée et écrit un module de noyau dans le répertoire /dev pour laisser les utilisateur commander les modules même sans connaître l'adresse du port.

# TP 1

Le but de ce TP est de paramétrer l'écran LCD, et d'afficher une valeur (dans ce cas, la température donnée par un capteur)

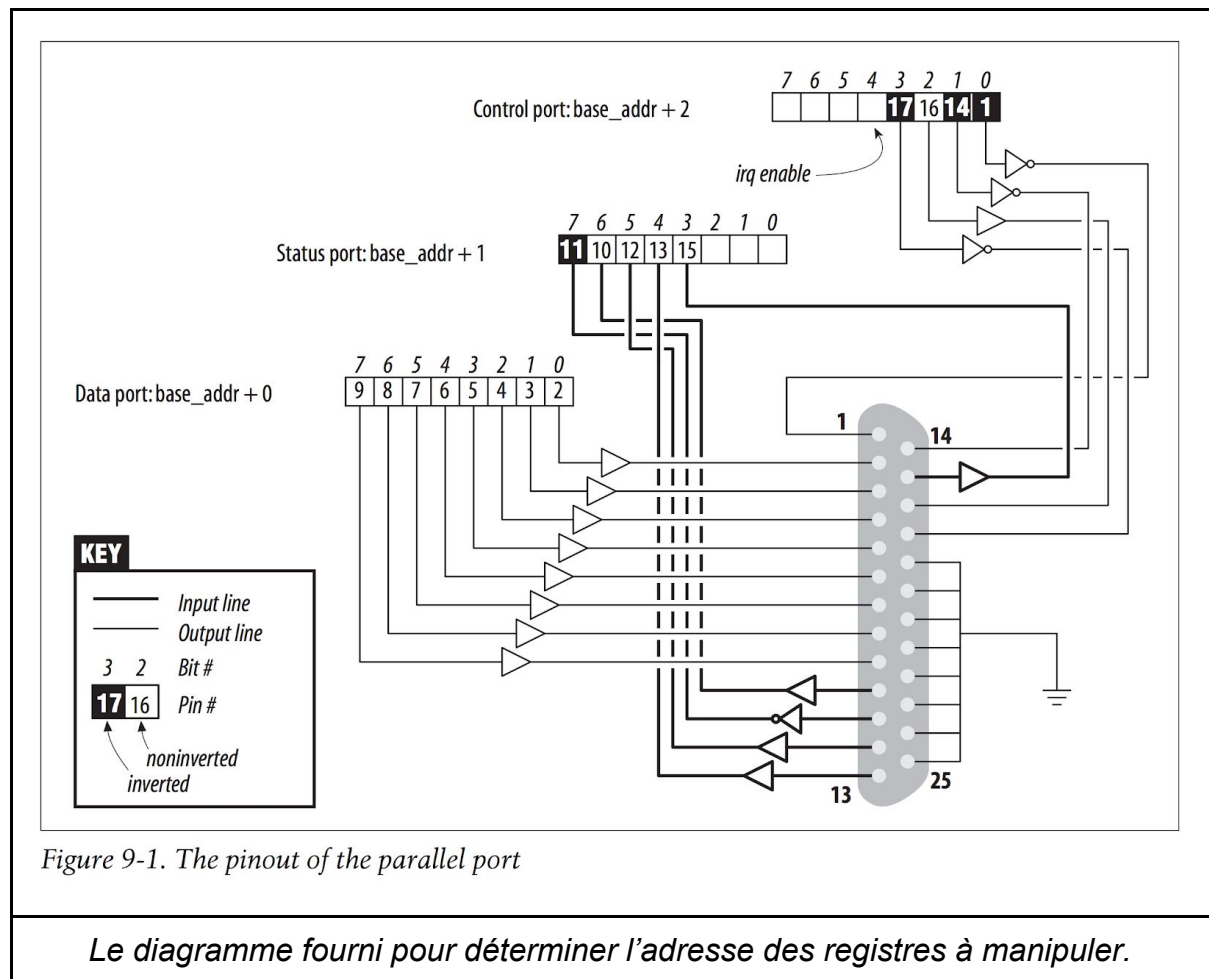
Il faut donc lire la valeur du capteur, la stocker, puis l'afficher grâce aux fonctions à notre disposition

## Port Parallèle

Le port parallèle qu'on utilise est LPT1 dont l'adresse de base vaut en générale 0x378h.

Chaque adresse possède un registre d'un octet qui peut être lu et écrit. Chaque bit des registres représente un seul pin.

Lors de l'écriture et la lecture de pin, il faut prendre en compte le fait que les signal de certaines pins soient inversés (soit **p17** la valeur écrite au pin 17, la valeur lue par un module est  $\overline{p17}$  ).



L'adresse du registre du

- data port : **0x378h**

- status port : **0x379h**
- control port : **0x37Ah**

## Codage

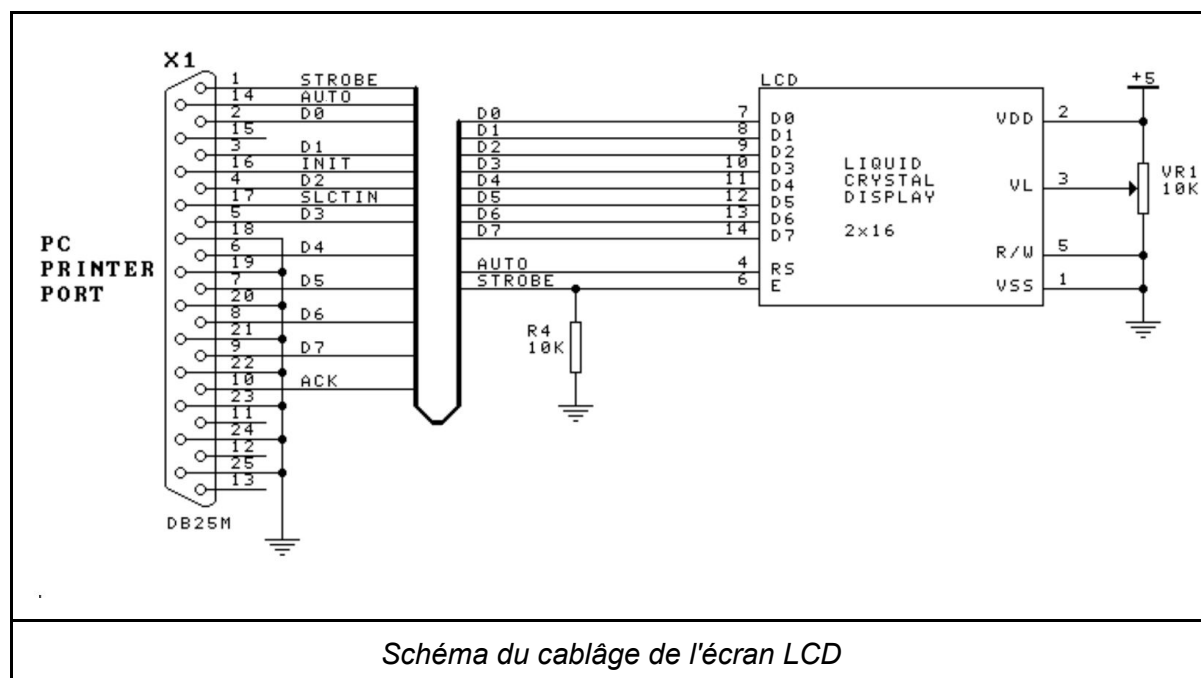
```
#define ADDRESS_LPT1      0x378
#define ADDRESS_DATA      ADDRESS_LPT1 + 0
#define ADDRESS_STATUS    ADDRESS_LPT1 + 1
#define ADDRESS_CONTROL   ADDRESS_LPT1 + 2

// Input Output Permission => ioperm
// Autoriser l'écriture et la lecture du registre à l'adresse donnée
ioperm(ADDRESS_LPT1, 3, 1);

//Initialiser le registre de commande
// -> STROBE à 0 (broche 1, bit 0 de 0x37A)
// -> AUTO à 0 (broche 14, bit 1 de 0x37A)
// -> INIT à 0 (broche 16, bit 2 de 0x37A)
// -> SLCTIN à 1 (broche 17, bit 3 de 0x37A)

//Registre :0b1000 => 0b0011 = 0x3
outb((unsigned char)0x3, ADDRESS_CONTROL); // 0b0011
```

## Ecran LCD



## Pinout

Nom du pin	Pin du port	Nom du reg	Adresse	Offset	Inversé
<b>D0</b>	2	Data	0x378h	0	X
<b>D1</b>	3	Data	0x378h	1	X
<b>D2</b>	4	Data	0x378h	2	X
<b>D3</b>	5	Data	0x378h	3	X
<b>D4</b>	6	Data	0x378h	4	X
<b>D5</b>	7	Data	0x378h	5	X
<b>D6</b>	8	Data	0x378h	6	X
<b>D7</b>	9	Data	0x378h	7	X
<b>STROBE</b>	1	Control	0x37Ah	0	○
<b>AUTO</b>	14	Control	0x37Ah	1	○

## Data pins

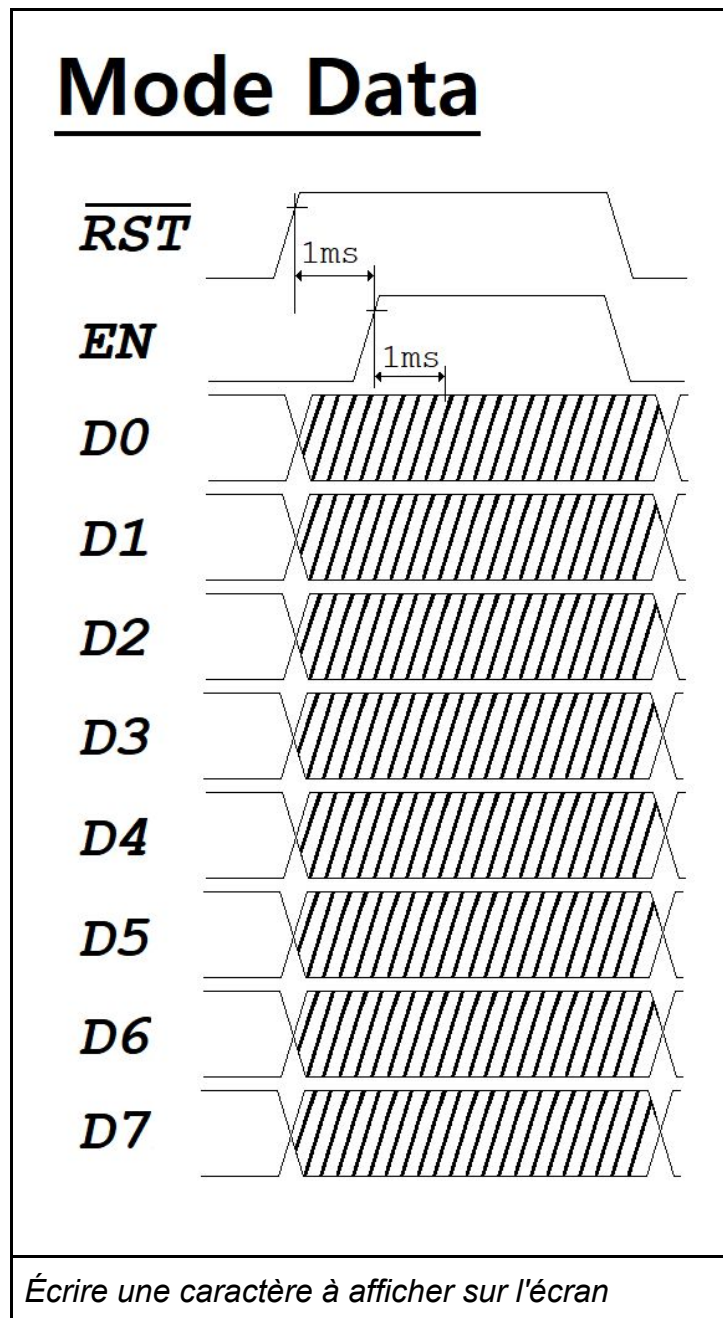
Les pins **D0 ~ D7** sort les données à afficher à l'écran LCD. Ces données représentent une caractère ASCII.

## Control pins

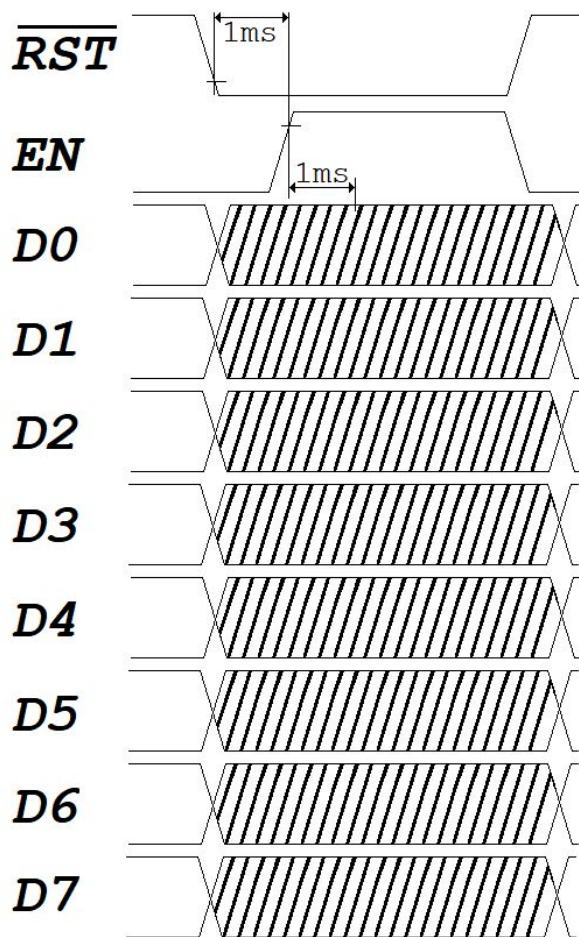
- **AUTO** correspond au signal **RESET (RST)** de l'écran.

- Pour **0**, le mode **commande** est activé
- Pour **1**, le mode **écriture** est activé
- **STROBE** correspond au signal **ENABLE (EN)** de l'écran.
  - Pour **0**, le mode lecture est activé
  - Pour **1**, le mode écriture est activé

## Chronogramme



# Mode Command



*Changer le mode du fonctionnement de l'écran*

La seule différence entre les 2 modes est l'état du signal **RST**

## Codage

```
#define ADDRESS_LPT1      0x378
#define ADDRESS_DATA      ADDRESS_LPT1 + 0
#define ADDRESS_STATUS    ADDRESS_LPT1 + 1
#define ADDRESS_CONTROL   ADDRESS_LPT1 + 2
```



```

#define OFFSET_EN 0
#define OFFSET_RST 1

// inb = Lire le registre de l'adresse.
// outb = Écrire le registre de l'adresse.

// Q5) Ecrire la fonction LCD_E_HIGH() qui permet de mettre E a 1
void LCD_E_HIGH(void)
{
    outb(inb(port) & ~(1 << OFFSET_EN), ADDRESS_CONTROL);
}

// Q6) Ecrire la fonction LCD_E_LOW() qui permet de mettre E a 0
void LCD_E_LOW(void)
{
    outb(inb(port) | (1 << OFFSET_EN), ADDRESS_CONTROL);
}

// Q7) Ecrire la fonction LCD_RS_HIGH() qui permet de mettre RS a 1
void LCD_RS_HIGH(void)
{
    outb(inb(port) & ~(1 << OFFSET_EN), ADDRESS_CONTROL);
}

// Q8) Ecrire la fonction LCD_RS_LOW() qui permet de mettre RS a 0
void LCD_RS_LOW(void)
{
    outb(inb(port) | 1 << OFFSET_EN, ADDRESS_CONTROL);
}

void LCD_CMD(unsigned char data)
{
    outb(data, ADDRESS_DATA );
    LCD_RS_LOW();
    usleep(1000) ;
    LCD_E_HIGH();
    usleep(1000) ;
    LCD_E_LOW();
    usleep(1000) ;
}

void LCD_CHAR(unsigned char data)
{
    outb(data, ADDRESS_DATA );
    LCD_RS_HIGH();
    usleep(1000) ;
    LCD_E_HIGH();
}

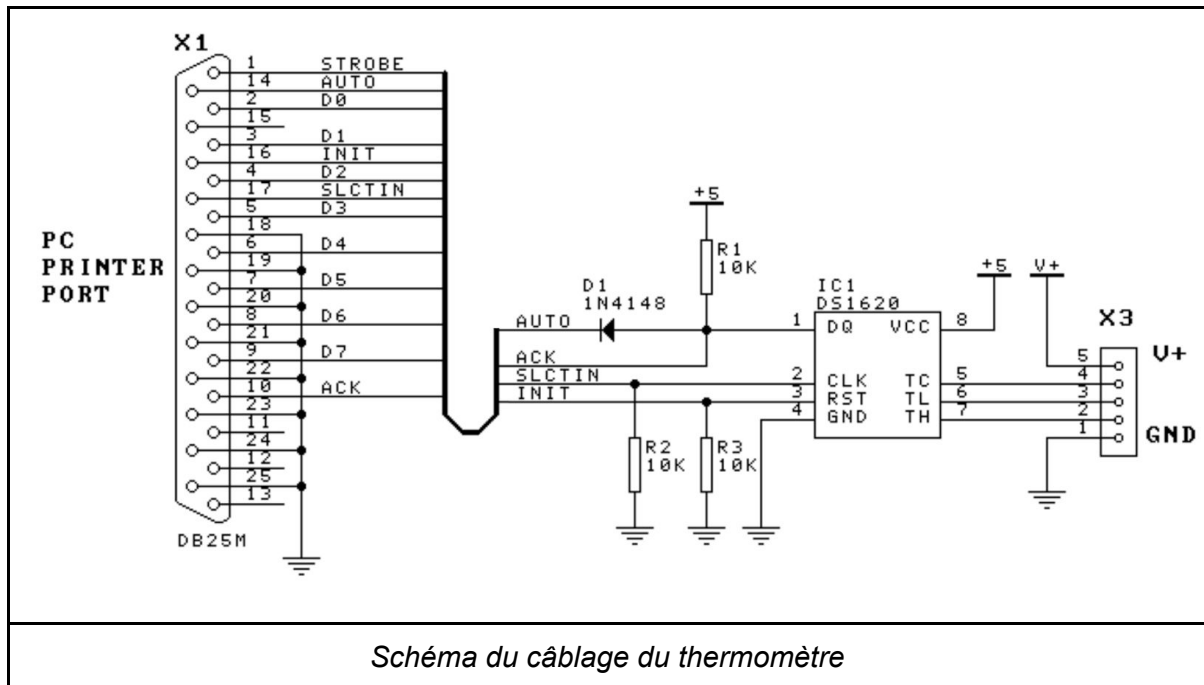
```

```

usleep(1000);
LCD_E_LOW();
usleep(1000);
}

```

## DS1620 (Thermomètre)



## Pinout

Nom du pin	Pin du port	Nom du reg	Adresse	Offset	Inversé
<b>AUTO</b>	2	Control	0x37Ah	1	○
<b>INIT</b>	3	Control	0x37Ah	2	✕
<b>SLCTIN</b>	4	Control	0x37Ah	3	○
<b>ACK</b>	5	Status	0x379h	6	✕

## Status pins

En mode lecture, **ACK** correspond au pin **DQ** du thermomètre.

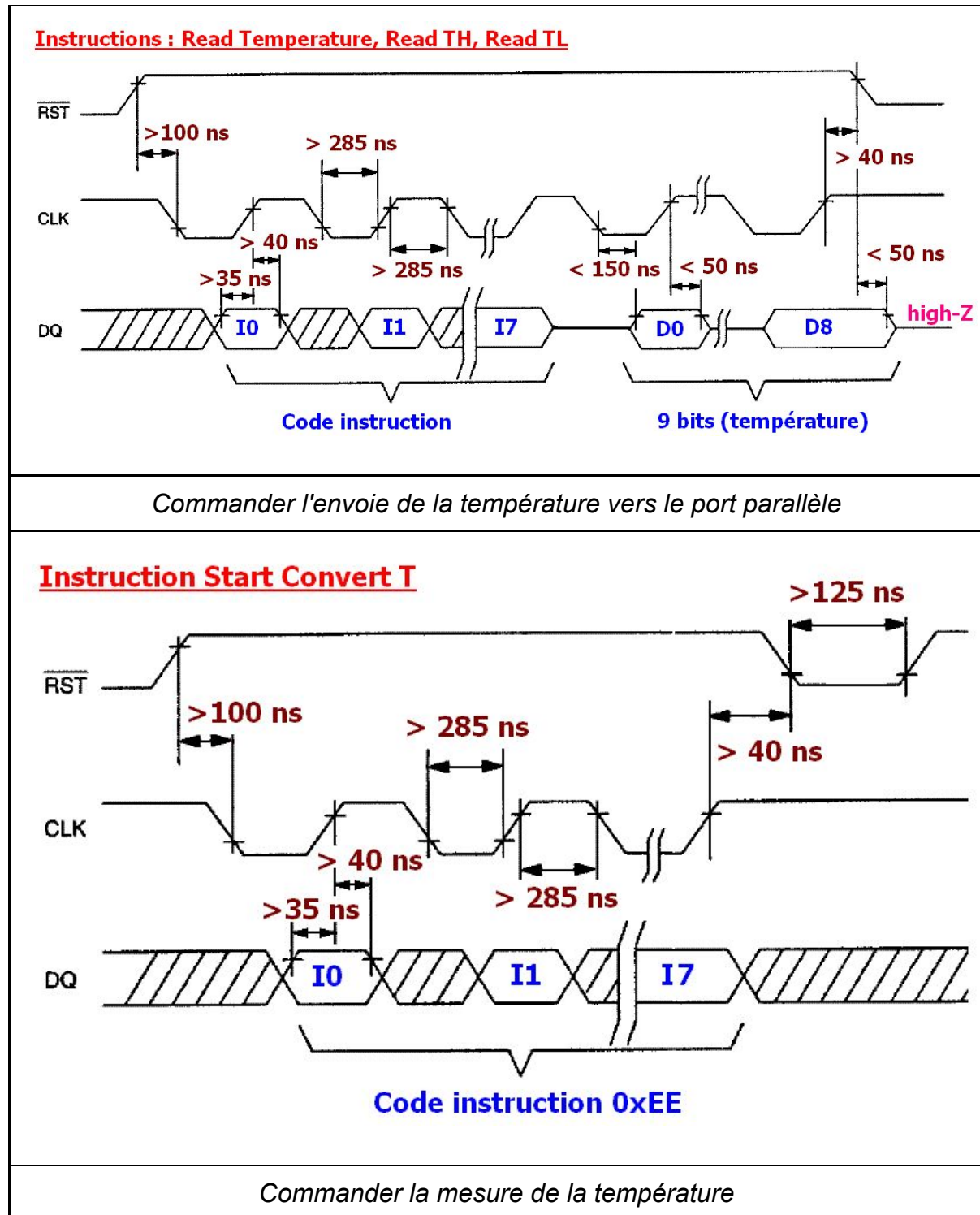
## Control pins

En mode écriture, **AUTO** correspond au pin **DQ** du thermomètre.

**SLCTIN** correspond au pin **CLK** du thermomètre

**RST** correspond au pin **INIT** du thermomètre.

## Chronogramme



Le protocole de la communication est similaire à SPI (Serial Peripheral Interface).

## Codage

On peut se référencer à l'écriture des fonctions **LCD\_XXX\_HIGH** / **LCD\_XXX\_LOW** pour écrire :

**DS1620\_CLK\_LOW**

**DS1620\_CLK\_HIGH**

```
// Q21) Ecrire la fonction DS1620_WRITECOMMAND() qui permet  
// d'envoyer une commande sur 8 bits au thermometre.  
// La commande s'envoie bit apres bit, en commençant par le  
// bit de poids faible
```

```
void DS1620_WRITECOMMAND(unsigned char command)  
{  
    int i = 0;  
  
    tme_tempo(2);  
  
    for (i = 0; i < 8; i++) {  
        // Mettre le signal clk en etat bas.  
        DS1620_CLK_LOW();  
        tme_tempo(2);  
  
        // Envoyer la commande  
        if(command & 0x1)  
            DS1620_DQ_HIGH();  
        else  
            DS1620_DQ_LOW();  
  
        tme_tempo(2);  
  
        // clk en etat haut = front montant  
        DS1620_CLK_HIGH();  
        tme_tempo(2);  
        command >>= 1;  
    }  
}
```

```
// Q22) Ecrire la fonction DS1620_READ() qui permet  
// de lire une temperature sur 9 bits depuis le thermometre  
// la temperature se lit bit apres bit en commençant par  
// le bit de poids faible.
```

```
int DS1620_READ(void)  
{
```

```

    int i = 0, data;

    DS1620_DQ_HIGH();          //setup to read from data pin
    tme_tempo(2);
    data=0;                    //initialize data byte

    for (i = 0; i < 9; i++) {
        DS1620_CLK_LOW();
        tme_tempo(2);

        port = ADDRESS_STATUS;
        if(inb(port) & (1 << BROCHE_ACK))
            data += 1 << i; //if bit is high, add its value to total
        DS1620_CLK_HIGH();
        tme_tempo(2);
    }
    tme_tempo(2);
    return data;
}

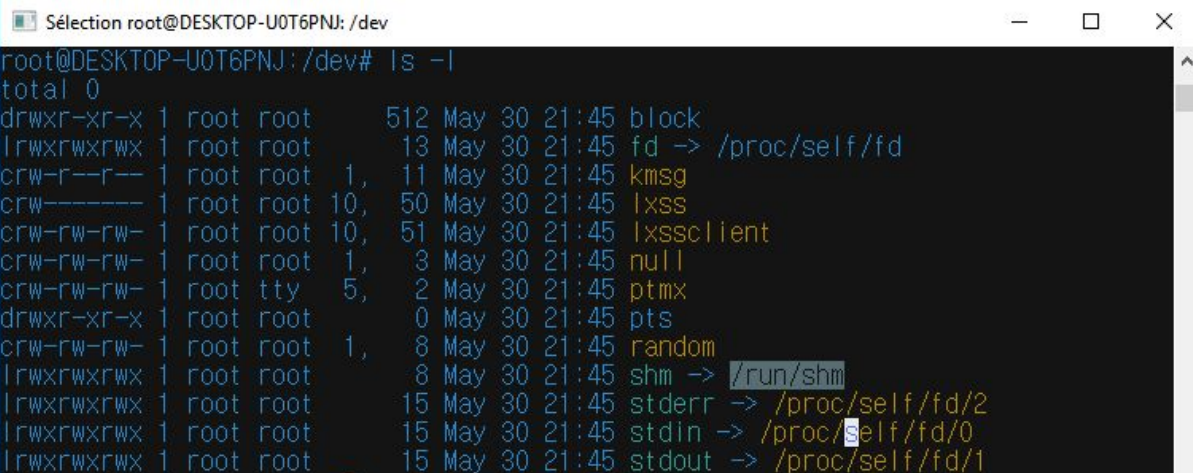
```

## TP 2

Le but de ce TP est de commander le module de l'écran LCD et du thermomètre comme le TP1, mais on va utiliser le module noyau (**ppdev**) pour piloter individuellement les broches du port parallèle au lieu d'utiliser la bibliothèque existante.

On programme alors nous-même **inb** et **outb**, qui ont pour but de gérer le port parallèle. Afin de pouvoir programmer ces fonctions, il faut comprendre que les périphérique d'entrée et de sortie sont gérés en fichier lisible et modifiable.

## Module de noyau



```
Sélection root@DESKTOP-U0T6PNJ: /dev
root@DESKTOP-U0T6PNJ:/dev# ls -l
total 0
drwxr-xr-x 1 root root 512 May 30 21:45 block
lrwxrwxrwx 1 root root 13 May 30 21:45 fd -> /proc/self/fd
crw-r--r-- 1 root root 1, 11 May 30 21:45 kmsg
crw----- 1 root root 10, 50 May 30 21:45 lxss
crw-rw-rw- 1 root root 10, 51 May 30 21:45 lxssclient
crw-rw-rw- 1 root root 1, 3 May 30 21:45 null
crw-rw-rw- 1 root tty 5, 2 May 30 21:45 ptmx
drwxr-xr-x 1 root root 0 May 30 21:45 pts
crw-rw-rw- 1 root root 1, 8 May 30 21:45 random
lrwxrwxrwx 1 root root 8 May 30 21:45 shm -> /run/shm
lrwxrwxrwx 1 root root 15 May 30 21:45 stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root 15 May 30 21:45 stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root 15 May 30 21:45 stdout -> /proc/self/fd/1
```

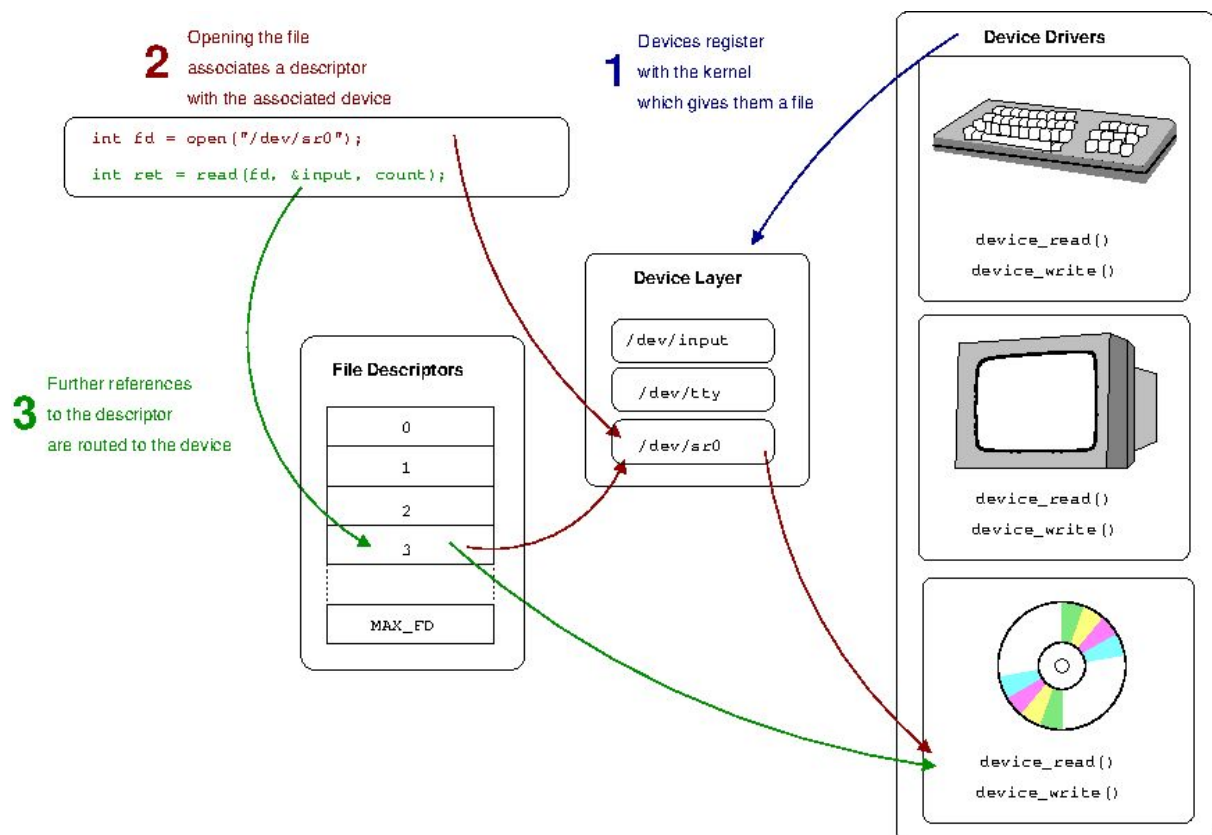
Les modules de noyau sont enregistré en répertoire **/dev/**. Dans ce TP, on ouvre le module **/dev/parport0** qui représente le port parallèle LPT1.

Pour piloter le port parallèle, il est donc nécessaire de comprendre comment gérer un fichier comme le module est représenté par un fichier et comment gérer un périphérique.

## Gestion de fichier

### File Descriptor

Pour pouvoir obtenir un accès à un fichier, il faut récupérer un « file descriptor » (*descripteur de fichier* en français). Un descripteur de fichier est une clé abstraite pour accéder à un fichier. Dans un système d'exploitation, des programmes identifient les fichiers à ouvrir grâce aux descripteurs de fichier. Ainsi, on identifie le module de noyau à ouvrir dans ce TP comme l'illustration ci-dessous.



[https://www.bottomupcs.com/file\\_descriptors.shtml](https://www.bottomupcs.com/file_descriptors.shtml)

Dans le langage C, on obtient le descripteur de fichier avec la fonction **open** :

```
int fd=open("/dev/parport0", O_RDWR);
```

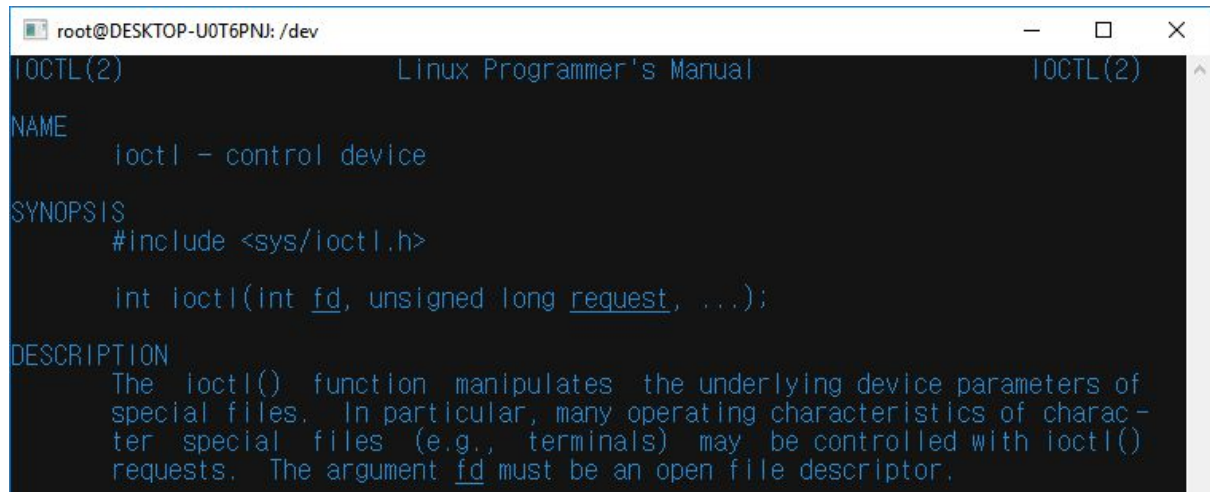
A la fin du programme, on ferme le fichier :

```
close(fd);
```



# Gestion de périphérique

## ioctl (input / output control)



```
root@DESKTOP-U0T6PNJ: /dev
ioctl(2)                                Linux Programmer's Manual                                ioctl(2)

NAME
    ioctl - control device

SYNOPSIS
    #include <sys/ioctl.h>

    int ioctl(int fd, unsigned long request, ...);

DESCRIPTION
    The ioctl() function manipulates the underlying device parameters of
    special files. In particular, many operating characteristics of charac-
    ter special files (e.g., terminals) may be controlled with ioctl()
    requests. The argument fd must be an open file descriptor.
```

On identifie le périphérique à piloter par le descripteur de fichier et on le pilote en utilisant la fonction **ioctl**.

Les paramètres disponibles pour le deuxième argument **request** sont ci-dessous

```
23  /* Read status */
24  #define PPRSTATUS      _IOR(PP_IOCTL, 0x81, unsigned char)
25  #define PPWSTATUS      OBSOLETE__IOW(PP_IOCTL, 0x82, unsigned char)
26
27  /* Read/write control */
28  #define PPRCONTROL     _IOR(PP_IOCTL, 0x83, unsigned char)
29  #define PPWCONTROL     _IOW(PP_IOCTL, 0x84, unsigned char)
30
36
37  /* Read/write data */
38  #define PPRDATA        _IOR(PP_IOCTL, 0x85, unsigned char)
39  #define PPWDATA        _IOW(PP_IOCTL, 0x86, unsigned char)
40
48
49  /* Claim the port to start using it */
50  #define PPCLAIM        _IO(PP_IOCTL, 0x8b)
51
52  /* Release the port when you aren't using it */
53  #define PPRELEASE      _IO(PP_IOCTL, 0x8c)
```

<https://github.com/torvalds/linux/blob/master/include/uapi/linux/ppdev.h>

Le préfix



- **PPR** signifie **P**arallel **P**ort **R**ead
- **PPW** signifie **P**arallel **P**ort **W**rite

Pour paramétrer le périphérique à piloter, on doit déclarer utilisant du port parallèle et le verrouiller pour les autres processus:

```
ioctl(fd,PPCLAIM);
```

On permet des autres à utiliser le port parallèle à la fin du programme.

```
ioctl(fd,PPRELEASE);
```

Pour piloter les broches du port parallèle, on écrit les fonctions **outb** et **inb** de notre propre version :

```
void myoutb(char v, unsigned int adr)
{
    switch (adr)
    {
        case 0x378:
            ioctl(fd,PPWDATA,&v);
            break;
        case 0x37A:
            ioctl(fd,PPWCONTROL,&v);
            break;
    }
}

char myinb(unsigned int adr)
{
    char b;
    switch (adr)
    {
        case 0x378:
            ioctl(fd,PPRDATA,&b);
            break;
        case 0x379:
            ioctl(fd,PPRSTATUS,&b);
            break;
        case 0x37A:
            ioctl(fd,PPRCONTROL,&b);
            break;
    }
    return b;
}
```

## TP 3

Le but de ce TP est d'utiliser l'écran et le capteur comme précédemment, mais en utilisant le module de noyau. Il faut donc enregistrer un « character device » qui permet les utilisateurs de commander les modules depuis le terminal.

### Module de Noyau

- La commande **insmod** est liée à **lcd\_init()** par **module\_init()**, et installe le noyau.
- La commande **dmesg** affiche le journal d'événement que l'on peut écrire avec la fonction **printk**.
- La commande **mknod** crée le noeud fichier (**/dev/lcd**).

Pour créer le module, on lance le script fourni :

```
#!/bin/sh
# $Id: lcd_load,v 1.4 2004/11/03 06:19:49 rubini Exp $
module="lcd"
device="lcd"
mode="666"

# Group: since distributions do it differently, look for wheel or use staff
if grep -q '^staff:' /etc/group; then
    group="staff"
else
    group="wheel"
fi

# invoke insmod with all arguments we got
# and use a pathname, as insmod doesn't look in . by default
/sbin/insmod ./module.ko $* || exit 1

# retrieve major number
major=$(awk "\$2==\"$module\" {print \$1}" /proc/devices)

# Remove stale nodes and replace them, then give gid and perms
# Usually the script is shorter, it's scull that has several devices in it.

rm -f /dev/${device}
mknod /dev/${device}0 c $major 0
ln -sf ${device}0 /dev/${device}
chgrp $group /dev/${device}0
chmod $mode /dev/${device}0
```

## Périphérique

Nous avons écrit plusieurs fonctions pour faire fonctionner l'écran LCD :

```
int lcd_open(struct inode *, struct file *)
int lcd_release(struct inode *, struct file *)
ssize_t lcd_write(struct file *, const char __user *, size_t, loff_t *)
ssize_t lcd_read(struct file *, char __user *, size_t, loff_t *)
```

On copie et colle ce qu'on a écrit précédemment lors du TP 1 et du TP 2.

*Remarque : **lcd\_read** commande également le module DS1620 et affiche la température dans l'écran LCD.*

Ensuite, il faut relier ces fonctions au « character device » à créer et enregistrer dans la fonction **lcd\_init(void)** :

```
struct file_operations lcd_fops = {
    .owner = THIS_MODULE,
    .read = lcd_read,
    .write = lcd_write,
    .open = lcd_open,
    .release = lcd_release
};

register_chrdev(major, DEVICE_NAME, &lcd_fops);
```

## Usage

Nous pouvons à présent lancer le module noyau. Pour faire marcher les fonctions codées, nous utilisons **echo** qui permet d'écrire dans le **stdout** et rediriger le message écrit vers le module. On appelle ainsi **lcd\_write**.

Nous utilisons **cat** pour appeler **lcd\_read** pour mesurer et afficher la température.

### Exemple :

**echo "testé" > /dev/lcd** va donc obtenir le droit à lire/piloter le port parallèle avec **lcd\_open()**, puis écrire avec **lcd\_write()**, puis le refermer avec **lcd\_release()**, et ainsi afficher test sur l'écran.

`cat /dev/lcd > /dev/lcd` va obtenir le droit à lire piloter le port parallèle avec ***lcd-open()***, puis lire avec ***lcd\_read()***, puis le refermer avec ***lcd\_release()***, et ainsi afficher la valeur de la température sur l'écran

## Conclusion

Ces TP nous ont permis de remplacer le module du noyau. La compréhension du fonctionnement du module noyau s'est donc fait par étapes : En commençant par la manipulation du module de noyau, en passant par le contrôle des entrées sorties, et en finissant par la création du module noyaux et « character device », nous avons donc piloté le port parallèle.