RAPPORT MODULE OSK

MODULE NOYAU LINUX

Achab Lounes : achab.lounes@gmail.com
Boulanger Matthieu :matthieu.blngr@gmail.com
Khalghdoost Afshin: afshin_90@hotmail.fr

SOMMAIRE

1) Introduction

Dans le cadre du module «OSK» en 4eme année de la spécialité Électronique et Informatique système embarqué (EISE-4) au sein de l'école Polytech UPMC, les élèves ingénieurs sont amenés à étudier les modules noyau Linux afin d'avoir les bases et notions nécessaires dans ce domaine pour leur future carrière d'ingénieur.

Ce document décrit les notions abordées en TP, et explique plus ou moins en détails les points importants concernant la programmation d'un module noyau sous Linux. Ainsi ce document est présenté sous forme de «rapport de TP», réalisé par notre groupe d'élèves ingénieurs.

2) Contexte et définition

2.1) Noyau de système d'exploitation

Un noyau de système d'exploitation, ou simplement noyau (ou kernel), est une des parties fondamentales de certains systèmes d'exploitation. Il gère les ressources de l'ordinateur et permet aux différents composants matériels et logiciels de communiquer entre eux.

En tant que partie intégrante du système d'exploitation, le noyau fournit des mécanismes d'abstraction du matériel, notamment de la mémoire, du (ou des) processeur(s), et des échanges d'informations entre logiciels et périphériques matériels. Le noyau autorise aussi diverses abstractions logicielles et facilite la communication entre les processus.

Le noyau d'un système d'exploitation est lui-même un logiciel, mais il ne peut cependant pas utiliser tous les mécanismes d'abstraction qu'il fournit aux autres logiciels. Son rôle central impose par ailleurs des performances élevées. Cela fait du noyau, la partie la plus critique d'un système d'exploitation et rend sa conception et sa programmation particulièrement délicates. Plusieurs techniques sont mises en œuvre pour simplifier la programmation des noyaux tout en garantissant de bonnes performances.

2.2) Appel système

En informatique, un appel système (en anglais, *system call*, abrégé en *syscall*) est une fonction primitive fournie par le noyau d'un système d'exploitation et est utilisée par les programmes s'exécutant dans l'espace utilisateur (en d'autres termes, tous les processus distincts du noyau). Ce système permet de contrôler de façon sécurisée, les applications dans l'espace utilisateur.

Le rôle du noyau est de gérer les ressources matérielles (il contient des pilotes de périphériques) et de fournir aux programmes une interface uniforme pour l'accès à ces ressources.

Quelques appels systèmes classiques:

- *open*, *read*, *write* et *close* qui permettent les manipulations sur les systèmes de fichiers.
- *brk*, *sbrk*, utilisés par *malloc* et *free* pour allouer et désallouer de la mémoire.

2.3) Gestion du matériel

La gestion du matériel se fait par l'intermédiaire de pilotes de périphériques. Les pilotes sont des petits logiciels légers dédiés à un matériel donné qui permettent de faire communiquer ce matériel. En raison du très grand nombre d'accès à certains matériels (disques durs par exemple), certains pilotes sont très sollicités. Ils sont généralement inclus dans l'espace noyau et communiquent avec l'espace utilisateur via les appels système.

En effet, comme cela a été vu dans le précédent paragraphe, un appel système est coûteux: il nécessite au moins deux changements de contexte. Afin de réduire le nombre des appels système effectués pour accéder à un périphérique, les interactions basiques avec le périphérique sont faites dans l'espace noyau. Les programmes utilisent ces périphériques au travers d'un nombre restreint d'appels système.

Cependant, indépendamment de l'architecture, de nombreux périphériques lents (certains appareils photographiques numériques, outils sur liaison série, etc.) sont/peuvent être pilotés depuis l'espace utilisateur, le noyau intervenant au minimum.

2.4) Pilote de périphérique

Un pilote de périphérique (device driver) permet à un programme utilisateur d'accéder à des ressources externes. Ces ressources peuvent être de type:

- matérielles: carte d'extension, périphérique sur la carte mère
- logicielles: mémoire physique

A de rares exceptions près, ce type de ressource est accessible exclusivement au noyau (kernel space) et non aux programmes utilisateurs (user space). Il existe cependant des exceptions célèbres comme le serveur XFree86 ou bien la svgalib qui effectuent des accès directs à des périphériques.

Il existe différents types de pilotes suivant le périphérique à contrôler:

- les pilotes en mode caractère (char drivers): ce type de pilote est le plus simple à mettre en œuvre. Il permet de dialoguer avec le périphérique en échangeant un nombre variable d'informations binaires. Les ports séries et parallèles sont des exemples de périphériques contrôles en mode caractère.
- les pilotes en mode bloc (block drivers): ces pilotes échangent des informations avec les périphériques uniquement par blocs de données. Un exemple d'un tel périphérique est le disque dur.
- les pilotes réseau (network drivers): ces pilotes sont destinés à contrôler des ressources réseau.

Dans la suite du document, nous nous intéresserons *exclusivement* aux pilotes en mode caractère.

3) LCD et Capteur de température

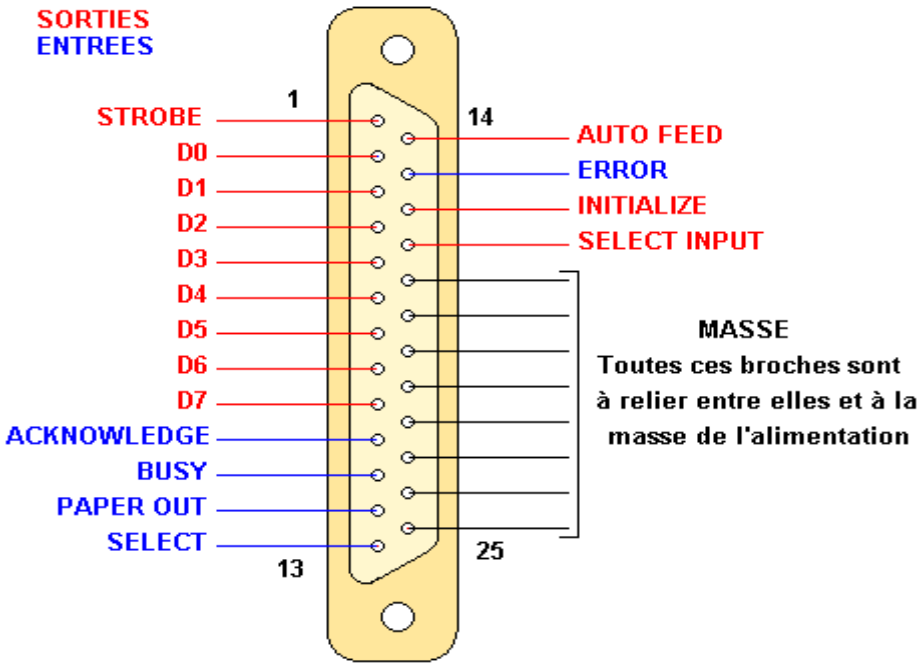
3.1) Présentation

Dans cette partie du rapport nous allons étudier et analyser l'utilisation de deux périphériques qui sont: un écran LCD 2*16 et un capteur de température en 3 parties bien distinctes. Nous verrons d'abord la partie une qui consistera en l'écriture et le test d'un programme de contrôle des périphériques. Celui-ci configurera les deux périphériques directement. Dans cette partie, il nous faudra être root pour pouvoir exécuter le programme et le tester. Puis, dans une seconde partie, nous allons nous mettre en tant que simple utilisateur et allons utiliser les deux périphériques à travers un module noyau PPDEV déjà fourni. Nous verrons donc ici comment utiliser le module noyau d'un périphérique déjà installé et cela via les appels systèmes appropriés. Pour finir, nous allons écrire notre propre module noyau pour contrôler les périphériques.

3.1.1) Port parallèle

La première interface parallèle a été définie par IBM et était destinée à connecter une imprimante ou un affichage monochrome. Elle est maintenant aussi utilisée dans certains montages simples se branchant sur un PC, et dans sa version la plus évoluée (ECP ou EPP), pour brancher des périphériques plus évolués, comme des scanners.

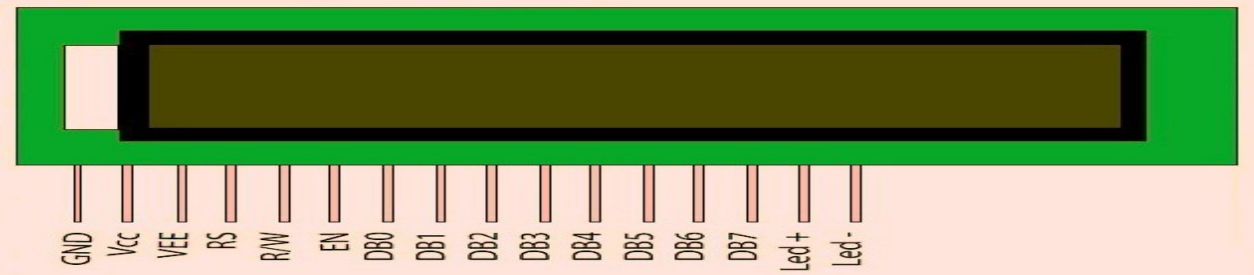
Le port parallèle des PC se présente sous la forme d'un prise DB25 femelle, alors que côté imprimante, un connecteur de type Centronics à 36 broches est généralement utilisé. Ce port a été pensé pour communiquer avec une imprimante et la plupart de ses signaux ont un rapport avec ce périphérique. Il possède 17 broches utilisables, les broches restantes étant reliées à la masse.



3.1.2) LCD 2*16

a) Présentation

Les afficheurs à cristaux liquides, autrement appelés afficheurs LCD (Liquid Crystal Display), sont des modules compacts intelligents et nécessitent peu de composants externes pour un bon fonctionnement. Ils consomment relativement peu (de 1 à 5 mA), sont bon marché et s'utilisent avec beaucoup de facilité.



Broche	Nom	Niveau	Fonction
1	VSS	-	Masse
2	VDD	-	Alimentation positive (+5V).
3	VEE	0-5V	Cette tension permet, en la faisant varier entre 0 et +5V, le réglage du contraste de l'afficheur.
4	RS	TTL	Selection du registre (Register Select) Grâce à cette broche, l'afficheur est capable de faire la différence entre une commande et une donnée. Un niveau bas indique une commande et un niveau haut indique une donnée.
5	RW	TTL	Lecture ou écriture (Read/Write) L : Écriture H : Lecture
6	E	TTL	Entrée de validation (Enable) active sur front descendant. Le niveau haut doit être maintenue pendant au moins 450 ns à l'état haut.
7	D0	TTL	Bus de données bidirectionnel 3 états (haute impédance lorsque E=0)
8	D1	TTL	
9	D2	TTL	
10	D3	TTL	
11	D4	TTL	
12	D5	TTL	
13	D6	TTL	
14	D7	TTL	
15	A	-	Anode rétroéclairage (+5V)
16	K	-	Cathode rétroéclairage (masse)

b) La mémoire

L'afficheur possède deux types de mémoire, la DD RAM et la CG RAM. La DD RAM correspond à la mémoire d'affichage tandis que la CG RAM correspond à la mémoire du générateur de caractères.

b.1) La mémoire d'affichage (DD RAM)

La DD RAM est la mémoire qui stocke les caractères actuellement affichés à l'écran. Pour un afficheur de 2 lignes de 16 caractères, les adresses sont définies de la façon suivante :

Ligne	Visible	Invisible
Haut	00H.....0FH	10H.....27H
Bas	40H.....4FH	50H.....67H

L'adresse 00H correspond à la ligne du haut à gauche, 0FH à droite. L'adresse 40H correspond à la ligne du bas à gauche, 4FH à droite. La zone invisible correspond à la mémoire de l'afficheur (48 caractères). Lorsqu'un caractère est inscrit à l'adresse 27H, le caractère suivant apparaît à la ligne suivante.

b.2) La mémoire du générateur de caractères (CG RAM)

Le générateur de caractère est quelque chose de très utile. Il permet la création d'un maximum de 8 caractères ou symboles 5x7. Une fois les nouveaux caractères chargés en mémoire, il est possible d'y accéder comme s'il s'agissait de caractères classiques stockés en ROM.

La CG RAM utilise des mots de 8 bits de large, mais seul les 5 bits de poids faible apparaissent sur le LCD. Ainsi D4 représente le point le plus à gauche et D0 le point le plus à droite. Par exemple, le fait de charger un octet de la CG RAM à 1FH fait apparaître tous les points de cette rangée ; le fait de charger un octet à 00H éteint tous ces points. Les 8 lignes d'un caractère doivent être chargées dans la CG RAM.

La CG RAM peut être utilisée afin de créer des caractères en vidéo inversée, des caractères avec des accents, etc. La limitation d'un total de 8 caractères peut être contournée en utilisant une bibliothèque de 8 symboles résidant dans le système hôte. Un maximum de 8 caractères peut être affiché à la fois.

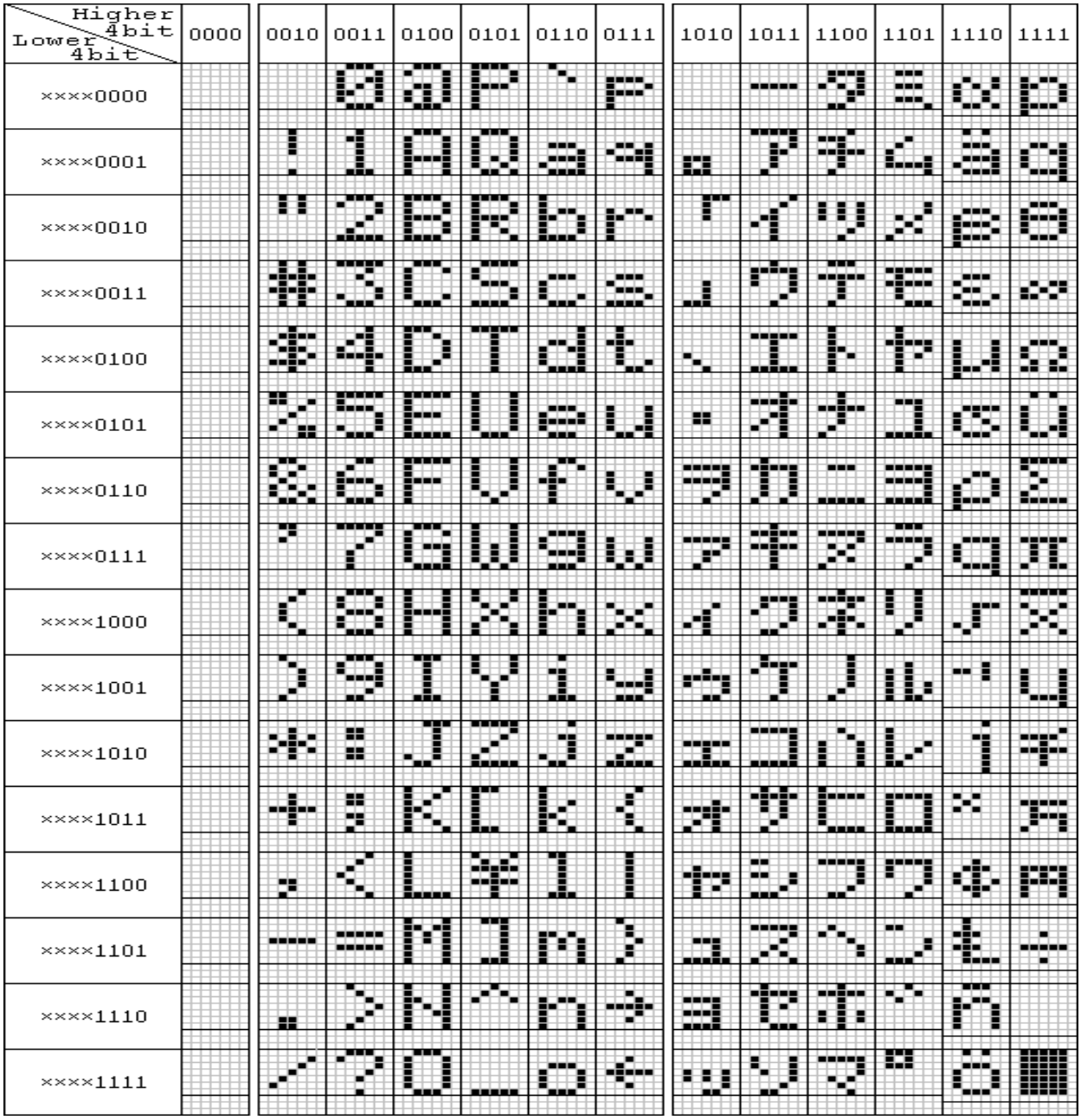
La CG RAM peut être rechargée périodiquement en fonction des besoins. Si un caractère de la CG RAM qui est actuellement sur l'afficheur est changé, alors le changement est immédiatement apparent sur l’afficheur.

c) Le jeu de caractères standard

Les afficheurs, qui permet d'afficher l'ensemble des caractères ASCII, quelques symboles et même un jeu de caractères japonais stylisé, utilisent tous le même type de contrôleur. Les caractères accentués français ne sont pas considérés par ces contrôleurs. Fort heureusement, on peut tout de même afficher tous les caractères désirés grâce à des caractères définissables par l'utilisateur. Ceux-ci sont au nombre de huit et correspondent aux caractères de 0 à 7.

L'afficheur est en mesure d'afficher 200 caractères :

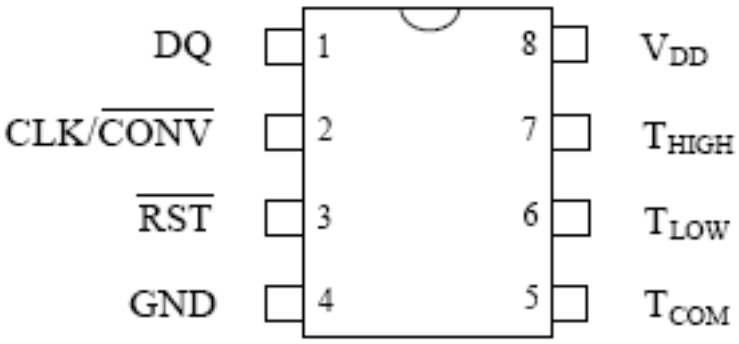
- de 00H à 07H : 8 caractères définissables par l'utilisateur.
- de 20H à 7FH : 96 caractères ASCII excepté le caractère \ qui est remplacé par le signe ¥.
- de A0H à DFH : 64 caractères japonais (alphabet kana).
- de E0H à FFH : 32 caractères spéciaux (accent, lettres grecques, …).



3.1.3) Capteur de température

a) Présentation et caractéristique

Concernant notre application, nous allons utiliser le capteur de température numérique DS1620.



DS1620 8-Pin DIP (300-mil)

Le tableau suivant est issu de la datasheet du composant et explique le rôle de chaque pin, ainsi que celui des registres présents dans le composant :

DETAILED PIN DESCRIPTION Table 1

PIN	SYMBOL	DESCRIPTION
1	DQ	**Data Input/Output pin** for 3-wire communication port.
2	CLK/$\overline{\text{CONV}}$	**Clock input pin** for 3-wire communication port. When the DS1620 is used in a stand-alone application with no 3–wire port, this pin can be used as a convert pin. Temperature conversion will begin on the falling edge of $\overline{\text{CONV}}$.
3	$\overline{\text{RST}}$	**Reset input pin** for 3-wire communication port.
4	GND	**Ground pin.**
5	T_{COM}	**High/Low Combination Trigger**. Goes high when temperature exceeds TH; will reset to low when temperature falls below TL.
6	T_{LOW}	**Low Temperature Trigger.** Goes high when temperature falls below TL.
7	T_{HIGH}	**High Temperature Trigger.** Goes high when temperature exceeds TH.
8	V_{DD}	**Supply Voltage.** 2.7V – 5.5V input power pin.

Table 2. DS1620 REGISTER SUMMARY

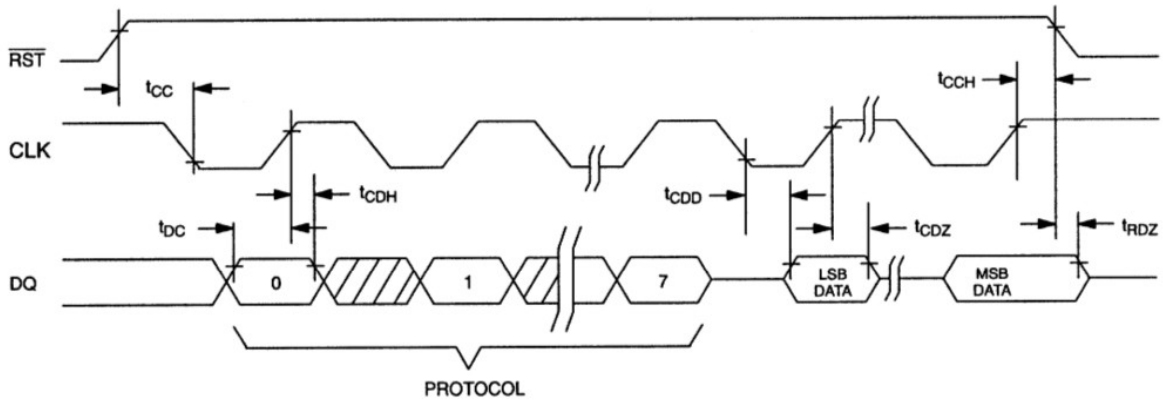
REGISTER NAME (USER ACCESS)	SIZE	MEMORY TYPE	REGISTER CONTENTS AND POWER-UP/POR STATE
Temperature (Read Only)	9 Bits	SRAM	Measured Temperature (Two's Complement) Power-Up/POR State: -60ºC (1 1000 1000)
T_H (Read/Write)	9 Bits	EEPROM	Upper Alarm Trip Point (Two's Complement) Power-Up/POR State: User-Defined. Initial State from Factory: +15°C (0 0001 1110)
T_L (Read/Write)	9 Bits	EEPROM	Lower Alarm Trip Point (Two's Complement) Power-Up/POR State: User-Defined. Initial State from Factory: +10°C (0 0001 0100)

b) Écriture et lecture

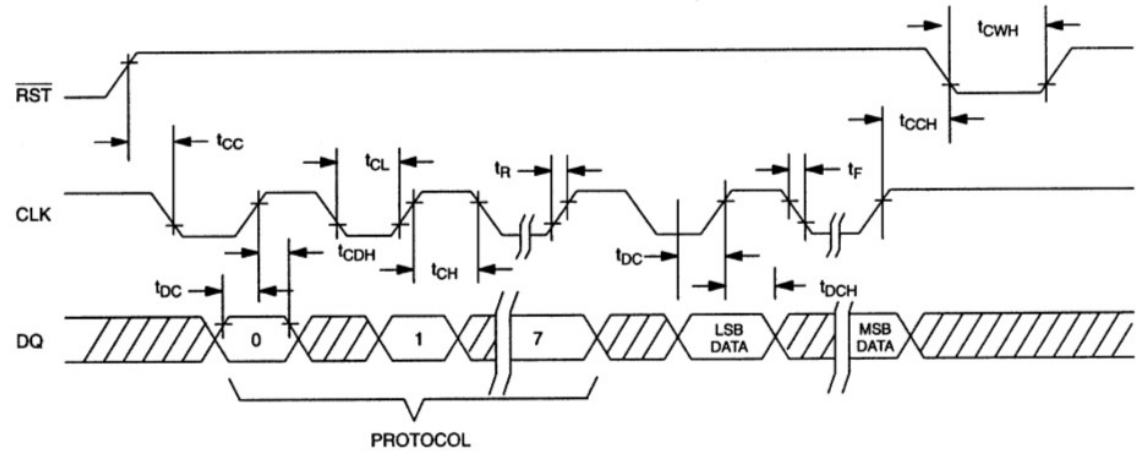
Ici l'écriture correspond à l'envoie de commande de configuration ou de requêtes (tel que la demande de mesure …). Et la lecture correspond à la réception des données du capteur. Sachant que la valeur reçue est numérique et que la précision du capteur est de 0.5°C, alors la réception de la donnée 45 correspond à la température (45/2) = 27.5°C.

Les deux chronogrammes suivants sont tirés de la datasheet du composant et illustrent parfaitement les 2 opérations de lecture et d'écriture:

READ DATA TRANSFER Figure 4



WRITE DATA TRANSFER Figure 5

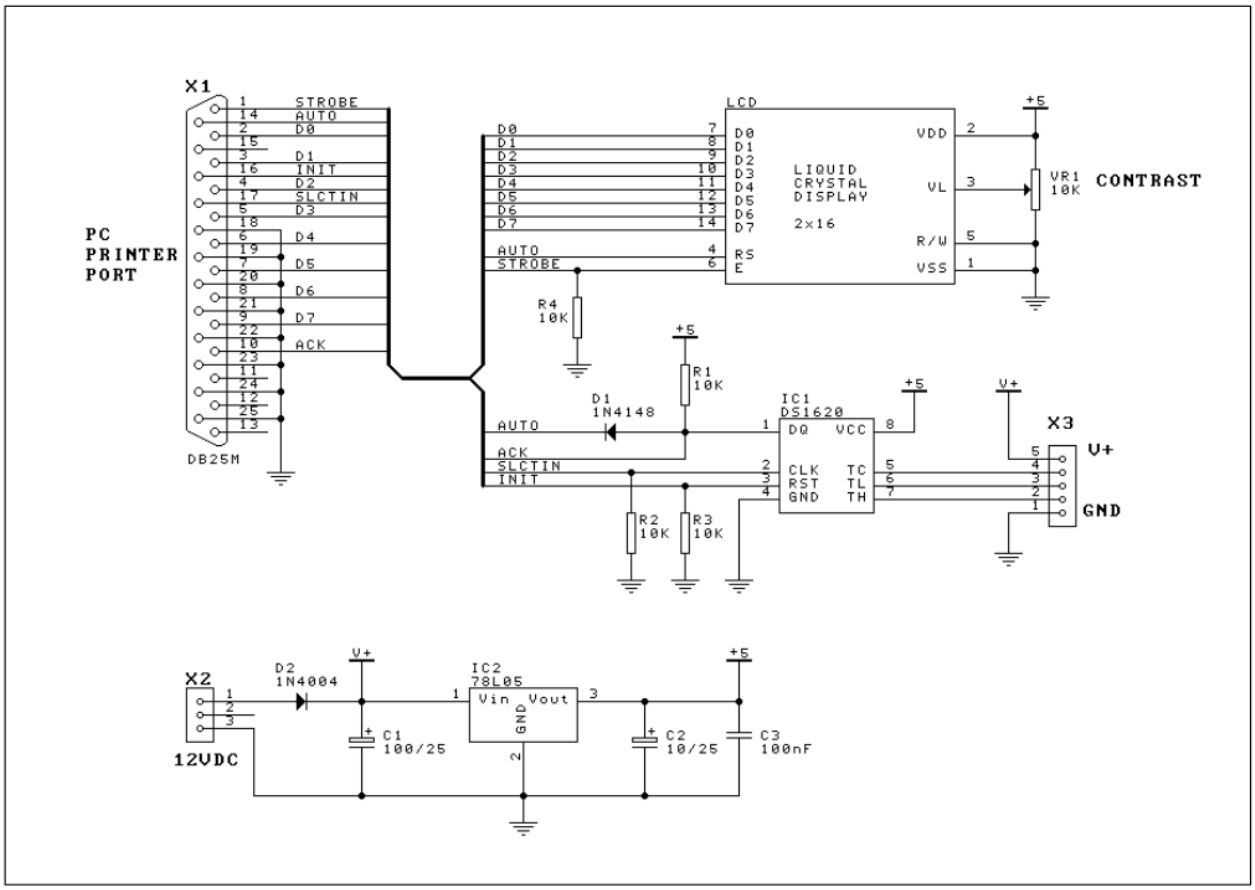


NOTE: t_{CL}, t_{CH}, t_R, and t_F apply to both read and write data transfer.

3.1.4) Carte utilisée pour les TP

Durant les TP nous avons utilisé une carte électronique sur laquelle le LCD et le capteur de température sont tous les deux connectés à une broche parallèle.

L'image suivante illustre le schéma électronique de la carte:



Le contrôle de la carte se fait à travers 3 ports:

DATA PORT : 0x378
STATUS PORT : 0x379
CONTROL PORT : 0x37A

3.2) Partie 1: écriture d'un programme en tant que super-utilisateur

3.2.1) Objectifs et intérêt

Dans cette partie du TP nous allons, grâce à la datasheet et aux explications vues en cours, écrire un programme permettant d'initialiser le LCD et le capteur de température. On pourra les contrôler ainsi, via le port parallèle du PC.

Ici le code sera écrit pour la partie utilisateur et comprendra des appels systèmes du type «usleep()», mais également «ioperm()» pour les permissions en lecture et écriture. Le code ne fonctionnera que s'il est exécuté en tant que super-utilisateur, il est donc nécessaire de se mettre en root avant d'essayer d'exécuter le code.

> ***Remarque:*** *le fichier lcdtemp_etu.c contenant le code de cette partie se trouve dans le répertoire TP1*

3.2.2) Travail réalisé en TP

a) Définition des adresses pour les ports

Dans un premier temps on commence par déclarer l'adresse de base ainsi que celles des différents ports (data, status, et contrôle):

```
#define PORT_BASE        0x378
#define DATA_BASE        PORT_BASE
#define STATUS_BASE      PORT_BASE + 1
#define CONTROL_BASE  PORT_BASE + 2
```

La fonction tem_temp(int d) prend en paramètre un entier et fait appel à l'appel système usleep(). Il permet de faire une temporisation de «d» millisecondes. Dans notre fonction, on multiplie la variable «d» par 1000 car l'appel system usleep() prend en paramètre un temps en micro-seconde.

```
void tme_tempo(int d)
{
        usleep(d*1000);
}
```

b) Initialisation

Le fonction PORT_INIT(), permet d'accorder les droit d'accès au processus sur les ports et broches voulus via la fonction:

```
int ioperm(unsigned long from, unsigned long num, int turn_on);
```

ioperm() positionne les bits de permission d'accès du processus appelant aux ports commençant à l'adresse ***from*** étalés sur ***num*** octets à la valeur ***turn_on***. Si ***turn_on*** n'est pas nul, le processus appelant doit être privilégié.

Puis on initialise ces dernières pour configurer le périphérique, et cela de la façon suivante:

SLCTIN à 1 (broche 17, bit 3 de 0x37A)
INIT à 0 (broche 16, bit 2 de 0x37A)
AUTO à 0 (broche 1, bit 1 de 0x37A)
STROBE à 0 (broche 14, bit 0 de 0x37A)

```
void PORT_INIT(void)
{
        ioperm(DATA_PORT, 3, 1);
        outb(3,CONTROL_PORT);
        tme_tempo(10);
}
```

c) Utilisation des fonctions inb() et outb()

```
unsigned char inb(unsigned short int port);
void outb(unsigned char value, unsigned short int port);
```

Cette famille de fonction est utilisée pour des entrées-sorties de bas niveau. Les fonctions out* effectuent une écriture sur un port alors que les fonctions in* effectuent une lecture. Les fonctions suffixées avec «b» travaillent sur des octets alors que les fonctions suffixées avec «w» travaillent sur des mots. Les fonctions suffixées avec «_p» attendent que les entrées-sorties soient achevées. Elles ont principalement été conçues pour un usage interne au noyau mais elles sont quand même utilisables avec des processus utilisateurs.

Ces fonctions seront utilisées dans plusieurs autres fonctions afin de pouvoir modifier les valeurs des différentes broches du port parallèle. On pourra ainsi piloter le périphérique, mais aussi lire les données en sortie.

Pour mettre une broche d'un port à 0:

```
outb( inb(Adresse_du_port) & ~(1<<numéro_de_broche), Adresse_du_port))
```

Pour mettre une broche d'un port à 1:

```
outb( inb(Adresse_du_port) | (1<<numéro_de_broche), Adresse_du_port);
```

Par la suite, il y a plusieurs broches dont nous avons besoin pour piloter:

- Le LCD:
  - ✗ broche E
  - ✗ broche RS

- Le capteur de température DS1620:
  - ✗ broche DQ
  - ✗ broche CLK
  - ✗ broche RESET

Il faudra faire attention à la polarisation des broches. En effet, certaines broches sont montées en inverse. Pour pouvoir mettre ces broches à 0, il faudra mettre la broche associée du port parallèle à 1 et non pas à 0.

Les broches dont la polarisation est inversée sont:

- ✗ E
- ✗ RS
- ✗ DQ
- ✗ CLK

La broche dont la polarisation est directe est:

- ✗ RESET

Ci-dessous la liste des fonctions concernées:

```
void LCD_E_HIGH(void)
{
	outb( inb(CONTROL_PORT) & ~(1<<0), CONTROL_PORT);
}
void LCD_E_LOW(void)
{
	outb( inb(CONTROL_PORT) | (1<<0), CONTROL_PORT);
}
void LCD_RS_HIGH(void)
{
	outb( inb(CONTROL_PORT) & ~(1<<1), CONTROL_PORT);
}
void LCD_RS_LOW(void)
{
	outb( inb(CONTROL_PORT) | (1<<1), CONTROL_PORT);
}
void DS1620_RESET_HIGH(void)
{
	outb( inb(CONTROL_PORT) | (1<<2), CONTROL_PORT);
}
void DS1620_RESET_LOW(void)
{
	outb( inb(CONTROL_PORT) & ~(1<<2), CONTROL_PORT);
}
void DS1620_CLK_HIGH(void)
{
	outb( inb(CONTROL_PORT) & ~(1<<3), CONTROL_PORT);
}
void DS1620_CLK_LOW(void)
{
	outb( inb(CONTROL_PORT) | (1<<3), CONTROL_PORT);
}
void DS1620_DQ_HIGH(void)
{
	outb( inb(CONTROL_PORT) & ~(1<<1), CONTROL_PORT);
}
void DS1620_DQ_LOW(void)
{
	outb( inb(CONTROL_PORT) | (1<<1), CONTROL_PORT);
}
```

d) Émission de données

Pour émettre des données du port parallèle vers les périphériques, on utilise encore une fois la fonction outb (le fonctionnement de cette dernière à été précédemment expliqué). Comme précédemment, dans la partie dédiée au LCD et au capteur de température, il est impératif de générer des trames de données avec une variation de la broche E pour que la donnée soit prise en compte par le périphérique. Entre ces variations, on intégrera un temps de latence d'une milliseconde afin de bien avoir un signal rectangulaire sur E (génération d'un pulse) et ainsi permettre à la donnée émie, d'être prise en compte. Pour envoyer une chaîne de caractères il suffit d'envoyer caractère par caractère jusqu'à ce que celle-ci soit vide.

```
// permet de mettre une valeur dans le registre DATA du port parallèle
void LCD_DATA (unsigned char data)
{
  /* Positionne les données sur les entrées D0 à D7 du LCD
     On écrit le caractère dans le registre 0x378 */
                outb(data, DATA_PORT);
}

// permet d'envoyer une commande vers le port parallèle . Et de générer un pulse sur E
void LCD_CMD(unsigned char data)
{
                LCD_DATA( data);
                LCD_RS_LOW();
                tme_tempo(1);
                LCD_E_HIGH();
                tme_tempo(1);
                LCD_E_LOW();
                tme_tempo(1);
}

// permet d'envoyer une caractère vers le port parallèle . Et de générer un pulse sur E
void LCD_CHAR(unsigned char data)
{
                LCD_DATA( data);
                LCD_RS_HIGH();
                tme_tempo(1);
                LCD_E_HIGH();
                tme_tempo(1);
                LCD_E_LOW();
                tme_tempo(1);
}

// permet d'envoyer une chaine de caracteres vers le LCD
void LCD_STRING(unsigned char* pdata)
{
        while (*pdata)
        {
                LCD_CHAR(*pdata);
                pdata++;
        }
}
```

e) Commande et initialisation du LCD

Pour contrôler le LCD, il suffit simplement de lui envoyer des commandes bien spécifiques en utilisant la fonction *LCD_CMD()*.

Pour effacer le contenu du LCD, il faut envoyer la commande 0x01:

```
void LCD_CLEAR(void)
{
        LCD_CMD(0x01);
}
```

Pour remettre le curseur du LCD en première ligne et première colonne, il faut envoyer la commande 0x02:

```
void LCD_HOME(void)
{
        LCD_CMD(0x02);
}
```

Pour initialiser ou réinitialiser le LCD, il faut envoyer une suite de commandes sur 5 octets 0x38, 0x38, 0x38, 0x0C, 0x06:

```
void LCD_INIT(void)
{
        LCD_CMD(0x38);
        LCD_CMD(0x38);
        LCD_CMD(0x38);
        LCD_CMD(0x0C);
        LCD_CMD(0x06);
}
```

f) Envoie de commande sur le capteur de température

La fonction suivante permet d'envoyer une commande sur 8 bits au thermomètre. La commande est envoyée bit après bit, en commençant par le bit de poids faible. Les bits sont envoyés sur la broche DQ. Donc pour envoyer un 1, on met la broche en HIGH et LOW pour envoyer un 0.
De plus, on génère une horloge en effectuant le même procédé sur les broches CLK et en introduisant à chaque variation, des temps de latence d'une milliseconde afin de reproduire les chronogrammes vus dans la section précédente (voir le chapitre sur le capteur de température).

A l'aide d'un printf() on affichera sur l'écran les bits envoyés au fur et à mesure. Et la fonction fflush() permet de vider le tampon du système.

```
void DS1620_WRITECOMMAND(unsigned char command)
{
	int i;

	for(i = 0; i<8 ; i++)
	{
		DS1620_CLK_LOW();
		tme_tempo(1);
		if (command & 1)
		{
			DS1620_DQ_HIGH();
			printf("1 "); fflush(stdout);
		}
		else
		{
			DS1620_DQ_LOW();
			printf("0 "); fflush(stdout);
		}
		tme_tempo(1);
		DS1620_CLK_HIGH();
		tme_tempo(1);
		command >>=1;
	}
	printf("\n");
}
```

g) Lecture de la température sur le capteur

La fonction suivante permet de lire la température. Celle-ci est codée sur 9 bits avec une précision d'un demi degré (voir chapitre sur le capteur de température).

Ici on reprend exactement le même principe que précédemment à un détail près. Une fois l'horloge générée à la place d'écrire sur la broche DQ, on va lire la broche bit à bit (un bit par période) pour construire un entier avec les 9 bits reçus puis le retourner.

Une fois encore, on utilisera la fonction printf() pour afficher à l'écran les bits reçus, et la fonction ffluch() pour vider le tampon système. Il est à noter que la lecture se fait à partir du bit de poids faible.

```
int DS1620_READ(void)
{
	int ret=0;

	int i;

	DS1620_DQ_HIGH();

	for(i = 0; i<9 ; i++)
	{
		DS1620_CLK_LOW();
		if (inb(STATUS_PORT) & 0x40)
		{
			ret+=1<<i;
			printf("1 "); fflush(stdout);
		}
		else
		{
			printf("0 "); fflush(stdout);
		}

		tme_tempo(1);
		DS1620_CLK_HIGH();
		tme_tempo(1);
	}
	printf("\n");

	return ret;
}
```

h) Programme principal

Dans le programme principal, on initialise les ports, puis le LCD. On affiche à l'écran le message «L & M», on met la broche RESET du thermomètre à un niveau haut et on envoie la commande STARTCONVERT qui correspond à une requête de début de traitement pour le capteur.

Dans la boucle principale, on envoie une commande de lecture de température de façon continue. On lit par la suite, la température que l'on divise par 2 (à cause de la précision qui est d'un demi degré). Cette dernière est affichée sur l'écran LCD, précédé du message «Température: ».

```
int main(int argc,char *argv[])
{
	int temp;
	char tempstring [16];

	tme_tempo(250);
	PORT_INIT();
	LCD_INIT();
	LCD_CLEAR();
	LCD_STRING("L & M ");
	tme_tempo(1000);

	DS1620_RESET_HIGH();
	DS1620_WRITECOMMAND(STARTCONVERT);
	DS1620_RESET_LOW();
	tme_tempo(1000);

	while (1)
	{
		DS1620_RESET_HIGH();
		DS1620_WRITECOMMAND(READTEMP);
		temp=DS1620_READ();
		DS1620_RESET_LOW();
		tme_tempo(1000);
		sprintf(tempstring,"Temperature %1.1f"',temp/2.0);
		LCD_CLEAR();
		LCD_STRING(tempstring);
	}
	return 0;
}
```

3.3) Partie 2 : écriture d'un programme en tant qu'utilisateur (ppdev)

3.3.1) Objectifs et intérêt

Dans cette partie du TP, nous allons écrire un code permettant d'accéder au périphérique sans avoir à être en super-utilisateur (root). Pour cela, nous allons utiliser le module noyau PPDEV. L'objectif est de reprendre le code du **TP1,** de modifier les fonctions d'accès au pin telles que **outb** et **inb,** et d'avoir une approche du système. C'est-à-dire qu'ici les périphériques sont vus comme un fichier et nous allons donc manipuler cette structure afin de contrôler les périphériques. Plus explicitement, nous aurons besoin de faire appel aux appels système standard pour la gestion de fichier Linux c'est-à-dire: **open, close, write, et read, (ioctl)**.

Ainsi faire un:

- Open correspondra à l'ouverture dufichier.
- Close correspondra à la fermeture du fichier.
- Read correspondra à la lecture du fichier.
- Write correspondra à l'écriture sur le fichier.
- Ioctl correspondra à la configuration du périphérique.

> ***remarque:*** *le fichier lcdtemp_etu_ppdev.c contenant le code de cette partie se trouve dans le répertoire TP2*

3.3.2) Module noyau PPDEV

Le module noyau **ppdev** fait partie du système Linux standard. C'est un gestionnaire de périphérique de type caractère (**char device driver**) qui permet de piloter individuellement les broches du port parallèle depuis une application utilisateur. Avec ce module, il n'est plus nécessaire d'être root pour commander l'écran LCD et le thermomètre.

Avec ppdev, l'accès au port parallèle se fait de manière canonique, comme avec tout gestionnaire de périphérique. En particulier, le port parallèle est représenté comme un fichier dont le nom est **/dev/parport0**. Pour le manipuler, la méthode standard consiste à ouvrir ce fichier avec l'appel système open(), puis à écrire dedans au moyen de write() ou de lire des données avec read(). Pour configurer le périphérique dans un certain mode, il faut utiliser l'appel système ioctl(). Pour notre application avec **ppdev**, seul l'appel système ioctl() va en fait être utilisé.

Pour pouvoir profiter de ce module noyau, il faut vérifier qu'il est bien installé sur notre système. Pour cela, il suffit de lancer la commander **/sbin/lsmod** qui permet de lister les modules noyau actuellement opérationnels et d'y repérer le module ppdev. On peut aussi vérifier que le fichier représentant le périphérique, **/dev/parport0**, existe bien.

int **open**(char *pathname, int mode)

Cet appel système ouvre le fichier de chemin d'accès pathname dans le mode mode. Pour accéder en écriture et en lecture dans ce fichier, l'argument mode doit être égal à O_RDWR. L'appel système retourne le numéro de descripteur de fichier ouvert ou -1 s'il y a eu une erreur.

```
int close( int filedesc )
```

Cet appel système permet de fermer le fichier de descripteur filedesc. A utiliser lorsque le programme se termine.

```
int ioctl(int filedesc, int cmd, ...)
```

Cet appel système est celui par lequel toutes vos actions vont s'effectuer. Il peut prendre de multiples formes :

int ioctl(fd, PPCLAIM) *doit être appelé dès que le fichier est ouvert, pour réclamer l'usage exclusif du périphérique port parallèle. Retourne -1 si erreur.*

int ioctl(fd, PPRELEASE) *doit être appelé avant que le fichier soit fermé, pour libérer l'usage exclusif du périphérique. Retourne -1 si erreur.*

int ioctl(fd, numreg, char *byte) *est la fonction que vous allez principalement utiliser. numreg peut prendre les valeurs PPRDATA (pour lire le port data), PPRSTATUS (pour lire le port d'état), PPRCONTROL (pour lire le port de contrôle), PPWDATA (pour écrire sur le port data), PPWCONTROL (pour écrire sur le port contrôle). Pour une lecture, *byte contient après appel la valeur lue. Pour une écriture, il faut affecter *byte avant appel. Retourne -1 si erreur.*

Ainsi, pour écrire la valeur 0xaa sur le port data (=0x378) du port parallèle, il faut écrire : ***(équivalent à outb())***

```
char byte=0xaa ;
if ( ioctl( fd, PPWDATA, &byte ))
{
        printf("erreur");
       exit(1);
}
```

Pour lire la valeur du registre d'état (=0x379) : ***(équivalent à inb())***

```
If ( ioctl(fd ,PPRSTATUS ,&byte ))
{
       printf("erreur");
      exit(1);
}
```

3.3.3) Travail réalisé en TP

Dans cette partie, nos accès aux périphériques se limiteront à l'utilisation des appels système standard du module PPDEV. En l'occurrence, le seul appel système utilisé sera ioctl(). Son but sera de remplacer les fonctions inb() et outb() qui sont des fonctions de bas niveau nécessitant d'être exécutées en mode super utilisateur.

a) Ouverture et fermeture de fichier

Avant toute manipulation d'écriture ou de lecture sur un fichier, il se doit tout naturellement d'ouvrir le fichier afin d'accéder à celui-ci. Une fois les modifications et/ou consultations sur le fichier terminées, il convient de refermer celui-ci avant d'arrêter le programme. Il est donc normal de commencer par expliquer ceci.

L'ouverture du fichier se fera lors de l'initialisation du port, et la fermeture du fichier, quant à elle, se fera à la fin de notre programme. Les deux appels système permettant d'effectuer cela sont open() et close(). Le code suivant montre leur utilisation et leur emplacement dans le code:

```
// Déclaration du descripteur de fichier en variable global
int file_des ;

void PORT_INIT(void)
{
	file_des = open("/dev/parport0", O_RDWR);

	//  réclame l'usage exclusif du périphérique port parallèle
	if(ioctl(file_des,PPCLAIM)){
		printf("erreur claim");
		exit(1);
	}

	unsigned char byte = 0x08;

	// equivallent a un outb()
	if(ioctl(file_des,PPWCONTROL,&byte)){
		printf("erreur");
		exit(1);
	}
	tme_tempo(10);
}
```

Le nom du fichier correspondant est parport0 et ce trouve dans dossier /dev/, donc l'ouverture se fait sur /dev/parport0 en Mode O_RDWR.

La fermeture de fichier se fera a la fin du main(). Hormis cette modification il n'y aura aucune autre modification à faire dans le programme principal:

```
int main(int argc,char *argv[])
{
	int temp;
	char tempstring [16];

	tme_tempo(250);
	PORT_INIT();
	LCD_INIT();
	LCD_CLEAR();
	LCD_STRING("L & M");
	tme_tempo(1000);

	DS1620_RESET_HIGH();
	DS1620_WRITECOMMAND(STARTCONVERT);
	DS1620_RESET_LOW();
	tme_tempo(1000);

	while (1)
	{
		DS1620_RESET_HIGH();
		DS1620_WRITECOMMAND(READTEMP);
		temp=DS1620_READ();
		DS1620_RESET_LOW();
		tme_tempo(1000);
		sprintf(tempstring,"Couscous %1.1f C",temp/2.0);
		LCD_CLEAR();
		LCD_STRING(tempstring);
	}

	// Fermeture du fichier
	close(file_des);

	return 0;
}
```

Toutes les fonctions ne feront pas appel directement à outb() ou inb() et ne seront pas modifiées. Vous trouverez ci-dessous la liste de ces fonctions qui resteront inchangées par rapport au TP1:

```
void LCD_INIT(void);
void LCD_HOME(void);
void LCD_CLEAR(void);
void LCD_CMD(unsigned char data);
void LCD_CHAR(unsigned char data);
void LCD_STRING(unsigned char* pdata);
void DS1620_WRITECOMMAND(unsigned char command);
```

b) Écriture et lecture sur le port de contrôle via le fichier

Lors de l'écriture ou de la lecture du fichier pour le contrôle du LCD ou du capteur de température, la commande qui est passée à la fonction ioctl (voir structure de la fonction ioctl dans la partie 3) est:

PP**R**CONTROL: pour la Lecture (R pour read) → remplace inb().
PP**W**CONTROL: pour l'écriture (W pour write) → remplace outb().

Pour la lecture on remplacera inb() par:

```
// déclaration de la variable pour la réception de la lecture
unsigned char byte;

// Utilisation de ioctl() avec la commande PPRCONTROL
if ( ioctl ( file_des, PPRCONTROL, &byte) )
{
        printf("erreur");
        exit(1);
}
```

Pour l'écriture on remplacera outb() par:

```
// masquage du bit qu'on veux modifier (ici mise à zéro du bit 0)
byte = byte & 0xFE;

// Utilisation de ioctl() avec la commande PPWCONTROL
if ( ioctl ( file_des, PPWCONTROL, &byte) )
{
        printf("erreur");
        exit(1);
}
```

Les fonctions concernées par ce changement sont toutes celles qui font appel à inb() et outb(), vous trouverez ci-dessous la liste des fonctions (voir le code pour plus de détails):

```
void LCD_E_HIGH(void) ;
void LCD_E_LOW(void) ;
void LCD_RS_HIGH(void);
void LCD_RS_LOW(void) ;
void DS1620_DQ_HIGH(void);
void DS1620_DQ_LOW(void);
void DS1620_CLK_LOW(void);
void DS1620_CLK_HIGH(void);
void DS1620_RESET_LOW(void);
void DS1620_RESET_HIGH(void);
```

c) Écriture et lecture sur le port de données via le fichier

Pour pouvoir écrire et lire sur le port data du port parallèle on utilisera cette fois-ci la commande PPWDATA et PPRSTATUS pour ioctl():

PP**W**DATA: pour l'écriture (W pour write)
PP**R**STATUS: pour la lecture (R pour read)

L'écriture se fera sur le LCD via la fonction LCD_DATA():

```
void LCD_DATA (unsigned char data)
{
	// Équivalent à outb(data ,port_base_addr);
	if(ioctl(file_des,PPWDATA,&data)){
		printf("erreur");
		exit(1);
	}
}
```

La lecture se fera sur le capteur de température via la fonction DS1620_READ():

```
int DS1620_READ(void)
{
	int ret=0,temp = 0;
	//DS1620_CLK_HIGH();
	int i;
	unsigned char byte = 0;

	DS1620_DQ_HIGH();
	tme_tempo(2);
	for( i = 0; i < 9 ; i++){
		byte = 0;
		DS1620_CLK_LOW();
		tme_tempo(2);
		if(ioctl(file_des,PPRSTATUS,&byte)){
			printf("erreur lecture");
			exit(1);
		}
		ret += ((byte >> 6) & 1) << i;
		tme_tempo(2);
		DS1620_CLK_HIGH();
		tme_tempo(2);
	}

	return ret;
}
```

i

3.3.4) Bonus: caractères spéciaux

Cette partie correspond à une extension ou plutôt, à une application pratique de l'utilisation des caractères spéciaux programmable de l'écran LCD dont on a parlé précédemment dans la section LCD.
En effet, certains caractères peuvent être programmés par l'utilisateur. Le principe est simple: chaque caractère est représenté par un tableau de 5*8 bits sur lequel on peut «dessiner» nos propres caractères. Afin d'avoir une représentation binaire de 5*8 bits, on définit une structure comme le montre l'image ci-dessous:

EGQ	Caractère n°1 :					Caractère n°2 :					Caractère n°3 :				
	B4	B3	B2	B1	B0	B4	B3	B2	B1	B0	B4	B3	B2	B1	B0
Rangée 1 :	1	1	1	1	1	0	1	1	1	0	0	1	1	1	0
Rangée 2 :	1	0	0	0	0	1	0	0	0	1	1	0	0	0	1
Rangée 3 :	1	0	0	0	0	1	0	0	0	0	1	0	0	0	1
Rangée 4 :	1	1	1	0	0	1	0	0	1	1	1	0	0	0	1
Rangée 5 :	1	0	0	0	0	1	0	0	0	1	1	0	1	0	1
Rangée 6 :	1	0	0	0	0	1	0	0	0	1	1	0	0	1	0
Rangée 7 :	1	1	1	1	1	0	1	1	1	0	0	1	1	0	1

Schéma non contractuel

Ici on peut voir que pour définir le caractère «E» par exemple, la suite de bit à fournir est la suivante:

1 1 1 1 1
1 0 0 0 0
1 0 00 0
1 1 1 0 0
1 0 0 0 0
1 0 0 0 0
1 1 1 1 1

Les '1' forment les points visibles du caractère. En théorie rien de bien compliqué. Cependant, en pratique, le LCD utilise les différentes mémoires qui sont à sa disposition pour enregistrer la majorité des caractères les plus utilisés. En effet, quand on sollicite le LCD pour écrire un «E», il va simplement chercher dans sa mémoire la suite de bit 8*5 qui encode ce caractère pour ensuite pouvoir l'afficher.

En se basant sur ce principe et sur la datasheet de l'écran LCD, nous allons voir maintenant une façon «simple» pour pouvoir concrètement créer nos propres caractères spéciaux ou autre.

L'affichage sur l'écran LCD se base sur le principe d'association de symboles à une structure de 8*5 bits («E» étant le symbole à envoyer au LCD pour qu'il puisse afficher la structure de 8*5 bits correspondant au caractère 'E').

En partant de ce principe, nous avons écrit notre propre fonction qui permet d'ajouter à la mémoire du LCD des associations entre symbole et structure de bits que nous aurons nous même définit. De plus, il faudra faire attention à ne pas écraser des symboles déjà programmés. A cet effet, on dispose de plusieurs espaces libres dans la mémoire comme on peut le voir sur le tableau des caractères du LCD (tableau présenté dans la section LCD).

Pour notre application, on a choisi de créer une fonction qui permet de modifier les 8 cases correspondants à des caractères vides. Comme on peut voir sur la photo, on peut en programmer davantage et même en effacer certain (plus compliqué car la mémoire est protégée en écriture). Donc la fonction suivante permet de donner un nombre qui correspondra à notre symbole et une structure de donnée (le symbole sera associé à la structure pour afficher le caractère décrit dans cette structure).

Remarque: *Ici, seuls les nombres entre 0 et 7 sont acceptés (3 bits). Tout autre symbole donné sera remplacé par zéro donc écrasera la case associée à 0:*

Associe le nombre (symbole) «location» à la structure «charmap»:

```
void create_custom_char(char location, unsigned char charmap[])
{
	location &= 0x7;

	//SETCGRAMADDR
	LCD_CMD(0x40 | (location << 3));
	int i = 0;
	for (i = 0; i < 8; i++)
		LCD_CHAR((unsigned char)charmap[i]);
}
```

L'utilisation est simple, on envoie une commande au LCD avec l'emplacement voulu puis on envoie nos 8 séries de 5bits. Ici, on envoie 8 séries de char donc 8 séries de 8 bits sachant que les 3 bits de poids fort ne seront pas pris en compte et donc mis à zéro:

CMD
b0
b1
b2
b3
b4
b5
b6
b7

Une fois cette fonction ajoutée à notre code, il suffit de créer nos propres structures de bits en formant 8*5. Pour cela on créera des tableaux de char de taille 8. Donc 8*8 où seuls les 5 bits de poids faible ont leurs importances et donc programmables à notre guise.

```
// Ici on crée notre tableau de 8 char avec notre propre caractère
unsigned char myChar_01[8] = {
                0b10001,
                0b11011,
                0b11111,
                0b00000,
                0b01010,
                0b00000,
                0b10101,
                0b01010
        };

// puis on appelle la fonction pour lui affecter le symbole 1
create_custom_char(1, myChar_01);
```

Pour pouvoir associer notre caractère au symbole **1**:
create_custom_char(1, myChar_01);

Pour pouvoir associer notre caractère au symbole **2**:
create_custom_char(2, myChar_01);

et ainsi de suite …

Une fois notre caractère créée et ajouté au tableau des symboles du LCD, il nous suffira de l'appeler dans un **printf()** en utilisant le symbole auquel il est rattaché.

Si le caractère est rattaché au symbole 1:

printf("mon caractère %c", 1);

affiche la chaîne de caractère «mon caractère» suivie de notre caractère programmé.

Même chose si le caractère est rattaché au symbole 2:

printf("mon caractère %c", 2);

affiche la chaîne de caractère «mon caractère» suivie de notre caractère programmé.

Remarque: *Tout le code correspondant aux explications précédentes se trouve dans le fichier lcdtemp_etu_ppdev.c dans le dossier TP2*

3.4) Partie 3: écriture du module noyau

3.4.1) Objectifs et intérêt

Dans cette partie, nous allons écrire nous même notre propre module noyau qui va contrôler nos deux périphériques et nous permettra de les utiliser à partir de l'espace utilisateur.
Nous allons d'abord commencer par vous présenter la structure d'un module noyau de type caractère, puis expliquer comment créer le module noyau pour le LCD et le capteur de température à partir de cette même structure.

> ***Remarque:*** *les fichiers contenant les codes de cette partie se trouvent dans le répertoire TP3*

3.4.2) Structure d'un module noyau

a) Débogage en mode noyau

Le débogage en mode noyau peut se faire via plusieurs outils.

- En mode console, via **printk()** et **dmesg** pour voir les messages.
- Avec **kgdb**, nécessite une seconde machine reliée par câble série et possédant GDB-client pour déboguer la cible.
- Avec **kdb**, un déboguer noyau embarqué.

Prototype:

```
int printk(const char *fmt, ...)
```

Exemple:

```
printk("<1> Hello World !\n");
```

Plusieurs niveaux de débogage sont définis dans **<linux/kernel.h>**

```
#define KERN_EMERG      "<0>"  /* système inutilisable */
#define KERN_ALERT      "<1>"  /* action à effectuer immédiatement*/
#define KERN_CRIT       "<2>"  /* conditions critiques */
#define KERN_ERR        "<3>"  /* conditions d'erreurs */
#define KERN_WARNING    "<4>"  /* message d'avertissement */
#define KERN_NOTICE     "<5>"  /* normal mais significatif */
#define KERN_INFO       "<6>"  /* informations */
#define KERN_DEBUG      "<7>"  /* messages de débugging */
```

Du coup cela devient :

```
printk(KERN_ALERT "Hello World !\n");
```

Pour afficher les messages sur la console on utilise la commande: **dmesg.**

b) Chargement et déchargement de module

Le chargement d'un module noyau se fait avec la commande **insmod**. De même, le déchargement se fait avec la commande **rmmod** qui appelle respectivement les fonctions ci-dessous:

- ➔ La fonction **module_init()** qui doit s'occuper de préparer le terrain pour l'utilisation de notre module (allocation mémoire, initialisation matérielle...).
- ➔ La fonction **module_exit()** qui doit, quant à elle, défaire ce qui a été fait par la fonction module_init().

***Remarque** : Un module ne possède pas de fonction main().*

Voici le source minimum d'un module:

```
#include <linux/module.h>
#include <linux/init.h>

static int __init mon_module_init(void)
{
        printk(KERN_DEBUG "Hello World !\n");
        return 0;
}

static void __exit mon_module_cleanup(void)
{
        printk(KERN_DEBUG "Goodbye World!\n");
}

module_init(mon_module_init);
module_exit(mon_module_cleanup);
```

Vous pouvez ainsi appeler vos **init** et **cleanup** comme bon vous semble.

Pour compiler, il suffit d'utiliser cette commande:
 # make

Avec le makefile suivant :

```
obj-m += module.o

default:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Le code suivant permet de tester le chargement et déchargement du module et d'afficher les messages via la commande dmesg:

```
$ insmod ./module.ko
$ lsmod
$ rmmod module.ko
$ dmesg
```

c) Description de module

Vous pouvez décrire vos modules à l'aide des différentes macros mises à votre disposition dans **<linux/module.h>**.

MODULE_AUTHOR(nom) : place le nom de l'auteur dans le fichier objet.

MODULE_DESCRIPTION(desc) : place une description du module dans le fichier objet.

MODULE_SUPPORTED_DEVICE(dev) : place une entrée indiquant le périphérique pris en charge par le module.

MODULE_LICENSE(type) : indique le type de licence du module.

On peut obtenir ces informations avec la commande **modinfo nom_module**.

```
#define MODULE
#include <linux/module.h>
#include <linux/init.h>

MODULE_AUTHOR("skyrunner");
MODULE_DESCRIPTION("exemple de module");
MODULE_SUPPORTED_DEVICE("none");
MODULE_LICENSE("none");

static int __init mon_module_init(void)
{
	printk(KERN_DEBUG "Hello World !\n");
	return 0;
}

static void __exit mon_module_cleanup(void)
{
	printk(KERN_DEBUG "Goodbye World!\n");
}

module_init(mon_module_init);
module_exit(mon_module_cleanup);
```

d) Passage de paramètres

Comme on peut s'en douter, il serait bien utile de passer des paramètres à notre module.
Une fonction et une macro sont donc disponibles pour cela:

module_param(nom, type, permissions)

MODULE_PARM_DESC(nom, desc)

Plusieurs types de paramètres sont actuellement supportés pour le paramètre type : short (entier court, 2 octet), int (entier, 4 octets), long (entier long) et char (chaîne de caractères).

```
#include <linux/module.h>
#include <linux/init.h>

MODULE_AUTHOR("skyrunner");
MODULE_DESCRIPTION("exemple de module");
MODULE_SUPPORTED_DEVICE("none");
MODULE_LICENSE("none");

static int param;

module_param(param, int, 0);
MODULE_PARM_DESC(param, "Un paramètre de ce module");

static int __init mon_module_init(void)
{
        printk(KERN_DEBUG "Hello World !\n");
        printk(KERN_DEBUG "param=%d !\n", param);
        return 0;
}

static void __exit mon_module_cleanup(void)
{
        printk(KERN_DEBUG "Goodbye World!\n");
}

module_init(mon_module_init);
module_exit(mon_module_cleanup);
```

Dans le cas de ce tableau, vous devez utiliser la fonction : module_param_array(name, type, addr, permission).
addr est l'adresse d'une variable qui contiendra le nombre d'éléments initialisés par la fonction.

Voici un code permettant de tester:

```
$insmod ./module.o param=2
```

e) Ajout d'un driver au noyau

Lors de l'ajout d'un driver au noyau, le système lui affecte un nombre majeur. Ce nombre majeur a pour but d'identifier notre driver.

L'enregistrement du driver dans le noyau doit se faire lors de l'initialisation du driver (c'est à dire lors du chargement du module, donc dans la fonction **init_module()**), en appelant la fonction : **register_chrdev()**.

De même, on supprimera le driver du noyau (lors du déchargement du module dans la fonction **cleanup_module()**) en appelant la fonction : **unregister_chrdev()**.

Ces fonctions sont définies dans <linux/fs.h>

Prototypes :

```
int register_chrdev(unsigned char major, const char *name, struct file_operations *fops);

int unregister_chrdev(unsigned int major, const char *name);
```

Ces fonctions renvoient **0** ou **>0** si tout se passe bien.

register_chrdev

- **major** : numéro majeur du driver, 0 indique que l'on souhaite une affectation dynamique.
- **name** : nom du périphérique qui apparaîtra dans /proc/devices.
- **fops** : pointeur vers une structure qui contient des pointeurs de fonction. Ils définissent les fonctions appelées lors des appels systèmes (open, read...) du côté utilisateur.

unregister_chrdev

- **major** : numéro majeur du driver, le même qui est utilisé dans **register_chrdev**.
- **name** : nom du périphérique utilisé dans **register_chrdev.**

```
static ssize_t my_read_function(struct file *file, char *buf, size_t count, loff_t *ppos)
{
   printk(KERN_DEBUG "read()\n");
   return 0;
}

static ssize_t my_write_function(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
   printk(KERN_DEBUG "write()\n");
   return 0;
}

static int my_open_function(struct inode *inode, struct file *file)
{
   printk(KERN_DEBUG "open()\n");
   return 0;
}

static int my_release_function(struct inode *inode, struct file *file)
{
   printk(KERN_DEBUG "close()\n");
   return 0;
}
int my_ioctl_function(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
   printk(KERN_DEBUG "ioctl()\n");
   return 0;
}
```

f) Implémentation des appels systèmes

Ces fonctions renvoient 0 (ou >0) en cas de succès, une valeur négative sinon.

La structure permettant de faire appel aux différentes fonctions se présentent sous cette forme:

```
static struct file_operations fops =
{
   read : my_read_function,
   write : my_write_function,
   open : my_open_function,
   release : my_release_function,
   ioctl : my_iotl_function;
};
```

La structure file définie dans **<linux/fs.h>** représente un fichier ouvert et est créée par le noyau sur l'appel système **open()**. Elle est transmise à toutes fonctions qui agissent sur le fichier, jusqu'à l'appel de **close()**.

Remarque: *Certaines méthodes non implémentées sont remplacées par des méthodes par défaut. Les autres méthodes non implémentées retournent -EINVAL.*

Un fichier disque sera lui représenté par la structure inode. On n'utilisera généralement qu'un seul champ de cette structure:
kdev_t i_rdev : le numéro du périphérique actif

Ces macros nous permettront d'extraire les nombres majeurs et mineurs d'un numéro de périphérique:
int minor = MINOR(inode->i_rdev);
int major = MAKOR(inode->i_rdev);

Le *MAJEUR* (MAJOR): qui identifie le pilote.

Le *MINEUR* (MINOR): qui représente une sous-adresse en cas de présence de plusieurs périphériques identiques, contrôlée par un même pilote.

Les fonctions **copy_from_user** et **copy_to_user** servent à transférer un buffer depuis/vers l'espace utilisateur.

```
copy_from_user(unsigned long dest, unsigned long src, unsigned long len);
copy_to_user(unsigned long dest, unsigned long src, unsigned long len);
```

g) Méthodes open et release

En général, la méthode open réalise ces différentes opérations :

- Incrémentation du compteur d'utilisation
- Contrôle d'erreur au niveau matériel
- Initialisation du périphérique
- Identification du nombre mineur
- Allocation et remplissage de la structure privée qui sera placée dans file->private_data.

Le rôle de la méthode release est tout simplement le contraire

- Libérer ce que open a alloué
- Éteindre la périphérique
- Décrémenter le compteur d'utilisation

```
static int __init mon_module_init(void)
{
   int ret;

   ret = register_chrdev(major, "mydriver", &fops);

   if(ret < 0)
   {
      printk(KERN_WARNING "Probleme sur le major\n");
      return ret;
   }

   printk(KERN_DEBUG "mydriver chargé avec succès\n");
   return 0;
}

static void __exit mon_module_cleanup(void)
{
   int ret;

   ret = unregister_chrdev(major, "mydriver");

   if(ret < 0)
   {
      printk(KERN_WARNING "Probleme unregister\n");
   }

   printk(KERN_DEBUG "mydriver déchargé avec succès\n");
}

module_init(mon_module_init);
module_exit(mon_module_cleanup);
```

Une fois le driver compilé et chargé, il faut tester son lien avec le mode utilisateur et plus particulièrement les appels systèmes.

Tout d'abord, nous devons créer son fichier spécial : **mknod /dev/mydriver c 254 0**.

Nous pouvons désormais effectuer notre premier test : **$ cat mydriver.c > /dev/mydriver**.

Maintenant vous pouvez vérifier en tapant dmesg, et voir les appels à **open()**, **write()** et **release()**.

Vous pouvez aussi créer votre propre programme qui ouvre le périphérique et envoit une donnée, puis le referme.

Exemple:

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

int main(void)
{
	int file = open("/dev/mydriver", O_RDWR);

	if(file < 0)
	{
		perror("open");
		exit(errno);
	}

	write(file, "hello", 6);

	close(file);

	return 0;
}
```

h) Allocation mémoire

En mode noyau, l'allocation mémoire se fait via la fonction **kmalloc**, la dés-allocation mémoire se fait via **kfree**.

Ces fonctions, sont très peu différentes des fonctions de la bibliothèque standard. La seule différence, est un argument supplémentaire pour la fonction **kmalloc()** : la priorité.

Cet argument peut-être :

- **GFP_KERNEL** : allocation normale de la mémoire du noyau
- **GFP_USER** : allocation mémoire pour le compte utilisateur (faible priorité)
- **GFP_ATOMIC** : alloue la mémoire à partir du gestionnaire d'interruptions

```
#include <linux/slab.h>

buffer = kmalloc(64, GFP_KERNEL);
if(buffer == NULL)
{
   printk(KERN_WARNING "problème kmalloc !\n");
   return -ENONEM;
}
kfree(buffer), buffer = NULL;
```

i) Méthode ioctl

Dans le cadre d'un pilote, la méthode **ioctl** sert généralement à contrôler le périphérique.

Cette fonction permet de passer des commandes particulières au périphérique. Les commandes sont codées sur un entier et peuvent avoir un argument. L'argument peut-être un entier ou un pointeur vers une structure.

Prototype :

```
int ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg);
```

Cette fonction est définie dans **<linux/fs.h>**.

La fonction **ioctl**, côté utilisateur, a ce prototype:

```
int ioctl(int fd, int cmd, char *argp);
```

Une fonction type ressemble à :

```
int my_ioctl_function(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
   int retval = 0;

   switch(cmd)
   {
      case ... : ... break;
      case ... : ... break;
      default : retval = -EINVAL; break;
   }
   return retval;
}
```

j) Gestion des interruptions

Une interruption est un signal envoyé par un matériel et qui est capable d'interrompre le processeur.

Notre pilote, met donc en place un gestionnaire d'interruptions pour les interruptions de son périphérique.

Un gestionnaire d'interruptions est déclaré par cette fonction : **request_irq()**.

Il est libéré par cette fonction : **free_irq()**.

Les prototypes définis dans **<linux/sched.h>** sont les suivants:

```
int request_irq(unsigned int irq,
                void (*handler) (int, void , struct pt_regs *),
                unsigned long flags /* SA_INTERRUPT ou SA_SHIRQ */,
                const char *device, void *dev_id);

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

Quand l'interruption qui a pour numéro **irq** se déclenche, la fonction **handler()** est appelée.

Le champ **dev_id** sert à identifier les périphériques en cas de partage d'IRQ (SA_SHIRQ).

La liste des IRQ déjà déclarées est disponible dans **/proc/interrupts**.

Le gestionnaire d'interruptions peut-être déclaré à 2 endroits différents :

- au chargement du pilote (**module_init()**)
- à la première ouverture du pilote par un programme (**open()**). Il faut alors utiliser le compteur d'utilisation et désinstaller le gestionnaire au dernier **close()**.

Le driver peut décider d'activer ou de désactiver le suivi de ses interruptions. Il dispose de 2 fonctions pour cela, définies dans **<linux/irq.h>:**

```
void enable_irq(int irq);
void disable_irq(int irq);
```

L'objectif est d'interdire une interruption d'un programme qui s'exécute. Typiquement, il est primordiale (ou pas) d'interrompre un programme qui est moins prioritaire qu'un autre.

3.4.3) Travail réalisé en TP

a) Bibliothèque et fonction de bas niveau

Dans cette partie, on se base sur les notions vues dans les sections précédentes pour programmer notre propre module noyau nous permettant ainsi de piloter notre LCD et notre capteur de température.

Dans un premier temps on commence par inclure les différentes bibliothèques dont on a parlé précédemment. Et d'autres dont on utilise des fonctions qui seront explicitées plus loin dans ce rapport.

Une fois toutes les bibliothèques ajoutées avec nos #include, on ajoute à notre code les différentes fonctions de contrôle bas niveau programmé au TP1 pour l'initialisation du port, du LCD et du capteur de température, ainsi que les fonctions permettant d'effectuer les différents échanges avec ses périphériques.

```
void tme_tempo(int d);                          //delay function
void PORT_INIT(void);                           //parallel port initialization
void LCD_INIT(void);                            //LCD initialization
void LCD_CMD(unsigned char data);               //write to LCD command register
void LCD_DATA(unsigned char data);              //write to LCD data register
void LCD_STRING(unsigned char *string);         //write string to LCD
void LCD_CLEAR(void);                           //clear LCD and home cursor
void LCD_HOME(void);                            //home cursor on LCD
void LCD_LINE2(void);                           //move cursor to line 2, position 1
void LCD_RS_HIGH(void);
void LCD_RS_LOW(void);
void LCD_E_HIGH(void);
void LCD_E_LOW(void);

void DS1620_CONFIG(unsigned char data);         //DS1620 configuration
void DS1620_WRITECOMMAND(unsigned char command); //Write command to DS1620
void DS1620_WRITEDATA(int data);                //Write data to DS1620
int DS1620_READ(void);                          //Read data from DS1620
void DS1620_RESET_HIGH(void);
void DS1620_RESET_LOW(void);
void DS1620_CLK_HIGH(void);
void DS1620_CLK_LOW(void);
void DS1620_DATA_HIGH(void);
void DS1620_DATA_LOW(void);
int powerof2(int);
```

Il nous faudra aussi inclure le fichier d'en-tête crée qui contient les différents #define des commandes utilisées:

```
#include "lcd.h"
```

b) Structure globale

De plus, il nous faudra déclarer les différentes variables qui nous seront utiles dans notre programme, ainsi que le define avec le nom du module (du périphérique) et la licence pour notre code. Pour plus d'informations sur les fonctions de définition de la licence et des paramètres merci de bien vouloir consulter la section précédente sur la structure d'un module noyau.

```
#define DEVICE_NAME "lcd"

MODULE_LICENSE("Dual BSD/GPL");

// adresse unique du Majeur de notre module
static int major = 124; /* dynamic by default */
module_param(major, int, 0);

// buffer utiliser dans la fonction de lecture
int buffer_read ;

// adresse du port de base du périphérique
static unsigned long port_base_addr = 0x378;
module_param(port_base_addr, long, 0);

// buffer kernel
unsigned char kernel_buf[100];
```

c) Initialisation et suppression

Nous allons maintenant passer à l'implémentation des fonctions d'initialisation et de suppression du module permettant l'ajout et la suppression du module noyau via les commandes **insmod** et **rmmod** (ces fonctions sont celles appellées lors de l'utilisation d'insmod et rmmod).

Le nom donné à nos fonctions est respectivement **lcd_init** pour la fonction d'initialisation appelée avec la commande insmod et **lcd_exit** pour la fonction de suppression du module noyau avec la commande rmmod.

```
module_init(lcd_init);
module_exit(lcd_exit);
```

La fonction d'initialisation appelée via la commande **insmod**, ajoute le module au noyau à l'aide de la fonction **register_chrdev()** (voir section précédente pour comprendre comment fonctionne cette fonction) avec le numéro de major **124** (major est un entier déclaré comme paramètre), le nom « lcd » et la structure **lcd_fops**. Ce dernier contient les pointeurs vers les différentes fonctions à utiliser lors des différents appels système, affiche des messages kernel (consultable avec **dmesg**) et initialise le port parallèle avec la fonction PORT_INIT() :

```
static int lcd_init(void)
{
        int err = 0;
        shortp_base = base;

        if (err = register_chrdev(major,DEVICE_NAME,&lcd_fops))
                printk("*** unable to get major LCD_MAJOR for Lcd devices ***\n");

        PORT_INIT();
        printk(KERN_ALERT "lcd_couscous:lcd_init\n");
        printk(KERN_ALERT "lcd_couscous:basePortAddress is 0x%lx\n", shortp_base);

        return err;
}
```

Quant à la fonction **lcd_exit(),** son ordre est l'inverse de celui de **lcd_init()**. Elle est appelée avec la commande **rmmod** et supprime le module noyau avec la fonction **unregister_chredev()** qui prend en paramètre le nom du module ainsi que son numéro de major.

```
static void lcd_exit(void)
{
         unregister_chrdev(major,DEVICE_NAME);

         printk(KERN_ALERT "lcd_couscous:lcd_exit\n");
}
```

la structure permettant de lier les différentes fonctions que nous allons voir avec les différends appelle système est la suivante :

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

d) L'appel système Open()

La fonction **open()** est appelée à chaque utilisation du fichier. Associé au périphérique, elle initialise le périphérique à l'aide des fonctions vues précédemment (TP1) et vérifie qu'il s'agit bien du bon périphérique auquel on essaye d'accéder via le test de son numéro de **MINEUR**. Cependant, on ne branche qu'une seule carte donc cette vérification ne sert pas à grand-chose dans notre situation mais peut être utile si on avait plusieurs périphériques semblables de connectés.

```
int lcd_open(struct inode *inode, struct file *filp)
{
        printk(KERN_INFO "lcd_couscous: release\n");

        LCD_INIT();
        LCD_HOME();
        LCD_CLEAR();

        if(MINOR(inode->i_rdev) != 0) {
                printk("mauvais numero d unite\n");
                return -EINVAL;
        }
        return 0;
}
```

e) L'appel système Close()

La fonction **release()** correspond à l'appel système close (fermeture du fichier). Celle-ci est appelée une fois l'accès sur le fichier terminé. Ici cette fonction ne fait pas grand-chose hormis afficher un message de debug dans les messages kernel. Cependant, dans d'autres cas, si par exemple on avait alloué de la mémoire ou autre, c'est ici qu'il faudrait libérer cette mémoire (pour plus de détails sur l'allocation de mémoire dynamique dans le kernel voir section précédente).

```
int lcd_release(struct inode *inode, struct file *filp)
{
      printk(KERN_INFO "lcd_couscous: release\n");

      return 0;
}
```

f) Fonction d’échange de données User/Kernel

Avant d'aborder les appels système Read et Write, il est primordial de s’intéresser aux différentes manières de communiquer des données de l'espace utilisateur (user) vers l'espace noyau (kernel). Ces échanges sont nécessaires pour pouvoir récupérer des informations d'un espace vers un autre. L’accès à la mémoire kernel étant critique, ces échanges ne peuvent se faire sans passer par des vérifications intermédiaires. Pour notre driver nous utilisons deux fonctions permettant la copie de données respectivement de l’espace utilisateur vers l'espace kernel, et de l'espace kernel vers l'espace utilisateur. Elles se trouvent dans la bibliothèque «asm/uaccess.h».

Les prototypes ainsi que les explications relatives à ces fonctions sont décrites ci-dessous:

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

La fonction permet la copie de données depuis l'espace utilisateur vers l'espace noyau, la fonction retourne le nombre d'octets qui n'ont pas pu être copié. Dans le cas d'une copie réussie la fonction renvoie zéro.

To : adresse de destination dans l'espace noyau.
From : adresse source dans l'espace utilisateur.
N : nombre d'octets à copier.

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

La fonction permet la copie de données depuis l'espace noyau vers l'espace utilisateur, la fonction retourne le nombre d'octets qui n'ont pas pu être copié, dans le cas d'une copie réussie la fonction renvoie zéro.

To : adresse de destination dans l'espace utilisateur.
From : adresse source dans l'espace noyau.
N : nombre d'octets à copier.

g) L'appel système Read()

Le but étant, via l'appel système read(), de pouvoir lire la température ambiante à travers le thermomètre de notre périphérique et de l'afficher sur la console (pour plus d'informations sur la structure et le fonctionnement de l'appel système Read() voir la section précédente). De plus le passage des données du noyau vers l'utilisateur se fait via la fonction copy_to_user() (voir section précédente). Il est à noter qu'ici, pour éviter une attente infinie à la lecture avec la console via un appel du type (/dev/lcd), nous utilisons une variable intermédiaire qui sert à mettre fin à la lecture en renvoyant un nombre lu de 6.

Il est à noter que l'appel à la fonction cat provoque une attente de 4048 octets.

Le contrôle de la lecture est simple et repose principalement sur le fonctionnement de l'appel read(). Aux premiers appels de read() la fonction lit la température et la passe à l'espace utilisateur pour que celle-ci soit affichée. Une fois cette opération effectuée, la valeur de la variable « je suis un lapin » est mise à « 1 », ce qui implique qu'au prochain appel provoqué par CAT (demande des 4048 octets), la fonction permettra de terminer cet appel indiquant à la fonction CAT que le transfert est terminé et qu'il n'y a plus rien à lire.

```
ssize_t lcd_read(struct file *filp, char __user *user_buf, size_t count, loff_t *f_pos)
{
	int temperature = 0;

	if(suis_je_un_lapin == 0)
	{
		DS1620_RESET_HIGH();
		DS1620_WRITECOMMAND(READTEMP);
		temperature = DS1620_READ();
		DS1620_RESET_LOW();
		printk(KERN_INFO "lcd_couscous: read |%d|\n",temperature/2);

		tme_tempo(100);

		sprintf(kernel_buf,"%d", temperature/2);
		copy_to_user(user_buf, kernel_buf, count);
		suis_je_un_lapin = 1;
		count = 6;
		return count;
	}
	else
	{
		suis_je_un_lapin = 0;
		count = 0;
	}
	return 0;
}
```

h) L'appel système Write()

L'appel système write() dans notre driver nous permet d'écrire une chaîne de caractères de 16 caractères sur l'écran LCD. La fonction fait donc passer des données depuis l'espace utilisateur vers l'espace noyau et pour cela nous utilisons la fonction copy_from_user() décrite précédemment. De plus, un contrôle du nombre max de caractères à écrire est faite par la fonction. Celui-ci force le nombre de caractères maximal à écrire à 16 si la taille de la chaîne passée en tant que paramètre depuis le mode utilisateur est supérieure à 16 caractères.

```
ssize_t lcd_write(struct file *filp, const char __user *user_buf, size_t count, loff_t *f_pos)
{
        if (count > 16)
                count = 16;

        copy_from_user(kernel_buf,user_buf,count);

        kernel_buf[count-1]='\0';
        LCD_STRING(kernel_buf);

        printk(KERN_INFO "lcd_couscous: write %s \n",kernel_buf);

      return count;
}
```

4) Réseau Xbee et base de données

4.1) Présentation

Dans cette partie nous allons voir qu'est-ce qu'un Xbee et parler du protocole Zigbee. Nous donnerons certaines spécificités et caractéristiques lié au Xbee, puis nous allons illustrer les possibilités du composant dans un système complet. Enfin, nous finirons par certaines explications sur le code vu en TP.

4.1.1) Présentation du Xbee

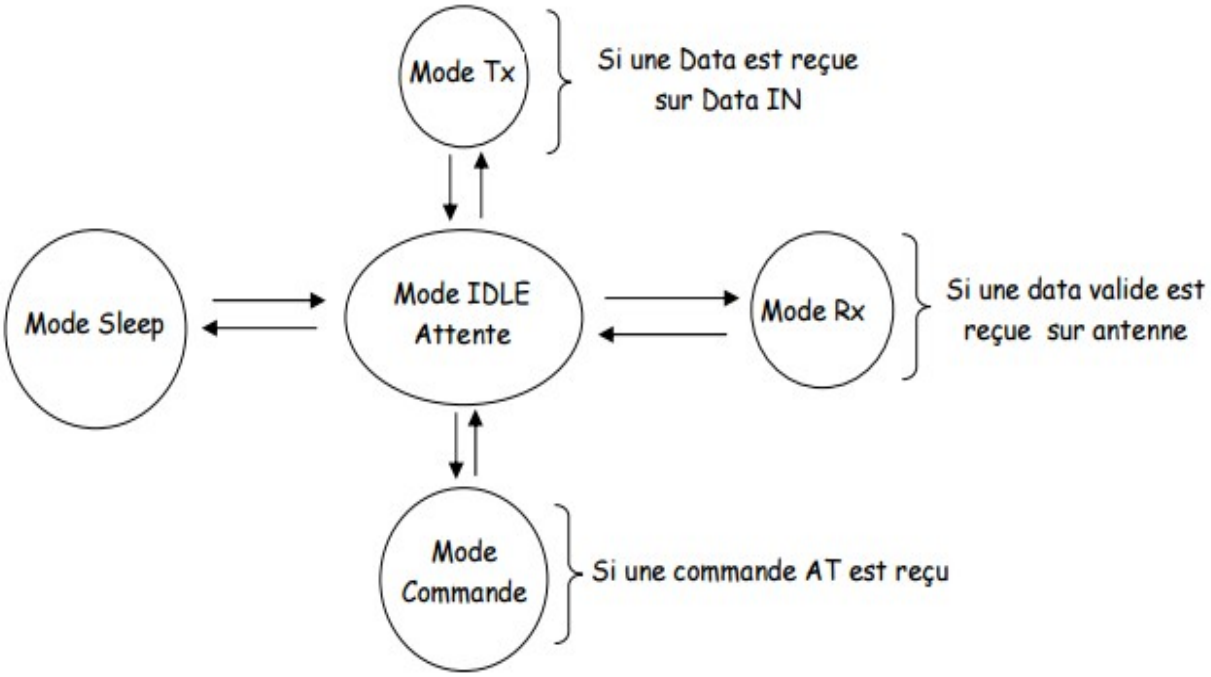
Les produits MaxStream XBeeTM sont des modules de communication sans fil très populaires fabriqués par l'entreprise Digi International. Ils ont été certifiés par la communauté industrielle ZigBee Alliance en 2006 après le rachat de MaxStream par Digi International. La certification Zigbee se base sur le standard IEEE 802.15.4 qui définit les fonctionnalités et spécifications des réseaux sans fil à dimension personnelle (Wireless Personal Area Networks : WPANs). Nous verrons plus loin chacun des termes qui peuvent poser problème.

4.1.2) Les principales caractéristiques du XBee

– fréquence porteuse : 2.4Ghz
– portées variables : assez faible pour les XBee 1 et 2 (10 - 100m), grande pour le XBee Pro (1000m)
– faible débit : 250kbps
– faible consommation : 3.3V @ 50mA
– entrées/sorties : 6 10-bit ADC input pins, 8 digital IO pins
– sécurité : communication fiable avec une clé de chiffrement de 128-bits
– faible coût : ~25€
– simplicité d'utilisation : communication via le port série
– ensemble de commandes AT et API
– flexibilité du réseau : sa capacité à faire face à un nœud hors service ou à intégrer de nouveaux nœuds rapidement
– grand nombre de nœuds dans le réseau : 65000
– topologies de réseaux variées : maillé, point à point, point à multipoint

Les modules XBees fonctionnent dans 12 canaux de la bande 2,4 GHz. La puissance d’émission est ajustable entre 10 mW et 60 mW. La portée théorique à l'intérieur est de 100 m et de 1500 m en extérieur. Ils doivent être alimentés entre 2,8 et 3,4 V. La consommation en réception est 50 mA. Elle passe à 210 mA en émission 60 mW. En mode "sleep" la consommation est inférieure à 10 µA. Le protocole utilisé est le 802.15.4 de la norme ZigBee.

Les modes du Xbee possibles:



4.1.3) Entre Xbee et Zigbee

Bee signifiant "abeille", le choix du nom donne l'image qu'il peut y avoir plusieurs petits modules connectés ensemble, comme une colonie d'abeilles. Au début, on peut confondre les termes XBee et ZigBee.

En fait, comme expliqué au début, le ZigBee est un protocole de communication qui s'appuie sur le travail du groupe IEEE 802.15.4 et définit par le groupe de professionnels ZigBee Alliance. Le XBee est une marque, un produit qui utilise le protocole ZigBee.

Comparaison des protocoles Zigbee, Bluetooth et Wi-Fi:

Caractéristique	Zigbee	Bluetooth	Wi-Fi
IEEE	802.15.4	802.15.1	802.11a/b/g/n/ac
Besoins mémoire	4-32 ko	250 ko +	1 Mo +
Autonomie avec pile	Années	Mois	Jours
Nombre de nœuds	65 000+	255	256+
Vitesse de transfert	20-250 kb/s	1 Mb/s	11-54-108-320-1000 Mb/s
Portée (environ)	100 m	10 m	300 m

4.1.4) Séries 1 et 2

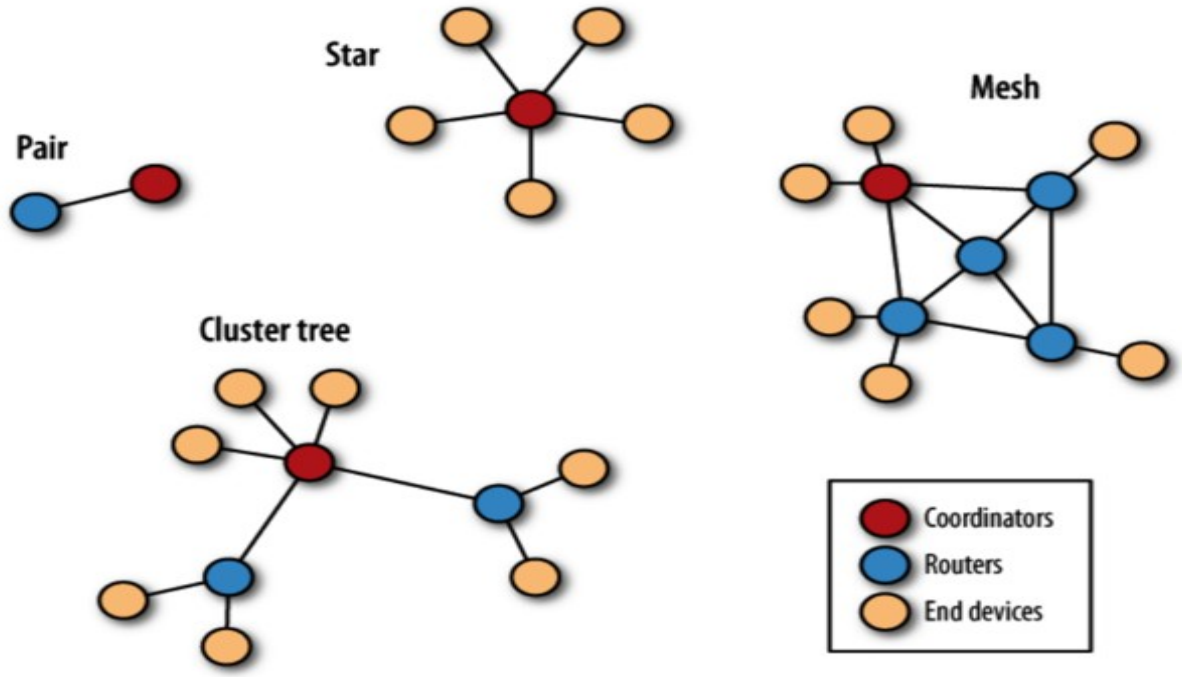
Plusieurs produits XBee existent, ce qui peut créer quelques confusions. Il faut retenir qu'il y a deux catégories de XBee : la série 1 et la série 2. Les modules de la série 1 ont souvent un "802.15.4" qui s'ajoutent à leurs noms.

Les modules de la série 2 sont disponibles en plusieurs versions : XBee ZNet 2.5 (obsolète), le ZB (l'actuel) et le 2B (le plus récent). Vous avez aussi des XBee Pro, qui font la même chose, mais avec de plus grandes capacités, notamment la portée qui semble pouvoir aller jusqu'à 1000 mètres! Pour en savoir plus, télécharger le tableau de comparaison des modules Xbee: http://www.digi.com/pdf/chart_xbee_rf_features.pdf.

	Series 1	Series 2
Typical (indoor/urban) range	30 meters	40 meters
Best (line of sight) range	100 meters	120 meters
Transmit/Receive current	45/50 mA	40/40 mA
Firmware (typical)	802.15.4 point-to-point	ZB ZigBee mesh
Digital input/output pins	8 (plus 1 input-only)	11
Analog input pins	7	4
Analog (PWM) output pins	2	None
Low power, low bandwidth, low cost, addressable, standardized, small, popular	Yes	Yes
Interoperable mesh routing, ad hoc network creation, self-healing networks	No	Yes
Point-to-point, star topologies	Yes	Yes
Mesh, cluster tree topologies	No	Yes
Single firmware for all modes	Yes	No
Requires coordinator node	No	Yes
Point-to-point configuration	Simple	More involved
Standards-based networking	Yes	Yes
Standards-based applications	No	Yes
Underlying chipset	Freescale	Ember
Firmware available	802.15.4 (IEEE standard), DigiMesh (proprietary)	ZB (ZigBee 2007), ZNet 2.5 (obsolete)
Up-to-date and actively supported	Yes	Yes

Ce qu'il faut retenir :

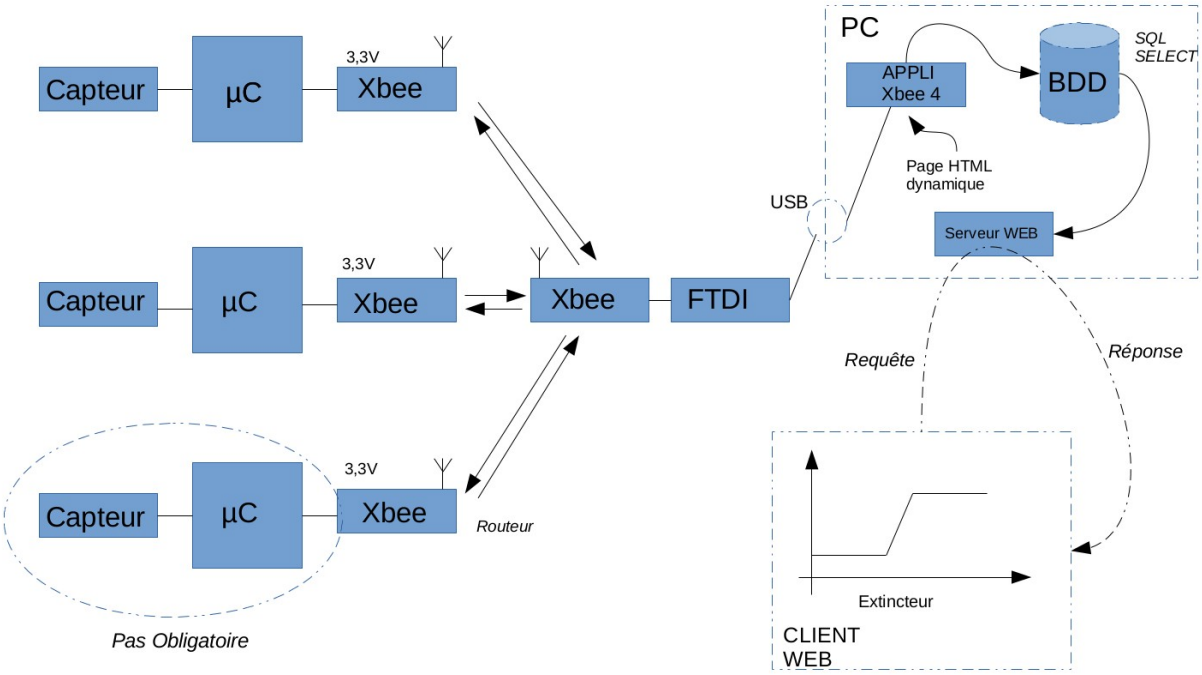
– les modules des séries 1 et 2 ne sont pas compatibles entre eux;
– la portée et la consommation sont sensiblement les mêmes;
– le nombre d'entrées et sorties est différent et surtout la série 2 ne possède pas de sorties analogiques PWM;
– les topologies de réseaux possibles ne sont pas les mêmes. Avec la série 1, l'architecture est simple : point à point (pair) ou multipoint (star). La série 2 permet en plus de créer des réseaux plus complexes: maillés (mesh) ou en "arbre" (cluster tree).



4.2) Illustration et utilisation

Dans cette partie nous allons présenter un système reposant sur l'internet des objets qui se compose de Xbee communiquant avec une base de données sous apache (ou autre). Par l'intermédiaire d'un ordinateur sous une architecture client serveur, il permet l'interaction entre le système et une page web.

4.2.1) Présentation et Schéma complet du système



Le système présenté dans l'image ci-dessus permet à un utilisateur beta, ayant préalablement équipé sa maison de capteurs de température reliés à des émetteurs récepteurs Xbee à un ordinateur avec une base de données sur serveur, de pouvoir à partir de nimporte quel ordinateur de se connecter au système, lire des données telle que la température (si capteur de température) ou autre et d'interagir avec les différents objets via des contacteurs ou autres.

4.2.2) Présentation et principe d'une base de donnée (client-serveur)

a) Base de données

Une base de données (en anglais: *database*) est un outil permettant de stocker et de retrouver l'intégralité de données brutes ou d'informations en rapport avec un thème ou une activité. Celles-ci peuvent être de natures différentes et plus ou moins reliées entre elles. Dans la très grande majorité des cas, ces informations sont très structurées, et la base est localisée dans un même lieu et sur un même support. Ce dernier est généralement informatisé.

La base de données est au centre des dispositifs informatiques de collecte, mise en forme, stockage, et utilisation d'informations. Le dispositif comporte un système de gestion de base de données (abr. SGBD): un logiciel moteur qui manipule la base de données et dirige l'accès à son contenu. De tels dispositifs —souvent appelés base de données— comportent également des logiciels applicatifs, et un ensemble de règles relatives à l'accès et l'utilisation des informations.

La manipulation de données est une des utilisations les plus courantes des ordinateurs. Les bases de données sont par exemple utilisées dans les secteurs de la finance, des assurances, des écoles, de l'épidémiologie, de l'administration publique (statistiques notamment) et des médias.

Lorsque plusieurs choses appelées bases de données sont constituées sous forme de collection, on parle alors d'une banque de données (en anglais: *data bank*).

b) Client-serveur

L'environnement client-serveur désigne un mode de communication à travers un réseau entre plusieurs programmes ou logiciels: l'un, qualifié de client, envoie des requêtes. L'autre ou les autres, qualifiés de serveurs, attendent les requêtes des clients et y répondent. Par extension, le client désigne également l'ordinateur sur lequel est exécuté le logiciel client, et le serveur, l'ordinateur sur lequel est exécuté le logiciel serveur.

En général, les serveurs sont des ordinateurs dédiés au logiciel serveur qu'ils abritent, et dotés de capacités supérieures à celles des ordinateurs personnels en termes de puissance de calcul, d'entrées-sorties et de connexions réseau. Les clients sont souvent des ordinateurs personnels ou des appareils individuels (téléphone, tablette), mais pas systématiquement. Un serveur peut répondre aux requêtes d'un grand nombre de clients.

Il existe une grande variété de logiciels serveurs et de logiciels clients en fonction des besoins à servir: un serveur web publie des pages web demandées par des navigateurs web; un serveur de messagerie électronique envoie des mails à des clients de messagerie; un serveur de fichiers permet de stocker et consulter des fichiers sur le réseau; un serveur de

données à communiquer des données stockées dans une base de données, etc.

Caractéristiques d'un processus serveur:

- il attend une connexion entrante sur un ou plusieurs ports réseaux;
- à la connexion d'un client sur le port en écoute, il ouvre un socket local au système d'exploitation;
- suite à la connexion, le processus serveur communique avec le client suivant le protocole prévu par la couche application du modèle OSI.

Caractéristiques d'un processus client:

- il établit la connexion au serveur à destination d'un ou plusieurs ports réseaux;
- lorsque la connexion est acceptée par le serveur, il communique comme le prévoit la couche applicative du modèle OSI.

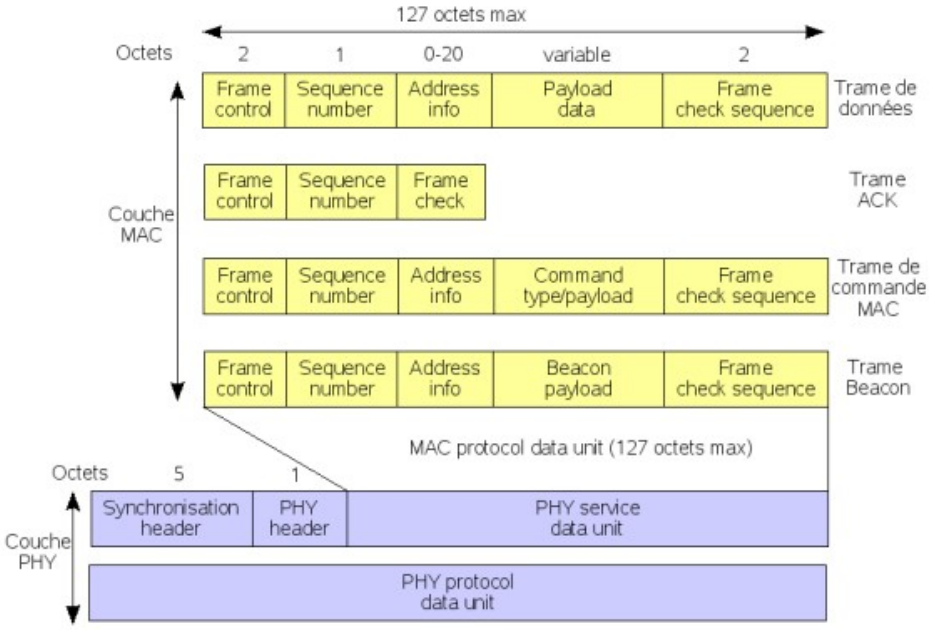
Le client et le serveur doivent bien sûr utiliser le même protocole de communication au niveau de la couche transport du modèle OSI. Un serveur est généralement capable de servir plusieurs clients simultanément. On parle souvent d'un service pour désigner la fonctionnalité offerte par un processus serveur. On définit aussi comme serveur, un ordinateur spécialisé ou une machine virtuelle ayant pour unique tâche l'exécution d'un ou plusieurs processus serveur.

4.3) Analyse du code vu en TP

Dans cette partie nous allons entrevoir certains points permettant la programmation des Xbee.

4.3.1) Construction d'une trame Xbee (xbee trame Maker)

Les trames Xbee répondent à un protocole bien définis qui est le Zigbee. Cette section se propose d'expliquer brièvement la structure de cette trame:



La structure de la trame MAC est flexible pour s'adapter aux différentes applications

- MPDU = MHR + MSDU + MFR (MAC Protocol Data Unit = MAC HeadeR + MAC Service Data Unit + MAC FooteR)
- MHR (MAC HeadeR):
  - Le champ Frame Control indique type de trame MAC et spécifie le format du champ adresse
  - Le champ Sequence Number assure l'ordre à la réception et permet l'acquittement des trames MAC
  - Le champ adresse est variable de 0 à 20 octets en fonction du type trame :
    - data : adresse source et adresse destination
    - aquittement : pas d'adresse
    - etc.
- MSDU (MAC Service Data Unit):
  - La trame de données permet une charge utile jusqu'à 104 octets.
- MFR (MAC FooteR) :
  - Le FCS (Frame Check Sequence) assure que la trame est transmise sans erreur.
  - CRC sur 16 bits

- Les types de trame MAC :
    - La trame de données
    - La trame ACK permet de garantir à l'expéditeur que sa trame a été reçue sans erreur. La trame ACK est émise juste après la réception pendant le "Quiet Time" d'entre deux trames.
    - La trame de commande MAC permet le contrôle et la configuration à distance des noeuds par le coordinateur PAN.
    - Les trames de balisage (Beacon) réveillent les modules clients qui attendent leur adresse et se rendorment s'ils ne la reçoivent pas. Les trames beacon sont importantes dans les réseaux maillés et les clusters d'étoile pour que les noeuds soient synchronisés avec une consommation d'énergie minimum.

Un outil bien pratique pour la construction de ses trame est DIGI API frame maker, disponible sur internet a l'adresse suivante:

http://ftp1.digi.com/support/utilities/digi_apiframes2.htm

4.3.2) La fonction select

Les fonctions **select()** et **pselect()** permettent à un programme de surveiller plusieurs descripteurs de fichier, attendant qu'au moins l'un des descripteurs de fichier devienne «prêt» pour certaines classes d'opérations d'entrées-sorties (par exemple, entrée possible). Un descripteur de fichier est considéré comme prêt s'il est possible d'effectuer l'opération d'entrées-sorties correspondante (par exemple, un read) sans bloquer.

select() et **pselect()** ont un comportement identique, avec trois différences:

(i)

La fonction **select()** utilise un délai exprimé avec une ***struct timeval*** (secondes et microsecondes), alors que **pselect()** utilise une ***struct timespec*** (secondes et nanosecondes).

(ii)

La fonction **select()** peut modifier le paramètre ***timeout*** pour indiquer le temps restant. La fonction **pselect()** ne change pas ce paramètre.

(iii)

La fonction **select()** n'a pas de paramètre ***sigmask*** et se comporte comme **pselect()** avec une valeur NULL pour ***sigmask***

Il y a trois ensembles indépendants de descripteurs surveillés simultanément. Ceux de l'ensemble ***readfds*** seront surveillés pour vérifier si des caractères deviennent disponibles en lecture. Plus précisément, on vérifie si un appel système de lecture ne bloquera pas; en particulier un descripteur en fin de fichier sera considéré comme prêt. Les descripteurs de l'ensemble ***writefds*** seront surveillés pour vérifier si une écriture ne bloquera pas. Ceux de ***exceptfds*** seront surveillés pour l'occurrence de conditions exceptionnelles. En sortie, les ensembles sont modifiés pour indiquer les descripteurs de fichier qui ont changé d'état. Chacun des trois ensembles de descripteurs de fichier peut être spécifié comme **NULL** si aucun descripteur de fichier n'est surveillé pour la classe correspondante d'événements.

Quatre macros sont disponibles pour la manipulation des ensembles FD_ZERO() efface un ensemble. **FD_SET()** et **FD_CLR()** ajoutent et suppriment, respectivement, un descripteur de fichier dans un ensemble. **FD_ISSET()** vérifie si un descripteur de fichier est contenu dans un ensemble, principalement utile après le retour de **select()**.

nfds est le numéro du plus grand descripteur de fichier des 3 ensembles, plus 1.

timeout est une limite supérieure au temps passé dans **select()** avant son retour. Elle peut être nulle, ce qui conduit select() à revenir immédiatement. (Ce qui sert pour des surveillances en ***polling***). Si le timeout est **NULL** (aucun), **select()** peut bloquer indéfiniment.

sigmask est un pointeur sur un masque de signaux . S'il n'est pas NULL, alors **pselect()** remplace d'abord le masque de signaux en cours par celui indiqué dans ***sigmask***, puis invoque la fonction **«select»**, et enfin restaure le masque de signaux à nouveau.

Outre la différence de précision de l'argument ***timeout***, l'appel **pselect()** suivant

ready = pselect(nfds, &readfds, &writefds, &exceptfds,
timeout, &sigmask);

est équivalent à l'exécution *atomique* des appels suivants:

sigset_t origmask;

sigprocmask(SIG_SETMASK, &sigmask, &origmask);
ready = select(nfds, &readfds, &writefds, &exceptfds, timeout);
sigprocmask(SIG_SETMASK, &origmask, NULL);

La raison pour laquelle **pselect()** est nécessaire est que si l'on veut attendre soit un signal, soit qu'un descripteur de fichier soit prêt, alors un test atomique est nécessaire pour éviter les situations de concurrence. (Supposons que le gestionnaire de signaux active un drapeau global et revienne. Alors un test de ce drapeau, suivi d'un appel **select()** peut bloquer indéfiniment si le signal arrive juste après le test mais avant l'appel. À l'inverse, **pselect()** permet de bloquer le signal d'abord, traiter les signaux déjà reçus, puis invoquer **pselect()** avec le ***sigmask***, désiré, en évitant la situation de blocage.)

VALEUR RENVOYÉE

En cas de réussite **select**() et **pselect**() renvoient le nombre de descripteurs de fichier dans les trois ensembles de descripteurs retournés (c'est-à-dire, le nombre total de bits à 1 dans *readfds*, *writefds*, *exceptfds*) qui peut être nul si le délai de timeout a expiré avant que quoi que ce soit d'intéressant ne se produise. Ils retournent -1 s'ils échouent, auquel cas *errno* contient le code d'erreur; les ensembles et *timeout* ne sont plus définis, ne vous fiez plus à leur contenu après une erreur.

ERREURS

EBADF:
Un descripteur de fichier (dans l'un des ensembles) est invalide. (Peut-être un descripteur de fichier qui était déjà fermé ou sur lequel une erreur s'est produite.

EINTR:
Un signal a été intercepté; voir **signal**(7).

EINVAL:
nfds est négatif ou la valeur contenue dans *timeout* n'est pas valide.

ENOMEM:
Pas assez de mémoire pour le noyau.

«Conclusion»

À travers les différentes notions abordées lors de ce module et à travers les TP, on peut voir que l'écriture de module noyau sous Linux (du moins les modules de type caractères) n'ont rien de compliqué car il se base sur une structure standard. En effet une fois ces bases et notions assimilées, le seul travail intellectuel restant étant la compréhension du fonctionnement du périphérique voulu et cela à travers sa documentation. Une fois les différentes fonctions d'initialisation et d'utilisation du périphérique codées et testées en mode super utilisateur, il suffira alors de les intégrer à notre structure et d'adapter ce dernier comme bon nous semble.

«Annexe»

http://pficheux.free.fr/articles/lmf/drivers/

http://www.linuxpedia.fr/doku.php/commande/modules

http://www.linux-france.org/prj/support/outils/drivers.html

http://www.developpez.net/forums/blogs/?blog=58&title=le_kit_de_survi_du_developpeur_linux&more=1&c=1&tb=1&pb=1

http://doc.ubuntu-fr.org/tutoriel/tout_savoir_sur_les_modules_linux

http://broux.developpez.com/articles/c/driver-c-linux/

http://lea-linux.org/documentations/Kernel-modules

http://lwn.net/Kernel/LDD3/

http://fr.wikipedia.org/wiki/Client-serveur

http://fr.wikipedia.org/wiki/Base_de_donn%C3%A9es

http://wapiti.telecom-lille1.eu/commun/ens/peda/options/ST/RIO/pub/exposes/exposesrio2006/Descharles-Waia/network.htm

http://manpagesfr.free.fr/man/man2/select.2.html