Communiquer en i²c avec un capteur de température*

Christophe BLAESS

Le protocole i²c est le plus épuré des bus de communication entre un processeur et ses périphériques. Très simple à employer par le hobbyiste, nous allons le mettre en œuvre pour lire et programmer un capteur de température externe. Ce dernier ne nous servira qu'à titre d'exemple, tout périphérique i²c pouvant convenir.

Bus i²c

On présente généralement **i²c** (*inter integrated circuit*) comme le plus simple des bus de communication utilisés dans l'électronique moderne. Il s'appuie simplement sur deux signaux appelés SDA (*Serial Data*) et SCL (*Serial Clock*), sans oublier la masse commune entre les équipements. Il s'agit d'une communication bidirectionnelle *half-duplex* – où chacun ne parle qu'à son tour – reposant sur une communication série synchrone.

Le protocole permet de mettre en communication un composant **maître** (généralement le microprocesseur) et plusieurs périphériques **esclaves**. Plusieurs maîtres peuvent partager le même bus, et un même composant peut passer du statut d'esclave à celui de maître ou inversement. Toutefois la communication n'a lieu qu'entre un seul maître et un seul esclave. Notons également que le maître peut également envoyer un ordre à tous ses esclaves simultanément (par exemple une mise en sommeil ou une demande de réinitialisation).

Signaux électriques

Au niveau électrique, le protocole utilise des signaux alternant entre des niveaux bas et hauts, le plus fréquemment il s'agit de {0, 5V} ou de {0, 3.3V}. Le signal d'horloge **SCL** est produit par le maître. Le signal de données **SDA** est mis au niveau haut ou bas par le maître ou l'esclave suivant la phase de communication.

Pendant toute la durée du créneau haut du signal d'horloge SCL, le signal de données SDA doit être maintenu au niveau haut ou bas suivant que l'on transmet un 1 ou un 0.

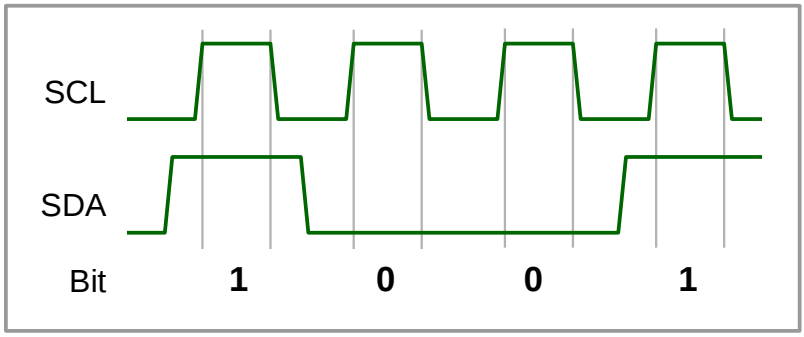


Fig. 1 : Exemple de signaux i²c

Enfin des configurations particulières des signaux (produits par le maître) permettent d'indiquer un début et une fin d'échange, en réalisant ce que l'on nomme les conditions START et STOP. Il s'agit d'une variation de signal SDA pendant un créneau de l'horloge.

* Cet article est paru dans Gnu/Linux Magazine Hors Série numéro 75 « *Raspberry Pi Avancé* » en novembre 2014, disponible sur `https://boutique.ed-diamond.com/anciens-numeros/789-gnulinux-magazine-hs-75.html`

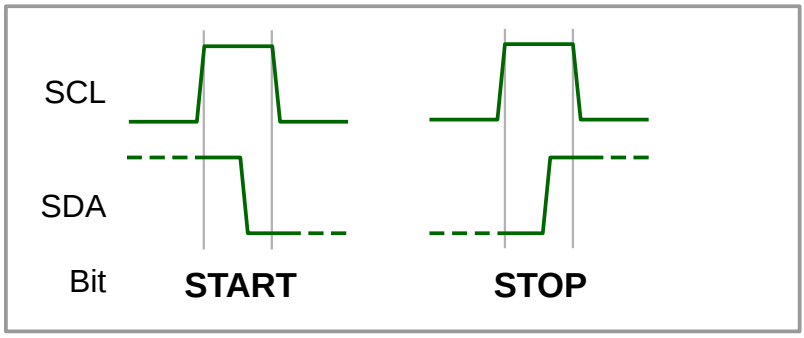


Fig. 2 : Condition START

Protocole de communication

La communication s'établit toujours à l'initiative du maître. Celui-ci présente une condition START sur la ligne SDA, suivie de l'**adresse** (sur sept bits) de l'esclave avec lequel il souhaite communiquer, puis un bit indiquant le sens (0 = écriture ou 1 = lecture) de la communication.

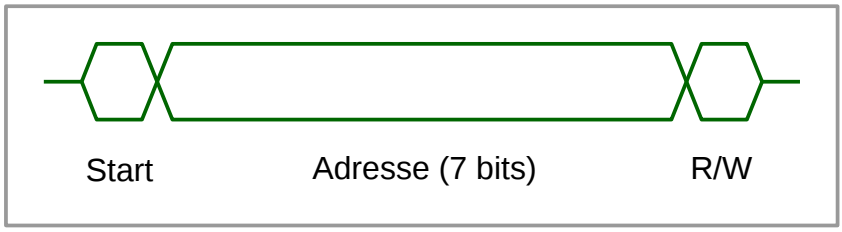


Fig. 3 : Initialisation de la communication

Le fait que les adresses soient sur sept bits, mais toujours suivies d'un bit indiquant le sens de transmission conduit parfois à considérer que chaque périphérique est doté de deux adresses unidirectionnelles sur huit bits : l'adresse paire est destinée à l'écriture vers le périphérique et l'adresse impaire à la lecture.

Il existe quelques adresses particulières, notamment **`0x00`** qui envoie un message à destination de tous les esclaves. Plus généralement, les adresses inférieures à **`0x08`** et supérieures à **`0x77`** ont des rôles spécifiques.

Lorsqu'un périphérique reconnaît son adresse, il doit écrire sur la ligne SDA un bit 0, indiquant ainsi un acquittement (ACK).

Si le message est une écriture vers le périphérique, le maître peut alors envoyer sa commande, octet par octet, l'esclave acquittant chacun d'entre eux par un bit ACK.

Il existe naturellement des subtilités dans ce protocole, pour gérer les erreurs de transmission, les répétitions de messages, les changements de statut esclave / maître et les arbitrages de bus entre plusieurs maîtres.

Pour en savoir plus, je recommande l'article « **i²c** » de Wikipedia.

SMBus

La plupart des documentations font référence au protocole i²c, mais l'implémentation que l'on rencontre sur les systèmes monocartes sous Linux (Raspberry Pi, BeagleBone Black, etc.) est plutôt une spécialisation de ce protocole nommé **SMBus** (*System Management Bus*). Les différences essentielles sont des limitations dans les tensions autorisées (3.3V maxi.) ou les fréquences de communication (10 à 100 kHz seulement).

Le protocole SMBus est plus particulièrement utilisé pour communiquer entre un microprocesseur et les périphériques annexes sur la carte-même (ventilateur, capteur de température, etc.)

Ceci explique que de nombreuses fonctions de l'API Linux pour le sous-système i²c soient préfixées par **`smbus`**.

Linux, Raspberry Pi et i²c.

Le protocole i²c est supporté par le noyau Linux depuis sa version 2.4. De nombreux périphériques sont reconnus par le kernel, notamment dans le sous-système **Hwmon** (*Hardware Monitor*).

L'accès depuis l'espace utilisateur est facilité par le module **`i2c-dev`** qui rend les bus i²c visibles dans le répertoire **`/dev`** sous forme de fichier spéciaux représentant des périphériques en mode caractère.

Démarrons notre Raspberry Pi avec une distribution classique – j'utilise ici une Raspbian. Initialement, le noyau Linux ne détecte aucun contrôleur i²c, c'est normal.

```
pi@raspberrypi:~$ ls /sys/class/i2c-adapter/
pi@raspberrypi:~$
```

Il nous faut charger dans le noyau le module qui gère le contrôleur i²c inclus dans le *system-on-chip* (de la famille Broadcom 2708) du Raspberry Pi, . Ce module est nommé **`i2c_bcm2708`**. On peut le charger manuellement ainsi :

```
pi@raspberrypi:~$ sudo -i
root@raspberrypi:~# modprobe i2c_bcm2708
```

Pour éviter que ce driver ne soit chargé systématiquement au démarrage, le module **`i2c_bcm2708`** est inscrit dans la *blacklist* de **`modprobe`**. Comme peu d'utilisateurs s'en servent, cela évite de consommer inutilement des ressources. Pour le retirer de la *blacklist*, il suffit d'éditer le fichier **`/etc/modprobe.d/raspi-blacklist.conf`** et de mettre en commentaire la ligne

```
blacklist i2c-bcm2708
```

en la précédant par un caractère dièse '**#**'. Ainsi au prochain démarrage le module sera automatiquement chargé.

```
root@raspberrypi:~# ls /sys/class/i2c-adapter/
i2c-0  i2c-1
```

Le Raspberry Pi dispose de deux interfaces i²c. Le bus numéro 0 était accessible à travers le connecteur P5 qui était apparu dans la seconde version du Raspberry Pi modèle B (voir http://elinux.org/Rpi_Low-level_peripherals#P5_header) et qui a disparu depuis le modèle B+. De plus ce connecteur étant présent sous forme de simples trous cuivrés, il était donc nécessaire de venir y souder des broches pour pouvoir l'utiliser. Les signaux du bus 0 sont toujours présents – mais pas très accessibles – sur les connecteurs J3 (*camera*) et J4 (*display*).

Nous allons plutôt nous intéresser au second bus, accessible via le port d'extension P1, sur deux broches (que l'on peut également employer pour des entrées / sorties par GPIO) identiques quel que soit le modèle de Raspberry. Il s'agit de la broche 3 (signal SDA) et de la broche 5 (signal SCL).

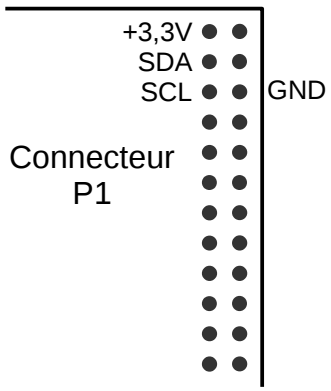


Fig. 4 : Connecteur P1 du Raspberry Pi

Le bus i²c-0 est vide, mais le bus i²c-1 contient déjà deux périphériques :

```
root@raspberrypi:~# ls /sys/class/i2c-adapter/i2c-0/
delete_device  device  name  new_device  power  subsystem  uevent
root@raspberrypi:~# ls /sys/class/i2c-adapter/i2c-1/
1-003b  delete_device  name        power     uevent
1-004c  device         new_device  subsystem
```

```
root@raspberrypi:~# cat /sys/class/i2c-adapter/i2c-1/1-003b/name
wm8804
root@raspberrypi:~# cat /sys/class/i2c-adapter/i2c-1/1-004c/name
pcm5122
```

Les deux périphériques intégrés dans le *system-on-chip* sont un Wolfson Microelectronics WM8804 (tranceiver audio) et un Texas Instrument PCM 5122 (Convertisseur analogique/numérique audio).

Ajout d'un périphérique i²c

Nous allons établir une communication avec un capteur de température **DS 1621** de Maxim Integrated. Ce composant peu coûteux présente l'avantage d'être largement disponible en boîtier DIP (*Dual Inline Package*) aisément utilisable sur la platine d'essai *breadboard* du bidouilleur amateur.

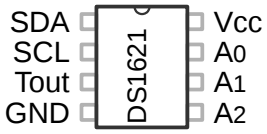


Fig. 5 : Brochage du boîtier DIP DS1621

Ce composant mesure la température ambiante (dans la plage -55°C à +125°C) et la communique, avec une résolution d'un demi-degré, par ses broches SDA et SCL. Il est accessible sur une adresse i²c dont les quatre bits de poids fort sont toujours **`1001`** et les trois bits de poids faibles configurables suivant la valeur indiquée sur les broches A_0, A_1 et A_2. Les adresses sont donc entre **`0x48`** (**`1001000b`** avec $A_0 = A_1 = A_2 = 0$) et **`0x4F`** (**`1001111b`** avec $A_0 = A_1 = A_2 = 1$).

On peut également programmer par l'interface i²c un seuil de température en deçà duquel la sortie T_{out} est à 0 et au-delà duquel elle passe automatiquement à 1.

Un driver pour DS1621 est déjà disponible dans le noyau Linux depuis longtemps, mais nous n'allons pas l'utiliser, ici, car c'est l'accès direct depuis l'espace utilisateur qui m'intéresse.

Vous trouverez sur de nombreux montages utilisant des communications en i²c des résistances de tirage reliant chacune des broches SDA et SCL (qui sont en collecteur ouvert) avec Vcc pour éviter que la tension flotte quand personne ne force de niveau bas sur ces lignes. Ceci est déjà réalisé par le circuit électronique du Raspberry Pi, deux résistances de 1,8 kOhm étant présentes entre les broches du BCM 2835 et l'alimentation +3.3V.

Nous relions tout naturellement la broche SDA du Raspberry Pi avec son homologue du DS1621, ainsi que les broches SCL des deux composants. On relie les deux broches de masse GND, et on peut utiliser la broche +3,3V du Raspberry Pi pour alimenter le DS1621 sur son entrée VCC.

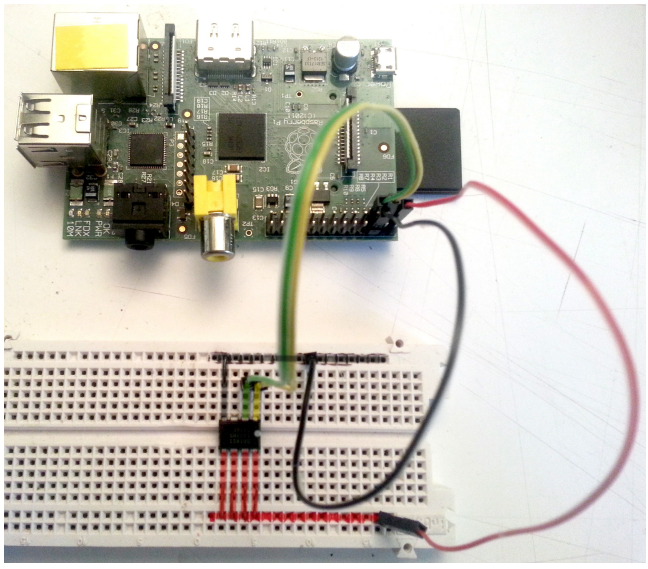


Fig 6 – Raspberry Pi et DS1621

On peut remarquer sur la photo (où j'ai surligné les connexions établies par la *breadboard*) que les broches A_0, A_1, A_2 du DS 1621 sont toutes reliées au +Vcc par l'intermédiaire de petits *straps* , ceci lui attribue donc

l'adresse **0x4F**.

Démarrons le Raspberry Pi et essayons de communiquer avec notre composant.

Accès depuis le shell

Bien que le module soit chargé, le noyau ne voit pas le DS1621. Le bus i²c, à l'opposé d'un bus PCIe par exemple ne permet pas d'énumérer et d'identifier automatiquement les périphériques présents. Pour que le kernel puisse le gérer directement, il faudrait lui indiquer la présence du DS1621 dans un fichier de configuration de la plate-forme avant la compilation du kernel ou dans un fichier de description *device tree* au moment du boot. Ceci n'est pas l'objet de cet article.

Après avoir chargé le module **i2c_bcm2708**, un second module va être nécessaire ; c'est celui qui va nous donner accès, depuis l'espace utilisateur aux périphériques connectés au bus i²c. Il les rendra accessibles à travers une interface dans le répertoire **/dev**, ce qui donne au module le nom **i2c_dev**.

```
root@raspberrypi:~# modprobe i2c_dev
root@raspberrypi:~# ls -l /dev/
total 0
 [...]
crw-rw---T 1 root i2c      89,   0 Jul 13 16:19 i2c-0
crw-rw---T 1 root i2c      89,   1 Jul 13 16:19 i2c-1
 [...]
root@raspberrypi:~#
```

Détection des périphériques

La manière la plus simple de communiquer depuis le shell est d'employer les utilitaires du package **i2c-tools**. Il faut tout d'abord installer ce dernier, car il n'est généralement pas inclus d'origine dans les distributions courantes :

```
root@raspberrypi:~# apt-get install i2c-tools
```

Nous pouvons commencer par lister les bus présents, puis scanner celui qui nous intéresse à l'aide de l'outil **i2cdetect**.

```
root@raspberrypi:~# i2cdetect -l
i2c-0   i2c             bcm2708_i2c.0           I2C adapter
i2c-1   i2c             bcm2708_i2c.1           I2C adapter
root@raspberrypi:~# i2cdetect 1
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-1.
I will probe address range 0x03-0x77.
Continue? [Y/n] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- UU -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- 4f
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

```
root@raspberrypi:~#
```

Le nom de la commande **i2cdetect** est suivi d'un **1** pour indiquer le numéro de bus à parcourir. On peut affiner la plage de recherche en indiquant des adresses de départ et de fin. Comme l'opération peut perturber certains périphériques, **i2cdetect** nous demande confirmation (à laquelle nous répondons **y**). Si nous souhaitons éviter cette confirmation (pour inclure l'invocation dans un script par exemple), il suffit d'ajouter l'option **-y** sur la ligne de commande.

Le tableau affiché nous indique les périphériques détectés (en remplissant la case avec leur adresse) et ceux dont la détection a été évitée car un driver s'en occupe déjà (avec les caractères '**UU**' comme c'est le cas du WM8804 à l'adresse **0x3b**).

Notre capteur de température a été détecté à l'adresse **0x4f**, ce qui est cohérent avec la configuration de ses entrées d'adresses.

Émission et réception de données.

Pour envoyer des ordres à un périphérique i²c depuis le shell, on utilise la commande **i2cset**. Celle-ci est conçue comme une interface pour remplir des registres distants. On lui passe en argument le numéro de bus et l'adresse du périphérique suivis du numéro de registre et de la valeur à y inscrire. Suivant les périphériques, ceci pourra également être interprété comme un numéro de commande suivi d'un ou plusieurs arguments.

Symétriquement, on emploiera **i2cget** pour lire l'état d'un registre distant. Ce qui peut aussi s'interpréter comme l'envoi d'une commande indiquant au périphérique de nous renvoyer immédiatement une valeur. Suivant les cas, on lira un ou plusieurs octets.

En consultant la documentation du DS1621, nous remarquons les commandes suivantes :

Nom commande	Numéro	Signification et arguments
START CONVERT	**0xEE**	Débute une mesure de température. Pas d'argument.
ACCESS CONFIG	**0xAC**	Lit l'état du convertisseur ou écrit sa configuration. La commande est suivie d'un seul octet. Le détail de la configuration est décrit dans la documentation du DS1621.
READ TEMPERATURE	**0xAA**	Demande à lire le résultat de la dernière acquisition de température.

Le DS1621 est un composant très simple. Il dispose de deux modes de fonctionnement : « o*ne shot* » (une seule mesure) ou « continu » (mesures de température en boucle permanente). Nous allons choisir le mode *one shot*.

Le principe de l'acquisition complète consistera donc à réaliser les étapes suivantes :

- écriture (avec la commande **ACCESS CONFIG**) dans le registre de configuration du bit **0x01** afin d'activer le mode *one shot* ;

- démarrage de la conversion avec **START CONVERT** ;

- lecture de l'état avec **ACCESS CONFIG** jusqu'à ce que le registre d'état contienne un bit de poids fort à 1 (**0x80**) ce qui indique que la mesure est terminée ;

- lecture du résultat avec **READ TEMPERATURE**.

Voici le script shell (**lecture-temperature.sh**) qui affiche la température ambiante.

```
#! /bin/sh

# Adresse du DS 1621.
I2C_BUS=1
I2C_ADDR=0x4F

# Commandes.
START_CONVERT=0xEE
ACCESS_CONFIG=0xAC
READ_TEMPERATURE=0xAA

# Lire la configuration.
status=$(i2cget -y ${I2C_BUS} ${I2C_ADDR} ${ACCESS_CONFIG})
      if [ $? -ne 0 ]; then exit 1; fi
```

```
# Ajouter le bit "One shot".
status=$((status | 0x01))

# Ecrire la configuration.
i2cset -y ${I2C_BUS} ${I2C_ADDR} ${ACCESS_CONFIG} $status
	if [ $? -ne 0 ]; then exit 1; fi

# Demarrer l'acquisition
i2cset -y ${I2C_BUS} ${I2C_ADDR} ${START_CONVERT}
	if [ $? -ne 0 ]; then exit 1; fi

while true
do
	# Lire l'etat.
	status=$(i2cget -y ${I2C_BUS} ${I2C_ADDR} ${ACCESS_CONFIG})
		if [ $? -ne 0 ]; then exit 1; fi

	# Si l'acquision est terminee, sortir de la boucle.
	if [ $(( $status & 0x80 )) -eq $((0x80)) ]; then break; fi
done

# Lire la temperature.
temp=$(i2cget -y ${I2C_BUS} ${I2C_ADDR} ${READ_TEMPERATURE})
	if [ $? -ne 0 ]; then exit 1; fi

# Gerer les valeurs negativeS
if [ $(($temp)) -gt 127 ]; then temp=$((temp - 256)); fi

# Ecrire la temperature sur la sortie standard.
echo $(($temp))
```

On peut faire divers essais pour vérifier le résultat. Par exemple le petit script suivant permet d'enregistrer la température au cours d'une expérience en l'horodatant toutes les secondes.

```
#! /bin/sh

PROG=./lecture-temperature.sh
d0=$(date +%s)

while true
do
	d=$(date +%s)
	printf "%d\t" $((d-d0))
	${PROG}
	sleep 1
done
```

Il devient très facile et amusant de visualiser (avec Gnuplot) les fluctuations de la température lors de diverses manipulations (poser une tasse de café sur le composant, placer la *breadboard* au congélateur, etc.)

Communication i²c avec d'autres langages

Accès avec Python

Langage favori de nombreux développeurs et hackers actuels, Python offre une interface pour la communication en i²c qui s'appuie sur les fonctionnalités **`i2c-dev`** que nous avons installées précédemment.

Il existe une bibliothèque permettant un accès simple aux périphériques qui supporte le sous-ensemble SMBus d'i²c. Les fonctions de haut-niveau permettent une communication avec le périphérique en dissimulant les multiples émissions/réceptions d'octets sous-jacentes. Installons sur notre Raspberry Pi cette bibliothèque.

```
root@raspberrypi:~# apt-get install python-smbus
```

Les principales méthodes sont :

`smbus.SMBus(`*n*`)`**	Renvoie un objet SMBus permettant l'accès au périphérique **`/dev/i2c-`***n*.

read_byte (adr) **write_byte (adr, val)**	Lit ou écrit un octet directement à l'adresse **adr**, sans préciser de registre.
read_byte_data (adr, reg) **write_byte_data (adr, reg, val)**	Lit ou écrit le contenu du registre **reg** du périphérique à l'adresse **adr** sous forme d'octet.
read_word_data (adr, reg) **write_word_data (adr, reg, val)**	Lit ou écrit lun mot de deux octets dans le registre **reg** du périphérique à l'adresse **adr**.
read_bloc_data (adr, reg) **write_bloc_data (adr, reg, val[])**	Lit ou écrit un tableau d'octets (maximum 32) dans le registre **reg** du périphérique à l'adresse **adr**. Les données sont précédées de la longueur du tableau.

Voici un script Python (**lecture-temperature.py**) qui réalise le même travail que le précédent script shell.

```
#! /usr/bin/python

import smbus

bus = smbus.SMBus(1)
addr= 0x4F

START_CONVERT=0xEE
ACCESS_CONFIG=0xAC
READ_TEMPERATURE=0xAA

bus.write_byte_data(addr, ACCESS_CONFIG, bus.read_byte_data(addr, ACCESS_CONFIG)| 0x01)

# Demarrer l'acquisition
bus.write_byte(addr, START_CONVERT)
while (bus.read_byte_data(addr, ACCESS_CONFIG) & 0x80) == 0:
   pass

v=bus.read_byte_data(addr, READ_TEMPERATURE)
if v > 127:
   v -= 256
print v
```

Accès en C

Si l'on choisit le langage C pour communiquer en i²c avec un périphérique, ce sera probablement pour des raisons d'efficacité, de rapidité, et de contrôle des opérations réalisées. L'API employée pourra néanmoins être proche de celles des autres langages.

Nous emploierons l'interface **i2c-dev** car c'est la meilleure pour accéder depuis l'espace utilisateur, mais nous lui enverrons des ordres spécifiques en utilisant le mécanisme des *ioctl* (*I/O Controls*). Nous ouvrirons donc **open()** le bus /**dev/i2c-***n* qui nous intéresse, et ferons des accès avec l'appel-système **ioctl()**.

```
int open (const char * filename, int flags, mode_t mode);
int ioctl (int fd, int request...);
```

Par exemple pour indiquer une fois pour toute l'adresse du périphérique nous utiliserons un appel avec l'argument **I2C_SLAVE** défini dans **/usr/include/linux/i2c-dev.h**. Il serait possible de coder tous les échanges au protocole SMBus avec des **ioctl()**, mais cela serait très fastidieux. Aussi existe-t-il une mini-bibliothèque, qui propose des fonctions *inline* s'appuyant sur les **ioctl()** bas-niveaux. La particularité de cette bibliothèque, et qu'elle s'installe en modifiant le fichier **i2c-dev.h** du système pour y inscrire ses fonctions *inline*..

On l'installe ainsi :

```
# apt-get install libi2c-dev
```

Nous disposons alors entre autres des fonctions :

```
__s32 i2c_smbus_write_quick(int fd, __u8 val)
__s32 i2c_smbus_read_byte(int fd)
__s32 i2c_smbus_write_byte(int fd, __u8 val)
__s32 i2c_smbus_read_byte_data(int fd, __u8 cmd)
__s32 i2c_smbus_write_byte_data(int fd, __u8 cmd, __u8 val)
__s32 i2c_smbus_read_word_data(int fd, __u8 cmd)
```

```
__s32 i2c_smbus_write_word_data(int fd, __u8 cmd, __u16 val)
__s32 i2c_smbus_read_block_data(int fd, __u8 cmd, __u8 *val)
__s32 i2c_smbus_write_block_data(int fd, __u8 cmd, __u8 lg, const __u8 *val)
```

Les fonctions qui nous intéressent sont essentiellement celles suffixées par **_data**, car elles permettent d'indiquer un numéro de registre (de commande) et les données à lire ou écrire.

Voici donc le programme qui réalise le même travail que les précédents. Son écriture est plus longue car les traitements d'erreur sont écrits de manière plus complète.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>

#define DS1621_I2C_BUS          "/dev/i2c-1"
#define DS1621_SLAVE_ADDR       0x4F
#define DS1621_ACCESS_CONFIG    0xAC
#define DS1621_START_CONVERT    0xEE
#define DS1621_READ_TEMPERATURE 0xAA
#define DS1621_ONE_SHOT         0x01
#define DS1621_DONE             0x80


int main(int argc, char * argv[])
{
        int fd;
        int value;

        // Obtenir l'acces au bus i2c.
        fd = open(DS1621_I2C_BUS, O_RDWR);
        if (fd < 0) {
                perror(DS1621_I2C_BUS);
                exit(EXIT_FAILURE);
        }

        // Fixer l'adresse de l'esclave avec qui communiquer.
        if (ioctl(fd, I2C_SLAVE, DS1621_SLAVE_ADDR) < 0) {
                perror("Slave unreachable");
                exit(EXIT_FAILURE);
        }

        // Lire le registre ACCESS_CONFIG.
        value = i2c_smbus_read_byte_data(fd, DS1621_ACCESS_CONFIG);
        if (value < 0) {
                perror("Read ACCESS_CONFIG");
                exit(EXIT_FAILURE);
        }

        value |= DS1621_ONE_SHOT;

        // Re-ecrire ACCESS_CONFIG
        if (i2c_smbus_write_byte_data(fd, DS1621_ACCESS_CONFIG, value) < 0) {
                perror("Write ACCESS_CONFIG");
                exit(EXIT_FAILURE);
        }

        // Demarrer l'acquisition.
        if (i2c_smbus_write_byte(fd, DS1621_START_CONVERT) < 0) {
                perror("Write START_CONVERT");
                exit(EXIT_FAILURE);
        }
        // Attendre qu'une acquisition soit terminee.
        do {
                value = i2c_smbus_read_byte_data(fd, DS1621_ACCESS_CONFIG);
                if (value < 0) {
                        perror("Read ACCESS_CONFIG");
```

```
                exit(EXIT_FAILURE);
            }
        } while ((value & DS1621_DONE) == 0);

        // Lire la temperature.
        value = i2c_smbus_read_byte_data(fd, DS1621_READ_TEMPERATURE);
        if (value < 0) {
            perror("Read TEMPERATURE");
            exit(EXIT_FAILURE);
        }

        if (value > 127)
            value -= 256;
        fprintf(stdout, "%d\n", value);

        return EXIT_SUCCESS;
}
```

On peut choisir de le *cross-compiler* sur un PC puis de transférer le code exécutable sur la cible, à l'instar des pratiques usuelles en programmation embarquée. Il est sans doute plus simple de le compiler directement sur le Raspberry Pi car la distribution Raspbian, comme la plupart des autres, inclut un compilateur **gcc**.

```
# gcc lecture-temperature.c -o lecture-temperature -Wall
# ./lecture-temperature
26
#
```

Conclusion

Nous avons réussi à communiquer facilement avec un périphérique par l'intermédiaire du bus i²c présent sur le Raspberry Pi, en utilisant trois langages différents. Nous avons employé un composant DS1621 pour sa facilité de mise en œuvre, mais il existe de très nombreux périphériques que l'on peut piloter avec l'i²c (capteurs en tous genres, afficheurs, télécommandes, etc.) et la simplicité de la connexion rend aisée l'expérimentation pour le *hacker* amateur.

Pour en savoir plus :

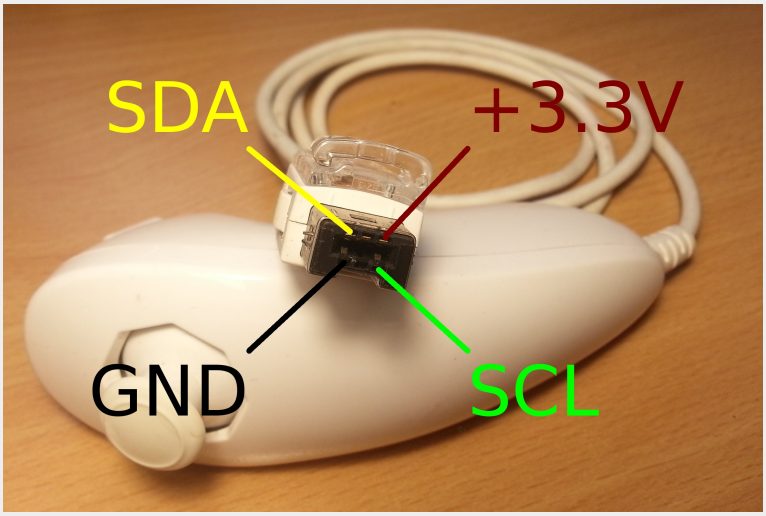
La page i²c de Wikipedia en langue française (**http://fr.wikipedia.org/wiki/I2C**) est claire et complète

La description du protocole SMBus sur son site officiel (**http://smbus.org**) est un peu ardue, mais on en trouve de nombreux résumés sur le web.

La documentation technique du DS1621 (**http://datasheets.maximintegrated.com/en/ds/DS1621.pdf**) courte, complète et facilement compréhensible ce qui n'est pas souvent le cas dans les documentations de composants électroniques !

Les scripts et programmes sources sont disponibles sur un dépôt Github à l'adresse **https://github.com/cpb-/Article-RPi-DS1621.git**

Une autre utilisation amusante du bus i²c est la communication avec un « *nunchunk* », extension de manette de jeu pour la console Wii, qui regroupe un joystick analogique, deux boutons et un accéléromètre trois axes. Il s'agit en effet d'un périphérique i²c SMBus tout à fait standard. Il est assez facile de glisser des petits câbles fins à l'extérieur des languettes du connecteur (au format propriétaire), permettant ainsi une connexion temporaire sans avoir besoin de couper le câble. On notera que les deux broches centrales ne sont pas utilisées. Le *nunchunk* s'alimente en principe en +3V mais j'ai constaté un fonctionnement plus fiable en le branchant sur le +5V (fourni par le Raspberry Pi).



Il existe deux modèles principaux de *nunchunks* : le blanc et le noir, qui s'installent tous deux à l'adresse **`0x52`** mais utilisent des séquences d'initialisation différentes. Pour le *nunchunk* blanc, il faut écrire un **`0x00`** dans le registre **`0x40`**, tandis que pour le *nunchunck* noir on doit écrire un **`0x55`** dans le registre **`0xF0`** suivi d'un **`0x00`** dans le registre **`0xFB`**.

Après quelques millisecondes d'initialisation, on pourra interroger périodiquement le *nunchunk* en lui envoyant d'abord un octet **`0x00`** pour demander une acquisition des capteurs, puis en lisant six octets décrits dans le tableau suivant.

Octet	Signification
1	Composante X du joystick (0-255)
2	Composante Y du joystick (0-255)
3	8 bits de poids fort de la composante X de l'accéléromètre.
4	8 bits de poids fort de la composante Y de l'accéléromètre.
5	8 bits de poids fort de la composante Z de l'accéléromètre.
6	État des boutons (2 bits) + 2 bits de poids faible de chacune des composantes de l'accéléromètre.

Comme on le voit, les composantes de l'accéléromètre sont fournies sur 10 bits (0-1023). Le programme suivant permet d'afficher en permanence les différentes valeurs.

```
#! /usr/bin/python

import smbus
import sys
import time

try:
	bus = smbus.SMBus(1)
except IOError:
	print "Please load i2C_bcm2708 and i2c_dev modules"
	sys.exit(1)
```

```
# Initialiser le nunchnuk (adresse = 0x52)
# Ecrire 0x00 dans le registre 0x40 pour le nunchunk blanc
# Ecrire 0x55 dans le registre 0xF0 puis 0x00 dans 0xFB pour le noir.
try:
        bus.write_byte_data(0x52, 0x40, 0x00)
except IOError:
        print "No nunchunk found"
        sys.exit(1)

# Attendre l'initialisation.
time.sleep(0.01)

while True:
    try:
        # Envoyer un 0x00 pour demander l'acquisition.
        bus.write_byte(0x52,0x00)
        # Attendre quelques millisecondes.
        time.sleep(0.05)
        joystick_x      =  bus.read_byte(0x52)
        joystick_y      =  bus.read_byte(0x52)
        accelerometer_x =  bus.read_byte(0x52)
        accelerometer_y =  bus.read_byte(0x52)
        accelerometer_z =  bus.read_byte(0x52)
        miscellaneous   =  bus.read_byte(0x52)
        accelerometer_x = accelerometer_x << 2
        accelerometer_y = accelerometer_y << 2
        accelerometer_z = accelerometer_z << 2
        accelerometer_x += (miscellaneous & 0x0C) >> 2
        accelerometer_y += (miscellaneous & 0x30) >> 4
        accelerometer_z += (miscellaneous & 0xC0) >> 6
        if ((miscellaneous & 0x03) == 0):
            button_c = 0
            button_z = 1
        elif ((miscellaneous & 0x03) == 1):
            button_c = 1
            button_z = 0
        elif ((miscellaneous & 0x03) == 2):
            button_c = 1
            button_z = 1
        else:
            button_c = 0
            button_z = 0
        sys.stdout.write('Joystick: (%3d, %3d), ' % (
              joystick_x, joystick_y))
        sys.stdout.write('Accelerometer: (%4d, %4d, %4d), ' %(
              accelerometer_x, accelerometer_y, accelerometer_z))
        sys.stdout.write('Buttons: (%d,%d)\n'     % (
                    button_c, button_z))
    except KeyboardInterrupt:
        sys.exit(0)
    except IOError:
        print "Nunchunk disconnected"
        sys.exit(1)
```