



Conception et implémentation des objets connectés

Dr. Ing. Chiheb Ameur ABID

Contact: chiheb.abid@fst.utm.tn

Année universitaire : 2021 - 2022



Plan

- 1 Entrées/Sorties GPIO
- 2 Programmation concurrentielle
- 3 Les interruptions
- 4 Programmation avec les machines à états
- 5 Les canaux Pulse Width Modulation (PWM)
- 6 Fonctions de communication
- 7 Communication UART
- 8 Communication sur un bus I2C (Inter Integrated Circuit)
- 9 Communication avec l'interface SPI (Serial Peripheral Interface)



Présentation du port GPIO

Présentation du GPIO sur RPi3

GPIO (General Purpose Input/Output) est un port d'entrées-sorties très utilisés dans le monde des microcontrôleurs et de l'électronique embarquée.

- Les GPIO sont gérés par les pilotes du noyau du système d'exploitation. **Il n'y a pas d'entrée/sortie analogique**
- Linux reconnaît nativement les ports GPIO, une documentation complète est même disponible : www.kernel.org/doc/Documentation/gpio/gpio.txt
- Depuis la version 4.8 du noyau Linux, les GPIO sont accessibles par le pilote de périphérique ABI (Application Binary Interface) chardev GPIO via `/dev/gpiochipN` ou `/sys/bus/gpio`



Présentation du port GPIO

Brochage

Raspberry Pi GPIO BCM numbering



Présentation du port GPIO

Brochage

Raspberry Pi 2 Model B (35 Header) pinout

WiringPi	BCM(Name)	Physical	Physical	BCM(Name)	WiringPi
	3v3 Power			5v Power	
8	BCM 2 (SDA)			5v Power	
9	BCM 3 (SCL)			Ground	
7	BCM 4 (GCLK0)			BCM 14 (TX0)	15
Ground				BCM 15 (RX0)	16
0	BCM 17			BCM 18 (PCM_C)	1
2	BCM 27 (PCM_D)			Ground	
3	BCM 22			BCM 23	4
	3v3 Power			BCM 24	5
12	BCM 10 (MOSI)			Ground	
13	BCM 9 (MISO)			BCM 25	6
14	BCM 11 (SCLK)			BCM 8 (CE0)	10
Ground				BCM 7 (CE1)	11
	BCM 0 (ID_SD)			BCM 1 (ID_SC)	
21	BCM 5			Ground	
22	BCM 6			BCM 12	26
23	BCM 13			Ground	
24	BCM 19 (MISO)			BCM 16	27
25	BCM 26			BCM 20 (MOSI)	28
Ground				BCM 21 (SCLK)	29

Présentation du port GPIO

Usage

- Une sortie peut être fixée sur un niveau haut (3V3) ou bas (0V). Une entrée peut être lue comme un niveau haut (3V3) ou bas (0V).
- Il est possible d'utiliser de résistances internes pull-up ou pull-down. Les GPIO2 et GPIO3 ont des résistances de pull-up fixes, mais pour les autres broches, elles peuvent être configurées logiquement.
- Software PWM (Pulse-Width Modulation) disponible sur toutes les broches et Hardware PWM seulement sur GPIO12, GPIO13 et GPIO18
- I2C : SDA (GPIO2) SCL (GPIO3) et EEPROM Data (GPIO0) EEPROM Clock (GPIO1)
- Port série UART : TX (GPIO14) RX (GPIO15)
- SPI
 - SPI0 : MOSI (GPIO10) MISO (GPIO9) SCLK (GPIO11) CE0 (GPIO8) CE1 (GPIO7)
 - SPI1 : MOSI (GPIO20) MISO (GPIO19) SCLK (GPIO21) CE0 (GPIO18) CE1 (GPIO17) CE2 (GPIO16)

Avertissement

L'alimentation maximale pour chaque broche est de 16mA. L'alimentation maximale pour toutes les broches est de 51mA.

Niveaux logiques des entrées/sorties Tout Ou Rien (TOR)

Tout Ou Rien (TOR)

- Une entrée/sortie tout ou rien est une entrée/sortie binaire. Soit elle prend la valeur 0, ou la valeur 1
- La carte Raspberry est basée sur la technologie CMOS : Complementary Metal-Oxide Semiconductor
- Les niveaux logiques selon la technologie CMOS (des valeurs approximatives)

Symbol	Minimum	Maximum	Description
V_{IL}	0.0 volts	$1/3 V_{DD}$	Logic value false (0)
V_{IH}	$2/3 V_{DD}$	V_{DD}	Logic value true (1)

- Les niveaux logiques de la RPi

Symbol	Low	High	Description
V_{IL}	0.0 volts	0.8 volts	Logic value false (0)
V_{IH}	1.3 volts	V_{DD}	Logic value true (1)

Les entrées/sorties TOR

Les résistances PULL-UP/PULL-DOWN

- Si une entrée est laissée libre (non connectée), son potentiel sera flottant en fonction de l'électricité statique ou des perturbations électromagnétiques.
 - Utiliser une résistance soit pour tirer vers le haut le potentiel (3.3V), ou pour tirer vers le bas (0V)
- Résistance PULL-UP : au repos, le potentiel de l'entrée est à 3.3 V.
- Résistance PULL-DOWN : au repos, le potentiel de l'entrée est à 0 V.

Accès aux GPIOs avec pigpio

Compilation native avec la bibliothèque pigpio

- On utilise la bibliothèque C de pigpio
- On utilise le dialecte moderne C++17
- Compilation native depuis la ligne de commande sur la Raspberry Pi
`g++ nomProgramme.cpp -o nomExecutable -std=c++17 -lpigpio`
- Les programmes utilisant pigpio doivent avoir les droits d'accès de l'utilisateur root pour s'exécuter
`sudo ./nomExecutable`



Accès aux GPIOs avec pigpio

Fonctions pigpio

- `int gpioInitialise(void)` : Initialiser la bibliothèque
 - ☞ Doit être appelée avant d'utiliser les autres fonctions de la bibliothèque
 - ☞ Elle retourne la version de la bibliothèque si l'initialisation est effectuée correctement, sinon elle retourne `PI_INIT_FAILED`
- `void gpioTerminate(void)` : Terminer l'utilisation de la bibliothèque
 - ☞ Libérer les ressources utilisées



Accès aux GPIOs avec pigpio

Fonctions pigpio

- `int gpioSetMode(unsigned gpio, unsigned mode)` : spécifier le mode d'accès à une broche
 - ☞ `gpio` : 0-53
 - ☞ `mode` : 0-7; `PI_INPUT`, `PI_OUTPUT`, `PI_ALT0`, `PI_ALT1`, `PI_ALT2`, `PI_ALT3`, `PI_ALT4` ou `PI_ALT5`
- `int gpioGetMode(unsigned gpio)` : retourne le mode d'accès d'une broche
 - ☞ `gpio` : 0-53
 - ☞ S'il y a une erreur, elle retourne `PI_BAD_GPIO`
- `int gpioSetPullUpDown(unsigned gpio, unsigned pud)` : définir une résistance pull-up ou pull-down à une broche
 - ☞ `pud` : 0-2; `PI_PUD_OFF`, `PI_PUD_DOWN` ou `PI_PUD_UP`



Accès aux GPIOs avec pigpio

Fonctions pigpio

- `int gpioWrite(unsigned gpio, unsigned level)` : spécifier l'état d'une broche de sortie
 - ☞ `gpio` : 0-53
 - ☞ `level` : 0 ou 1
 - ☞ Retourne 0 si OK, sinon `PI_BAD_GPIO` ou `PI_BAD_LEVEL`
 - ☞ Elle désactive la génération d'un signal PWM
- `int gpioRead(unsigned gpio)` : retourne l'état d'une broche
 - ☞ `gpio` : 0-53
 - ☞ S'il y a une erreur, elle retourne `PI_BAD_GPIO`

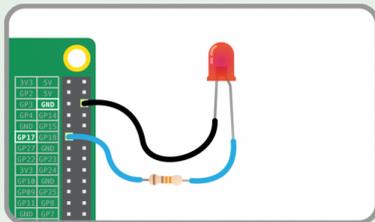


Accès aux GPIOs avec C et C++

Accès aux GPIOs avec C et C++

Exemple : flasher une diode LED

On considère le montage suivant



Exemple : flasher une diode LED

```
1 #include <iostream>
2 #include <unistd.h>
3 #include <pigpio.h>
4
5 constexpr uint8_t O_LED=17; // Using GPIO17
6 using namespace std;
7 int main(int argc, char *argv[]) {
8     cout << "Running pigpio program" << endl;
9     gpioInitialise(); // you MUST initialize the library !
10    gpioSetMode(O_LED, PI_OUTPUT);
11    while (1) {
12        gpioWrite(O_LED, 1);
13        sleep(1);
14        gpioWrite(O_LED, 0);
15        sleep(1);
16    }
17    cout << "gpioTerminate()..." << endl;
18    gpioTerminate(); // call this when done with library
19    return 0;
20 }
```

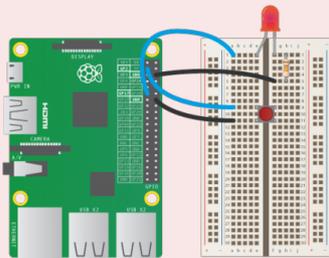


Accès aux GPIOs avec C et C++

Accès aux GPIOs avec C et C++

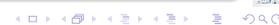
Exercice : contrôler l'état d'une diode LED à l'aide d'un bouton

Écrire un programme qui change l'état de la diode LED à chaque appuie sur le bouton poussoir



Exercice : contrôler l'état d'une diode LED à l'aide d'un bouton

```
1 #include <iostream>
2 #include <pigpio.h>
3
4 constexpr uint8_t O_LED=17; // Using GPIO17
5 constexpr uint8_t I_PUSH=2; // Using GPIO2
6
7 int main(int argc, char *argv[]) {
8     if (gpioInitialise() < 0) {
9         std::cout << "Failure..." << endl;
10        exit(-1);
11    }
12    gpioSetMode(O_LED, PI_OUTPUT);
13    gpioSetMode(I_PUSH, PI_INPUT);
14    gpioSetPullUpDown(I_PUSH, PI_PUD_UP);
15    while (1) {
16        if (gpioRead(I_PUSH) == 0) {
17            while (gpioRead(I_PUSH) == 0) ;
18            gpioWrite(O_LED, !gpioRead(O_LED));
19        }
20    }
21    gpioTerminate();
22    return 0;
23 }
```

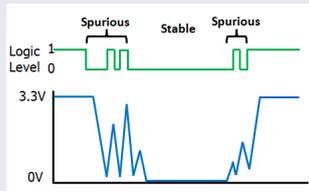


Accès aux GPIOs avec C et C++

Accès aux GPIOs avec C et C++

Problème de parasitage des contacts

- Utiliser un registre (variable de type `int`) pour vérifier que le signal se stabilise



Problème de parasitage des contacts

- Utiliser un registre (variable de type `uint32_t`) pour vérifier que le signal se stabilise
 - On procède par décalage : on lit un bit de valeur et on le met, par décalage, dans le registre
 - On s'arrête lorsque le signal se stabilise, i.e. tous les bits du registre sont identiques

```
1 uint32_t registre=0;
2 if (gpioRead(ENTREE)==0) {
3     while (registre!=0) { // Front descendant
4         registre=(registre <<1) | gpioRead(ENTREE);
5     }
6     while (gpioRead(ENTREE)==0);
7     while (registre!=(-0)) { // Front montant
8         registre=(registre <<1) | gpioRead(ENTREE);
9     }
10 }
```

Plan

Programmation Parallèle

- Entrées/Sorties GPIO
- Programmation concurrentielle
- Les interruptions
- Programmation avec les machines à états
- Les canaux Pulse Width Modulation (PWM)
- Fonctions de communication
- Communication UART
- Communication sur un bus I2C (Inter Integrated Circuit)
- Communication avec l'interface SPI (Serial Peripheral Interface)

Les threads

- Programmation concurrente
 - Travailler sur plusieurs tâches à la fois
- Un thread (tâche) représente (généralement) une fonction (une méthode) en cours d'exécution
- Un processus peut contenir un ou plusieurs threads. Ces threads partagent alors la mémoire ainsi que diverses autres ressources*
 - Deux threads qui veulent collaborer peuvent le faire par l'intermédiaire de variables partagées.
 - Le contexte d'un thread est léger par rapport à un processus

Programmation concurrentielle

Programmation concurrentielle

Création d'un thread

```
1 #include <iostream>
2 #include <thread>
3 void hello()
4 {
5     std::cout<<"Hello_Concurrent_World\n";
6 }
7 int main()
8 {
9     std::thread t(hello);
10    t.join();
11 }
```



Section critique

- Partie d'un thread dont l'exécution ne doit pas entrelacer avec d'autres threads
 - ☞ Indivisibilité de la section critique
- Une fois un thread entre dans la section critique
 - ☞ Le thread qui entre dans la SC doit terminer son travail en empêchant les autres threads de jouer sur les mêmes données
 - ☞ La section critique doit être verrouillée afin de devenir invisible



Programmation concurrentielle

Programmation concurrentielle

Problème de section critique

- Lorsqu'un thread manipule une donnée (ressource) partagée avec d'autres, nous disons qu'il se trouve dans une section critique
- Le problème de la SC est de trouver un algorithme d'exclusion mutuelle de threads
 - ☞ Les résultats de leurs actions ne dépend pas de l'ordre de leur entrelacement
- L'exécution de la section critique doit être mutuellement exclusive et indivisible
 - ☞ Un seul thread exécute la SC



Coordination par les mutex

- Un mutex (mutual exclusion) est une structure de données qui permet de contrôler l'accès à une ressource
- Un mutex qui contrôle une ressource peut se trouver dans deux états
 - 1 Libre (unlocked) : indique que la ressource est libre et peut être utilisée sans risquer de provoquer une violation d'exclusion mutuelle
 - 2 Réservée (locked) : indique que la ressource associée est actuellement utilisée et qu'elle ne peut pas être utilisée par un autre thread



Programmation concurrentielle

Les types atomiques standards

Mutex : protection des variables partagées

```

1 #include <list>
2 #include <mutex>
3 #include <algorithm>
4 std::list<int> some_list;
5 std::mutex some_mutex;
6 void add_to_list(int new_value) // Thread 1
7 {
8     std::lock_guard guard(some_mutex);
9     some_list.push_back(new_value);
10 }
11 bool list_contains(int value_to_find) // Thread 2
12 {
13     std::lock_guard<std::mutex> guard(some_mutex);
14     return std::find(some_list.begin(), some_list.end(), value_to_find) != some_list.end();
15 }
    
```

- ❶ std::list<int> some_list : variable globale, donc elle est une variable partagée
- ❷ std::mutex some_mutex : instance de mutex, le même mutex doit être accessibles aux différents threads
- ❸ std::lock_guard guard(some_mutex) : protège la section critique

Les types atomiques standards

- Une opération atomique est une opération indivisible
- Les types atomiques sont des types supportant les opérations atomiques
- Les types atomiques standards sont déclarés dans <atomic>
- Ces types peuvent ne pas être lock-free
 - ❶ Implémentés ou non à l'aide des mutex

Les types atomiques standards

Les types atomiques standards

➤ Ces types peuvent être non lock-free

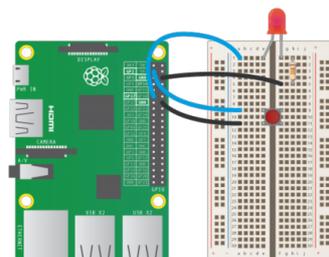
Type atomique	Spécialisation std::atomic<>
atomic_bool	std::atomic<bool>
atomic_char	std::atomic<char>
atomic_schar	std::atomic<signed char>
atomic_uchar	std::atomic<unsigned char>
atomic_int	std::atomic<int>
atomic_uint	std::atomic<unsigned>
atomic_short	std::atomic<short>
atomic_ushort	std::atomic<unsigned short>
atomic_long	std::atomic<long>
atomic_ulong	std::atomic<unsigned long>
atomic_llong	std::atomic<long long>
atomic_ullong	std::atomic<unsigned long long>
atomic_char16_t	std::atomic<char16_t>
atomic_char32_t	std::atomic<char32_t>
atomic_wchar_t	std::atomic<wchar_t>

Programmation concurrentielle

Mise en application

Créer deux threads :

- ❶ Un premier thread flashe une diode LED connectée à BCM17 à chaque seconde
- ❷ Un deuxième thread active/désactive le flashage de la diode LED à chaque appuie d'un bouton poussoir connecté à BCM2



Programmation concurrentielle

Mise en application

```
1 #include <iostream>
2 #include <unistd.h>
3 #include <pigpio.h>
4 #include <thread>
5 #include <atomic>
6 constexpr uint8_t O_LED = 17; // BCM17
7 constexpr uint8_t I_BUTTON = 2; // BCM2
8 volatile std::atomic<bool> flashEnabled = false; //volatile: tell compiler to not optimize
9
10 void flash() { // Flash the LED
11     while (1)
12         if (flashEnabled) {
13             gpioWrite(O_LED, 1);
14             sleep(1);
15             gpioWrite(O_LED, 0);
16             sleep(1);
17         }
18 }
19
20 void readButtonState() { // Check whether the Button pressed
21     while (1)
22         if (gpioRead(I_BUTTON)==0) {
23             while (gpioRead(I_BUTTON)==0);
24             flashEnabled=!flashEnabled;
25         }
26 }
```



Programmation concurrentielle

Mise en application

```
1 int main(int argc, char *argv[]) {
2     if (gpioInitialise() < 0) {
3         std::cout << "Echec_d'initialisation...\n";
4         exit(-1);
5     }
6     gpioSetMode(O_LED, PI_OUTPUT);
7     gpioSetMode(I_BUTTON, PI_INPUT);
8     gpioSetPullUpDown(I_BUTTON, PI_PUD_UP);
9     std::thread t1(flash);
10    std::thread t2(readButtonState);
11    while (1);
12    gpioTerminate();
13    return 0;
14 }
```



Plan

- 1 Entrées/Sorties GPIO
- 2 Programmation concurrentielle
- 3 Les interruptions
- 4 Programmation avec les machines à états
- 5 Les canaux Pulse Width Modulation (PWM)
- 6 Fonctions de communication
- 7 Communication UART
- 8 Communication sur un bus I2C (Inter Integrated Circuit)
- 9 Communication avec l'interface SPI (Serial Peripheral Interface)



L'intérêt des interruptions

Comment vérifier une condition/événement ?

➔ Vérifier qu'une broche d'entrée prends la valeur 1 !?

1 Attente active

```
1 while (1) {
2     while (gpioRead(BROCHE)==0);
3     Action 1
4     Action 2
5     ...
6     Action n
7 }
```

✘ Le processeur est bloqué sur la vérification de l'évènement



L'intérêt des interruptions

Comment vérifier une condition/événement ?

↳ Vérifier qu'une broche d'entrée prends la valeur 1 !?

② Lecture par scrutation (Polling)

```

1 while (1) {
2   Action 1
3   Action 2
4   ...
5   if (gpioRead(BROCHE)==1) Action();
6   Action i
7   ...
8   Action n
9 }

```

- ✓ Simple à mettre en oeuvre
- ✗ Il est possible de louper certains évènements
- ✗ Consomme du temps processeur



Interruptions

Comment vérifier une condition/événement ?

↳ Vérifier qu'une broche d'entrée prends la valeur 1 !?

③ Les interruptions

```

1  /**
2  * Programme principal
3  */
4  while (1) {
5    Action 1
6    Action 2
7    ...
8    Action n
9  }
10
11 /**
12 * Routine de gestion d'interruption
13 */
14 void ISR() {
15   Actions
16 }

```

- ✓ La fonction ISR est appelée à chaque demande d'interruption (IRQ)
- ✓ La charge de vérification du changement de la valeur est déléguée au contrôleur d'interruptions



Interruptions

Détection de changement de l'état d'une entrée

- ↳ Libérer le processeur en confiant la surveillance au contrôleur d'interruptions
- ↳ Lorsque un changement provient sur une entrée ou suite à un évènement, le processeur est avisé à travers une demande d'interruption (Interrupt ReQuest) afin d'interrompre son travail actuel et exécuter le traitement adéquat en faisant appel à une fonction ISR (Interrupt Service Routine)
 - ✓ Alléger la charge du processeur
 - ✓ Réaction plus rapide aux évènements reçus

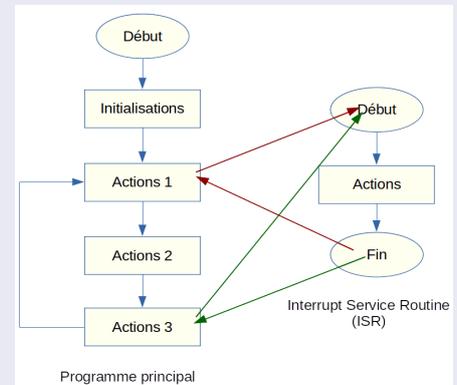
Avertissement

Raspberry OS n'est pas un système temps réel



Interruptions

Gestion des interruptions



Interruptions et GPIO

Gestion des interruptions avec pigpio

- ▶ `int gpioSetAlertFunc(unsigned user_gpio, gpioAlertFunc_t f)` : spécifier une fonction (callback) appelée suite à un changement d'état (demande d'interruption) d'une broche
 - `user_gpio`: 0-31
 - `f`: la routine de gestion d'interruption. `f` doit être écrite selon le type typedef `void (*gpioAlertFunc_t)(int gpio, int level, uint32_t tick)`;
 - Retourne 0 si OK, sinon `PI_BAD_USER_GPIO`
- ▶ `int gpioSetAlertFuncEx(unsigned user_gpio, gpioAlertFuncEx_t f, void *userdata)` : permet aussi de spécifier des données arbitraires à travers `userdata`

Avertissement

- ▶ Une seule fonction callback peut être associée à une broche
- ▶ Pour désactiver la fonction callback, il faut passer la valeur `nullptr` au deuxième argument de l'appel de `gpioSetAlertFunc()` ou de `gpioSetAlertFuncEx()`



Interruptions et GPIO

Exemple : changer l'état d'une diode LED à chaque appuie d'un bouton poussoir

```
1 #include <iostream>
2 #include <unistd.h>
3 #include <pigpio.h>
4
5 constexpr uint8_t O_LED=17; // Using GPIO17
6 constexpr uint8_t I_PUSH=2; // Using GPIO2
7 using namespace std;
8 void cbPushButton(int gpio, int level, uint32_t tick) {
9     if (level==0) gpioWrite(O_LED,!gpioRead(O_LED));
10 }
11 int main(int argc, char *argv[]) {
12     cout << "Running_pigpio_program" << endl;
13     if (gpioInitialise()<0) {
14         cout<<"Failure..."<<endl;
15         exit(-1);
16     }
17     gpioSetMode(O_LED, PI_OUTPUT);
18     gpioSetMode(I_PUSH,PI_INPUT);
19     gpioSetPullUpDown(I_PUSH,PI_PUD_UP);
20     gpioSetAlertFunc(I_PUSH,cbPushButton);
21     while (1);
22     gpioTerminate();
23     return 0;
24 }
```



Interruptions et GPIO

Gestion des interruptions avec pigpio

- ▶ `int gpioSetISRFunc(unsigned user_gpio, unsigned edge, int timeout, gpioISRFunc_t f)` : spécifier l'ISR relative à une interruption produite par une broche
 - `user_gpio`: 0-31
 - `edge`: 0-2: `RISING_EDGE`, `FALLING_EDGE` ou `EITHER_EDGE`
 - `timeout` : temps maximal pour gérer une demande d'interruption. Valeur négative pour désactiver le timeout
 - `f`: la routine de gestion d'interruption. `f` doit être écrite selon le type typedef `void (*gpioISRFunc_t)(int gpio, int level, uint32_t tick)`;
 - Retourne 0 si OK, sinon `PI_BAD_USER_GPIO`, `PI_BAD_EDGE` or `PI_BAD_ISR_INIT`
- ▶ `int gpioSetISRFuncEx(unsigned user_gpio, unsigned edge, int timeout, gpioISRFunc_t f, void *userdata)` : permet aussi de spécifier des données arbitraires à travers `userdata`



Interruptions et GPIO

Exemple : changer l'état d'une diode LED à chaque appuie d'un bouton poussoir

```
1 #include <iostream>
2 #include <unistd.h>
3 #include <pigpio.h>
4
5 constexpr uint8_t O_LED=17; // Using GPIO17
6 constexpr uint8_t I_PUSH=2; // Using GPIO2
7 using namespace std;
8 void cbPushButton(int gpio, int level, uint32_t tick) {
9     gpioWrite(O_LED,!gpioRead(O_LED));
10 }
11 int main(int argc, char *argv[]) {
12     cout << "Running_pigpio_program" << endl;
13     if (gpioInitialise()<0) {
14         cout<<"Failure..."<<endl;
15         exit(-1);
16     }
17     gpioSetMode(O_LED, PI_OUTPUT);
18     gpioSetMode(I_PUSH,PI_INPUT);
19     gpioSetPullUpDown(I_PUSH,PI_PUD_UP);
20     gpioSetISRFunc(I_PUSH,FALLING_EDGE,cbPushButton);
21     while (1);
22     gpioTerminate();
23     return 0;
24 }
```



Plan

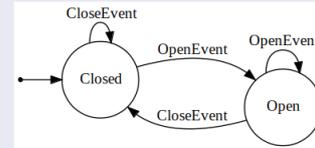
- 1 Entrées/Sorties GPIO
- 2 Programmation concurrentielle
- 3 Les interruptions
- 4 Programmation avec les machines à états**
- 5 Les canaux Pulse Width Modulation (PWM)
- 6 Fonctions de communication
- 7 Communication UART
- 8 Communication sur un bus I2C (Inter Integrated Circuit)
- 9 Communication avec l'interface SPI (Serial Peripheral Interface)



Les machines à états

Les machines à états

- ↳ Un automate fini ou machine à états finis (finite state machine) est un modèle mathématique pour la modélisation de comportement de systèmes numériques
 - ▣ Son rôle, est de décrire le fonctionnement d'une machine (ou d'un objet) ayant un comportement **séquentiel**
 - ▣ Modèle graphique
 - ▣ Utilisé dans de nombreux domaines : conception de programmes informatiques, protocoles de communication, contrôle des processus, analyse linguistique, etc.



Les machines à états

Les machines à états

- ↳ Dans un système séquentiel, le comportement d'une machine (ou d'un objet) dépend non seulement des évènements qu'il reçoit mais aussi de ce qui s'est passé avant ces évènements.
 - ▣ Par exemple, on ne peut pas faire descendre un ascenseur s'il est déjà en bas
 - ▣ Les machines ayant un fonctionnement séquentiel passent par un nombre de situations (états) limitées et clairement identifiées. Nous disons alors que ce sont des machines à états **finis**.



Les machines à états

Les machines à états

- ↳ Une machine à états est construite de quatre éléments de base
 - ▣ Des états
 - ▣ Des évènements
 - ▣ Des transitions
 - ▣ Des actions

États

- ↳ Un état est une situation stable qui possède une certaine durée pendant laquelle un objet exécute une activité ou attend un évènement
- ↳ Un état représenté par un cercle dans lequel le nom de l'état est inscrit



Les machines à états

Transitions

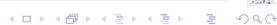
- Une transition définit la réponse d'un objet à l'arrivée d'un évènement. Elle indique qu'un objet qui se trouve dans un état peut « transiter » vers un autre état en exécutant éventuellement certaines activités
- Une transition est représentée par une flèche allant de l'état de départ vers l'état d'arrivée

Évènements

- Un évènement est un fait qui déclenche le changement d'état, qui fait donc passer un objet d'un état à un autre état
- Un évènement se produit à un instant précis et est dépourvu de durée. Quand un évènement est reçu, une transition peut être déclenchée et faire basculer l'objet dans un nouvel état.
- Un évènement est représenté par une expression inscrite sur une transition

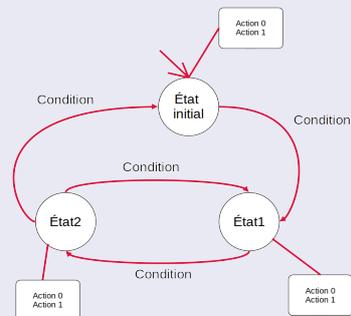
Actions

- Une action consiste à envoyer un signal, à faire appel à une méthode, à affecter une valeur à un attribut, etc.



Les machines à états

Représentation graphique d'une machine à états



Les machines à états

Traduction d'une machine à états en C++

- Définir un type énumératif


```
1 enum class State {etatInitial, etat1, etat2, ...};
```
- Définir une variable globale pour mémoriser l'état courant


```
1 State etat;
```
- L'implémentation de la machine états se fait par un switch en créant un case pour chaque état

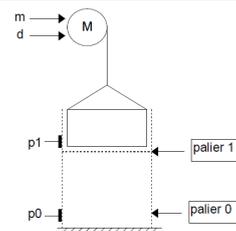

```
1 switch(etat) {
2 case State::etatInitial: // Actions de etatInitial
3   if (cond) { // Condition de transition
4     // Actions juste avant la transition
5     etat=etatSulvant;
6   }
7   break;
8 case State::etat0: ...
9 }
```



Les machines à états

Exercice

- On se propose de réaliser la commande d'une monte-charge. Le moteur est commandé par deux signaux m et d pour montée et descente. Cette commande élabore ces commandes à partir de ses entrées B , $p1$ et $p0$ suivant le cahier des charges suivant :
 - À l'initialisation, la monte-charge doit descendre au palier 0
 - Quand la cabine est entre deux paliers, la dernière commande (m ou d) est maintenue
 - Quand la cabine est à un palier, si B est inactif elle s'arrête, si B est actif, elle change de palier
 - L'entrée $p0$ (respectivement $p1$) indique, lorsqu'elle est active, que la monte charge est au palier 0 (respectivement palier 1).

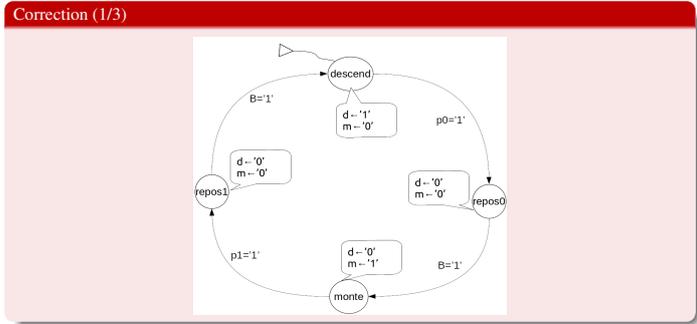


Les machines à états

Les machines à états

Exercice

- Donner la machine à états qui décrit la commande de monte-charge
- Donner le programme C/C++ implémentant la commande en adoptant les connexions suivantes : m GPIO14, d GPIO15, B GPIO17, p0 GPIO18 et p1 GPIO27



Les machines à états

Les machines à états

Correction (2/3)

```

1 #include <iostream>
2 #include <unistd.h>
3 #include <pigpio.h>
4
5 constexpr uint8_t m = 14;
6 constexpr uint8_t d = 15;
7 constexpr uint8_t B = 17;
8 constexpr uint8_t p0 = 18;
9 constexpr uint8_t p1 = 27;
10 enum class State {descend, repos0, monte, repos1};
11
12 State etat;
13 int main(int argc, char *argv[]) {
14     if (gpioInitialise() < 0) {
15         std::cout << "Echec_d'initialisation...\n";
16         exit(-1);
17     }
18
19     etat = State::descend;
20     gpioSetMode(m, PI_OUTPUT); gpioSetMode(d, PI_OUTPUT);
21     gpioSetMode(p0, PI_INPUT);
22     gpioSetMode(p1, PI_INPUT);
23     gpioSetMode(B, PI_INPUT);
    
```



Correction (3/3)

```

1 while (1) {
2     switch (etat) {
3     case State::descend:
4         gpioWrite(m, 0);
5         gpioWrite(d, 1);
6         if (gpioRead(p0)) etat = State::repos0;
7         break;
8     case State::repos0:
9         gpioWrite(m, 0);
10        gpioWrite(d, 0);
11        if (gpioRead(B)) etat = State::monte;
12        break;
13    case State::monte:
14        gpioWrite(m, 1);
15        gpioWrite(d, 0);
16        if (gpioRead(p1)) etat = State::repos1;
17        break;
18    case State::repos1:
19        gpioWrite(m, 0);
20        gpioWrite(d, 0);
21        if (gpioRead(B)) etat = State::descend;
22        break;
23    }
24 }
25 gpioTerminate();
26 return 0;
27 }
    
```



Plan

- 1 Entrées/Sorties GPIO
- 2 Programmation concurrentielle
- 3 Les interruptions
- 4 Programmation avec les machines à états
- 5 Les canaux Pulse Width Modulation (PWM)
- 6 Fonctions de communication
- 7 Communication UART
- 8 Communication sur un bus I2C (Inter Integrated Circuit)
- 9 Communication avec l'interface SPI (Serial Peripheral Interface)

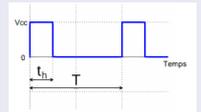
Les canaux Pulse Width Modulation (PWM)

Le signal Pulse Width Modulation (PWM)

↳ Un signal PWM ou Modulation par Largeur d'Impulsions (MLI) est un signal numérique périodique (de fréquence constante) et de rapport cyclique variable

• T désigne la période, la fréquence du signal est $f = 1/T$

• Le rapport cyclique (duty cycle) = t_h/T

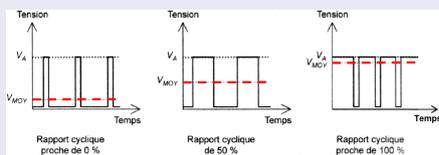


Les canaux Pulse Width Modulation (PWM)

Génération de PWM avec la RPi3

Intérêts

↳ Contrôler la puissance de la sortie



↳ Générer un signal pseudo-analogique

Connectiques des broches PWM

↳ Sur la RPi3, seulement la broche GPIO18, GPIO12 et GPIO13 peuvent générer des signaux PWM matériel (25 Mhz)

• GPIO18 et GPIO12 sont connectés au même canal PWM 0

• GPIO13 est connecté au canal PWM 1

• Ces deux canaux sont utilisées aussi pour la génération des signaux sonores

↳ Les autres broches peuvent aussi générer des signaux PWM logiciels (threads) ou en utilisant le DMA (Direct Memory Access)

• La fréquence maximale en utilisant les threads : 10 KHz

• La fréquence maximale en utilisant le DMA : 1 Mhz



Génération de PWM avec la RPi3

Génération de PWM avec pigpio : contrôler le rapport cyclique

- `int gpioPWM(unsigned user_gpio, unsigned dutycycle)` : génère un signal PWM sur une broche de sortie
 - `user_gpio` : 0-31
 - `dutycycle` : 0-RANGE; 0 pour rapport cyclique minimal et RANGE (255) pour rapport cyclique maximal
 - Retourne 0 si OK, sinon `PI_BAD_USER_GPIO` ou `PI_BAD_DUTYCYCLE`
- `int gpioGetPWMDutycycle(unsigned user_gpio)` : retourne le rapport cyclique pour une broche générant un signal PWM
 - `user_gpio` : 0-31
 - Retourne 0 si OK, sinon `PI_BAD_USER_GPIO` ou `PI_NOT_PWM_GPIO`



Génération de PWM avec la RPi3

Génération de PWM avec pigpio : contrôler la fréquence

- `int gpioSetPWmfrequency(unsigned user_gpio, unsigned frequency)` : définit la fréquence pour la génération d'un signal PWM sur une broche de sortie
 - `user_gpio` : 0-31
 - `frequency` : $0 \leq \text{frequency} \leq 1000000$ fréquence en hertz. La fréquence réellement utilisée dépend de taux d'échantillonnage
 - Retourne 0 si OK, sinon `PI_BAD_USER_GPIO`
- `int gpioGetPWmfrequency(unsigned user_gpio)` : retourne la fréquence réellement utilisée en hertz
 - `user_gpio` : 0-31
 - Retourne 0 si OK, sinon `PI_BAD_USER_GPIO`



Génération de PWM avec la RPi3

Exemple : allumer et éteindre progressivement une diode LED connectée à BCM17

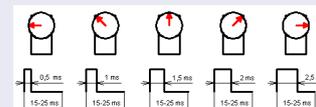
```
1 #include <iostream>
2 #include <unistd.h>
3 #include <pigpio.h>
4
5 constexpr uint8_t O_LED=17;
6
7 int main(int argc, char *argv[]) {
8     if (gpioInitialise() < 0) {
9         std::cout << "Failure...\n";
10        exit(-1);
11    }
12    gpioSetMode(O_LED, PI_OUTPUT);
13    while (1) {
14        for (uint16_t i=0; i<256; i++) {
15            gpioPWM(O_LED, i);
16            sleep(50);
17        }
18        for (uint16_t i=255; i>0; i--) {
19            gpioPWM(O_LED, i);
20            sleep(50);
21        }
22    }
23    gpioTerminate();
24    return 0;
25 }
```



Génération de PWM

Génération de PWM avec pigpio : contrôler un servomoteur

- Un servomoteur est un système qui a pour but de produire un mouvement précis en réponse à une commande externe.
- Un servomoteur est asservi en position angulaire à travers un signal PWM
- Le servomoteur SG90 9 g Micro Servo



Génération de PWM

Génération de PWM avec pigpio : contrôler un servomoteur

- ↳ Connexions du servomoteur SG90

PWM=Orange (┌┴┐)
Vcc=Red (+)
Ground=Brown (-)

- ↳ Montage : l'entrée PWM du servomoteur est connectée à BCM18



Génération de PWM pour contrôler un servomoteur

Génération de PWM avec pigpio : contrôler un servomoteur

- ↳ `int gpioServo(unsigned user_gpio, unsigned pulsewidth) :` spécifie la largeur de pulsation pour le contrôle d'un servomoteur
 - ↳ `user_gpio :` 0-31
 - ↳ `pulsewidth :` 0 (pas de signal), 500-2500
 - ↳ Retourne 0 si OK, sinon `PI_BAD_USER_GPIO`
 - ↳ Le signal PWM généré est de fréquence 50 Hz
- ↳ `int gpioGetServoPulsewidth(unsigned user_gpio) :` renvoie la largeur de pulsation d'un signal servo produit sur une sortie
 - ↳ `user_gpio :` 0-31
 - ↳ Retourne 0 si OK, sinon `PI_BAD_USER_GPIO`



Génération de PWM avec la RPi3

Exemple : contrôler un servomoteur

```
1 #include <iostream>
2 #include <unistd.h>
3 #include <pigpio.h>
4
5 int main(int argc, char *argv[]) {
6     if (gpioInitialise() < 0) {
7         std::cout << "Echec de l'initialisation\n";
8         exit(-1);
9     }
10    while (1) {
11        uint16_t position;
12        do {
13            std::cout << "Donner la position angulaire entre 0 et 180 : ";
14            std::cin >> position;
15        } while (position > 180);
16        uint16_t dureePulsation = 500 + (position * 2000) / 180;
17        gpioServo(18, dureePulsation);
18    }
19    gpioTerminate();
20    return 0;
21 }
```



Plan

- 1 Entrées/Sorties GPIO
- 2 Programmation concurrentielle
- 3 Les interruptions
- 4 Programmation avec les machines à états
- 5 Les canaux Pulse Width Modulation (PWM)
- 6 Fonctions de communication
- 7 Communication UART
- 8 Communication sur un bus I2C (Inter Integrated Circuit)
- 9 Communication avec l'interface SPI (Serial Peripheral Interface)



Bus

Définition d'un bus

- ☛ lignes de données : liaison bidirectionnelle qui assure le transfert des informations entre un élément et un autre
- ☛ lignes d'adresses : liaison bidirectionnelle qui permet la sélection des informations à traiter dans un emplacement mémoire.
- ☛ lignes de commandes : liaison pour assurer la synchronisation des flux d'informations sur les bus de données et d'adresses : l'horloge (« clock »), demandes d'interruption et d'accord (« acknowledge »), arbitrage des échanges, le contrôle des échanges (read/write, type de transfert, types des données, etc.)



Bus

Types de bus

- Bus série (i2c, uart, spi, usb, etc.) : permettent de relier entre un certain nombre de circuits en transmettant les données bit par bit
 - ✓ Lenteur (remède : augmenter la fréquence de transmission)
 - ✗ Résistant aux perturbations
 - ✗ Économique (moins de fils)
 - ✗ Adapté aux systèmes embarqués (moins d'espace)
- Bus parallèle (AGP, PCI, etc.) : est un ensemble de conducteurs électriques parallèles
 - ☛ À chaque cycle de temps, chaque conducteur transmet un bit
 - ☛ Les tailles les plus courantes (en bits) sont : 8, 16, 32, 64 ou plus
 - ✗ Couteux et encombrant
 - ✗ Moins résistant aux perturbations
 - ✓ Rapidité



Communication

Principe de communication

- Les communications se font par signaux
 - ☛ via un port : point de connexion
 - ☛ sur un bus : contient plusieurs câbles et se définit par le protocole (règles) de communication.
 - ☛ grâce à un contrôleur : les données échangées entre un périphérique et le processeur transitent par l'interface (contrôleur) associé à ce périphérique. L'interface possède de la mémoire tampon pour stocker les données échangées. Le contrôleur stocke aussi les informations nécessaires à la gestion de la communication



Communication

Caractéristiques

- Simplex/Duplex
 - ☛ Simplex : les données circulent dans un seul sens : émetteur vers récepteur (exemple : souris->ordinateur, ordinateur -> imprimante)
 - ☛ Half-duplex : les données circulent dans les 2 sens mais pas simultanément : la bande passante est utilisée en intégralité (aussi appelé alternat ou semi-duplex). Exemple : talkie/walkies, êtres humains (on ne coupe pas la parole).
 - ☛ Full-duplex : les données circulent de manière bidirectionnelle et simultanément : la bande passante est divisée par 2 pour chaque sens (duplex intégral).



Communication

Caractéristiques

- Rapidité : taux de transfert
- Il existe 2 unités pour qualifier la rapidité des échanges
 - ⚠ Bauds : nombre de bits de données transmis par seconde
 - ⚠ Bits/sec : nombres de bits (quelconques) transmis par seconde



Communication

Caractéristiques

- Synchrone/asynchrone
- Pour une liaison série, un seul fil transporte l'information
 - ⚠ Problème de synchronisation entre émetteur et récepteur (distinguer et reconnaître les séquences de bits utiles)
- ❶ Communication synchrone : l'émetteur et le récepteur sont cadencés à la même fréquence d'horloge
 - ⚠ Un signal horloge commun
- ❶ Communication asynchrone : la synchronisation est imposée par le protocole
 - ⚠ La transmission se fait caractère par caractère



Plan

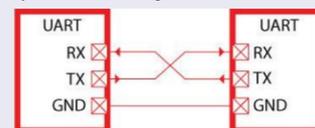
- 1 Entrées/Sorties GPIO
- 2 Programmation concurrentielle
- 3 Les interruptions
- 4 Programmation avec les machines à états
- 5 Les canaux Pulse Width Modulation (PWM)
- 6 Fonctions de communication
- 7 Communication UART**
- 8 Communication sur un bus I2C (Inter Integrated Circuit)
- 9 Communication avec l'interface SPI (Serial Peripheral Interface)



Principe de communication série avec UART

Principe de communication série avec UART

- UART (Universal Asynchronous Receiver Transmitter) est un contrôleur de communication série, asynchrone et full-duplex



- Une donnée est envoyée bit par bit
- L'envoi d'une donnée (7/8 bits) s'effectue sous forme d'une trame
- La vitesse de transmission est mesurée en bauds (bits de données par seconde) : 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, ou 230400



Principe de communication série avec UART

Principe de communication série avec UART

- L'envoi de données se fait octet par octet (mode caractère)
- Une trame UART pour envoyer un caractère



- ☞ Un bit de démarrage (start bit) dont le niveau logique est zéro
- ☞ Entre 7 et 8 bits de données
- ☞ Éventuellement un bit de parité paire ou impaire (parity)
- ☞ Un ou deux bits de stop avec un niveau logique de zéro (stop bits)



Principe de communication série avec UART

Avantages et inconvénients

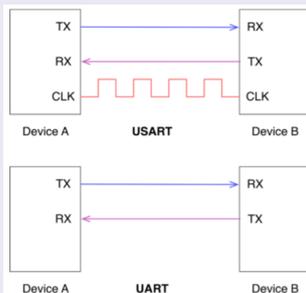
- Avantages
 - ✔ Standardisée, universelle
 - ✔ Pas chère : 3 fils suffisent (émission Tx, réception Rx, masse GND) et souvent l'alimentation + 5 V
- Inconvénients
 - ✘ Lenteur
 - ✘ Ne permet de relier que deux périphériques



Communication UART

Extensions du protocole UART

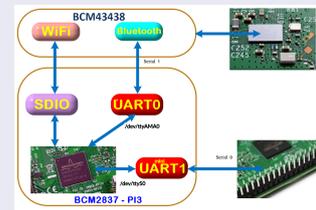
- USART (Universal Synchronous Asynchronous Receiver Transmitter)
 - ☞ On rend le protocole synchrone en ajoutant une ligne d'horloge, contrôlé par une des parties



Communication UART

Contrôleurs UART de la RPi

- Deux contrôleurs UART sont installés sur la RPi



Par défaut, le bluetooth utilise UART pour se connecter au SoC, alors que les pins GPIO8 et GPIO10 utilisent un mini-UART

- UART0 : un véritable contrôleur UART utilisé par défaut par le bluetooth
- UART1 : un mini-contrôleur UART dont le générateur de fréquence dépend du signal horloge du processeur
 - La fréquence du signal horloge du processeur est variable



Communication UART

Branchements

On utilisera un câble TTL/USB (Transistor-Transistor Logic / Universal Serial Bus)



Communication UART

Communication UART avec pigpio

`int serOpen(char *serTTY, unsigned baud, unsigned serFlags):` Ouvrir une communication série

`serTTY` : spécifie le périphérique série à ouvrir dont le nom doit commencer par `/dev/tty` ou `/dev/serial`

`baud` : le débit de transmissions en bauds : 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, ou 230400

`serFlags` : doit être à 0 (pas de drapeaux définis)

Retourne un `handle` ≥ 0 pour le périphérique dont l'adresse est spécifiée si OK, sinon `PI_NO_HANDLE` ou `PI_SER_OPEN_FAILED`

`int serClose(unsigned handle)` : termine la communication série avec le périphérique ayant le `handle` spécifié

`handle` : le `handle` ≥ 0 renvoyé par `serOpen()`

Retourne 0 si OK, sinon `PI_BAD_HANDLE`

Communication UART

Communication UART avec pigpio

Les fonctions d'envoi

`int serWriteByte(unsigned handle, unsigned bVal)` : envoyer un seul octet

`int serWrite(unsigned handle, char *buf, unsigned count)` : envoyer `count` octets spécifiés par le tableau `buf`

`handle` : le `handle` ≥ 0 renvoyé par `serOpen()`

Renvoie 0 si OK, sinon `PI_BAD_HANDLE`, `PI_BAD_PARAM` ou `PI_SER_WRITE_FAILED`

Communication UART

Communication UART avec pigpio

Les fonctions de réception

`int serReadByte(unsigned handle)` : lire un seul octet

`int serRead(unsigned handle, char *buf, unsigned count)` : lire `count` octets et les sauvegarder dans `buf`

`handle` : le `handle` ≥ 0 renvoyé par `serOpen()`

Renvoie 0 si OK, sinon `PI_BAD_HANDLE`, `PI_BAD_PARAM` ou `PI_SER_READ_NO_DATA`

`int serDataAvailable(unsigned handle)` : renvoie le nombre d'octets disponibles pour la réception

Plan

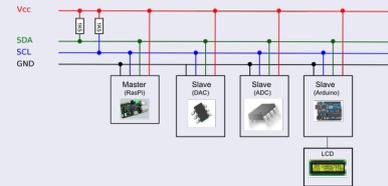
- 1 Entrées/Sorties GPIO
- 2 Programmation concurrentielle
- 3 Les interruptions
- 4 Programmation avec les machines à états
- 5 Les canaux Pulse Width Modulation (PWM)
- 6 Fonctions de communication
- 7 Communication UART
- 8 **Communication sur un bus I2C (Inter Integrated Circuit)**
- 9 Communication avec l'interface SPI (Serial Peripheral Interface)



Le bus I2C

Le bus I2C

- I2C est un bus permettant la communication synchrone en half-duplex
- Le protocole de communication 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.
- La communication n'a lieu qu'entre un seul maître et un seul esclave.
- Chaque esclave possède un identifiant unique (adresse). Cette adresse est fournie par l'esclave.



Communication sur I2C

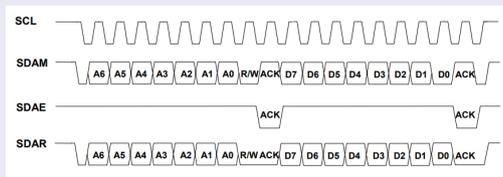
Principe de communication sur I2C

- 1 Le Maître envoie sur le bus l'adresse du composant avec qui il souhaite communiquer,
- 2 Chacun des esclaves ayant une adresse fixe, l'esclave qui se reconnaît répond à son tour par un signal de confirmation,
- 3 Puis le Maître continue la procédure de communication : écriture/lecture
- 4 Les transactions devraient être confirmées par un ACK (acquiescement)



Communication sur I2C

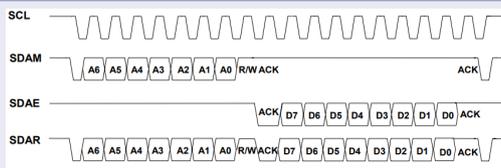
Envoi d'une donnée



Communication sur I2C

Le bus I2C

Lecture d'une donnée



Communication I2C avec pigpio

- `int i2cOpen(unsigned i2cBus, unsigned i2cAddr, unsigned i2cFlags)` : initialise la communication sur le bus I2C pour un périphérique
 - ☛ `i2cBus` : ≥ 0 ; spécifie le bus i2c à utiliser
 - ☛ `i2cAddr` : 0-0x7F; spécifie l'adresse du périphérique sur le bus i2c
 - ☛ `i2cFlags` : doit être à 0 (pas de drapeaux définis)
 - ☛ Retourne un handle ≥ 0 pour le périphérique dont l'adresse est spécifiée si OK, sinon `PI_BAD_I2C_BUS`, `PI_BAD_I2C_ADDR`, `PI_BAD_FLAGS`, `PI_NO_HANDLE` ou `PI_I2C_OPEN_FAILED`
- `int i2cClose(unsigned handle)` : termine la communication sur le bus i2c avec le périphérique ayant le handle spécifié
 - ☛ `handle` : le handle ≥ 0 renvoyé par `i2cOpen()`



Le bus I2C

Le bus I2C

Communication I2C avec pigpio

➤ Les fonctions d'envoi

- `int i2cWriteQuick(unsigned handle, unsigned bit)` : envoyer un seul bit
- `int i2cWriteByte(unsigned handle, unsigned bVal)` : envoyer un seul octet
- `int i2cWriteDevice(unsigned handle, char *buf, unsigned count)` : envoyer jusqu'à 32 octets
- `int i2cWriteByteData(unsigned handle, unsigned i2cReg, unsigned bVal)` : écrire un seul octet dans un registre
- `int i2cWriteWordData(unsigned handle, unsigned i2cReg, unsigned wVal)` : écrire un mot (16 bits) dans un registre
- `int i2cWriteI2CBlockData(unsigned handle, unsigned i2cReg, char *buf, unsigned count)` : écrire jusqu'à 32 octets dans un registre
- ☛ `handle` : le handle ≥ 0 renvoyé par `i2cOpen()`
- ☛ Renvoie 0 si OK, sinon `PI_BAD_HANDLE`, `PI_BAD_PARAM` ou `PI_I2C_WRITE_FAILED`

Communication I2C avec pigpio

➤ Les fonctions de réception

- `int i2cReadByte(unsigned handle)` : lire un seul octet
- `int i2cReadDevice(unsigned handle, char *buf, unsigned count)` : lire jusqu'à 32 octets
- `int i2cReadByteData(unsigned handle, unsigned i2cReg)` : lire un seul octet depuis un registre
- `int i2cReadWordData(unsigned handle, unsigned i2cReg)` : lire un mot (16 bits) depuis un registre
- `int i2cReadI2CBlockData(unsigned handle, unsigned i2cReg, char *buf, unsigned count)` : lire jusqu'à 32 octets depuis un registre
- ☛ `handle` : le handle ≥ 0 renvoyé par `i2cOpen()`
- ☛ Renvoie ≥ 0 si OK, sinon `PI_BAD_HANDLE`, `PI_BAD_PARAM` ou `PI_I2C_READ_FAILED`



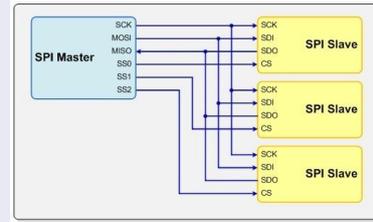
Plan

- 1 Entrées/Sorties GPIO
- 2 Programmation concurrentielle
- 3 Les interruptions
- 4 Programmation avec les machines à états
- 5 Les canaux Pulse Width Modulation (PWM)
- 6 Fonctions de communication
- 7 Communication UART
- 8 Communication sur un bus I2C (Inter Integrated Circuit)
- 9 Communication avec l'interface SPI (Serial Peripheral Interface)

L'interface SPI

L'interface SPI

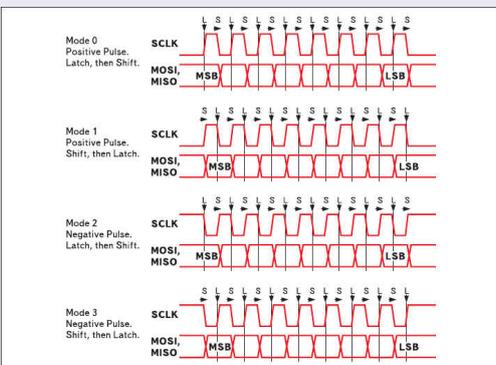
- SPI a été conçue par Motorola dans les années 1980
- SPI est une interface de communication en série synchrone et full-duplex
- SPI permet de mettre en communication un composant maître (généralement le microprocesseur) et plusieurs périphériques esclaves



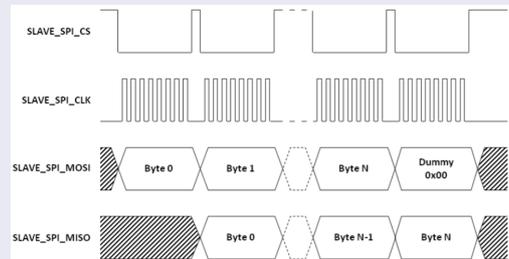
L'interface SPI

L'interface SPI

Modes de transmission avec SPI

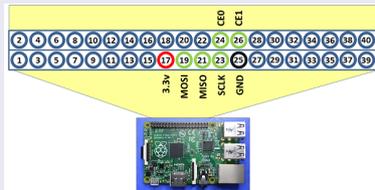


Transmissions des données



L'interface SPI sur la RPi3

Connexions SPI sur la RPi3



L'interface SPI sur la RPi3

Communication SPI avec pigpio

- int spiOpen(unsigned spiChan, unsigned baud, unsigned spiFlags) : initialiser la gestion de l'interface SPI
 - spiChan : 0-1 ; spécifier le canal. 2 pour un canal auxiliaire
 - baud : 32K-125M (valeur >30M ne fonctionnent pas)
 - spiFlags : les 22 bits du poids le plus faible spécifient les paramètres
 - Returns a handle (>=0) if OK, otherwise PI_BAD_SPI_CHANNEL, PI_BAD_SPI_SPEED, PI_BAD_FLAGS, PI_NO_AUX_SPI ou PI_SPI_OPEN_FAILED

Structure de spiFlags

```

21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
b b b b b R T n n n n n W A u2 u1 u0 p2 p1 p0 m m

```

1 mm : spécifie le mode de transmission SPI

Mode	POL	PHA
0	0	0
1	0	1
2	1	0
3	1	1

L'interface SPI sur la RPi3

Communication SPI avec pigpio

Structure de spiFlags

```

21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
b b b b b R T n n n n n W A u2 u1 u0 p2 p1 p0 m m

```

- px : est 0 si CE_x est actif à l'état bas (par défaut), et 1 pour l'état haut
- ux : est 0 si la broche CE_x est réservée pour SPI (par défaut), sinon 1
- A : est 0 pour le périphérique standard SPI, 1 pour le SPI auxiliaire
- W : is 0 if the device is not 3-wire, 1 if the device is 3-wire. Standard SPI device only.
- nnnn : defines the number of bytes (0-15) to write before switching the MOSI line to MISO to read data. This field is ignored if W is not set. Standard SPI device only.
- T : is 1 if the least significant bit is transmitted on MOSI first, the default (0) shifts the most significant bit out first. Auxiliary SPI device only.
- R : is 1 if the least significant bit is received on MISO first, the default (0) receives the most significant bit first. Auxiliary SPI device only.
- bbbbbb : defines the word size in bits (0-32). The default (0) sets 8 bits per word. Auxiliary SPI device only

MERCI POUR VOTRE ATTENTION



Questions ?