





Conception des objets connectés

Dr. Eng. Chiheb Ameer ABID

 /in/chiheb-ameur-abid
 chiheb.abid@gmail.com

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`

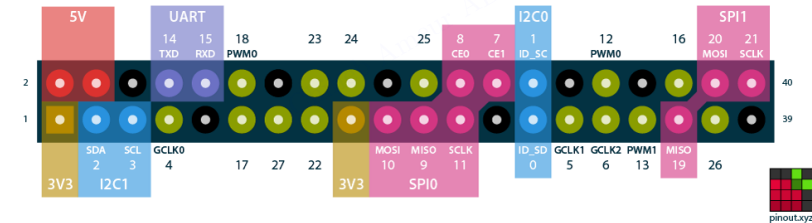
Plan

- 1 Entrées/Sorties GPIO
- 2 Programmation concurrentielle
- 3 Les interruptions

Présentation du port GPIO

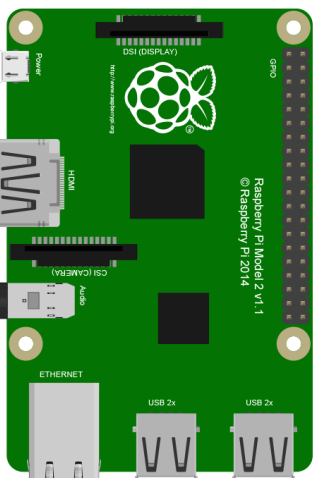
Brochage

Raspberry Pi GPIO BCM numbering



Présentation du port GPIO

Brochage



WiringPi	BCM(Name)	Physical	Physical	BCM(Name)	WiringPi
3v3 Power		1	17	5v Power	
8	BCM 2 (SDA)	3	5	5v Power	
9	BCM 3 (SCL)	7	9	Ground	
7	BCM 4 (GCLKO)	11	13	BCM 14 (TXD)	15
Ground		15	19	BCM 15 (RXD)	16
0	BCM 17	21	23	BCM 18 (PCM_C)	1
2	BCM 27 (PCM_D)	25	27	BCM 23	4
3	BCM 22	29	31	BCM 24	5
3v3 Power		33	35	Ground	
12	BCM 10 (MOSI)	37	39	Ground	
13	BCM 9 (MISO)	41	43	BCM 8 (CE0)	10
14	BCM 11 (SCLK)	45	47	BCM 7 (CE1)	11
Ground		49	51	BCM 1 (ID_SC)	
BCM 0 (ID_SD)		53	55	Ground	
21	BCM 5	57	59	BCM 12	26
22	BCM 6	61	63	Ground	
23	BCM 13	65	67	BCM 16	27
24	BCM 19 (MISO)	69	71	BCM 20 (MOSI)	28
25	BCM 26	73	75	BCM 21 (SCLK)	29
Ground		77	79		

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)

Présentation du port GPIO

Usage

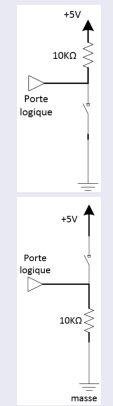
- Une sortie peut être fixée sur un niveau haut (3V3) ou bas (0V).
- Une entrée peut être lue comme un niveau haut (3V3) ou bas (0V).
- Toutes les broches ont des résistances internes pull-up ou pull-down
 - GPIO2 et GPIO3 ont des résistances de pull-up fixes
 - Les autres broches peuvent être configurées
- 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)

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

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

- Initialiser la bibliothèque

```
int gpioInitialise(void)
```

- ☞ Doit être appelée avant d'utiliser les autres fonctions de la bibliothèques
- ☞ Elle retourne la version de la bibliothèque si l'initialisation est effectuée correctement, sinon elle retourne PI_INIT_FAILED

- Terminer l'utilisation de la bibliothèque

```
void gpioTerminate(void)
```

- ☞ Libérer les ressources utilisées

Accès aux GPIOs avec pigpio

Fonctions pigpio

- Spécifier le mode d'accès à une broche

```
int gpioSetMode(unsigned gpio, unsigned mode)
```

- ☞ gpio : 0-53
- ☞ mode : 0-7; PI_INPUT, PI_OUTPUT, PI_ALT0, PI_ALT1, PI_ALT2, PI_ALT3, PI_ALT4 ou PI_ALT5

- Lire le mode d'accès d'une broche

```
int gpioGetMode(unsigned gpio)
```

- ☞ gpio : 0-53
- ☞ S'il y a une erreur, elle retourne PI_BAD_GPIO

- Définir une résistance pull-up ou pull-down à une broche

```
int gpioSetPullUpDown(unsigned gpio, unsigned pud)
```

- ☞ pud : 0-2; PI_PUD_OFF, PI_PUD_DOWN ou PI_PUD_UP

Accès aux GPIOs avec pigpio

Fonctions pigpio

- Spécifier l'état d'une broche de sortie

```
int gpioWrite(unsigned gpio, unsigned level)
```

- ☞ 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

- Lire l'état d'une broche

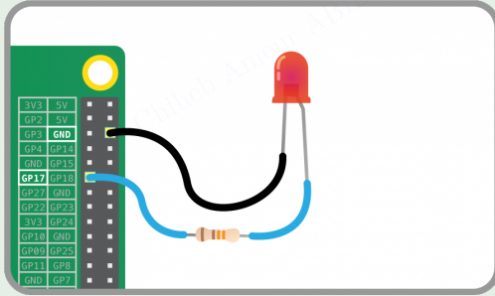
```
int gpioRead(unsigned gpio)
```

- ☞ gpio : 0-53
- ☞ S'il y a une erreur, elle retourne PI_BAD_GPIO

Accès aux GPIOs avec C et C++

Exemple : flasher une diode LED

➤ On considère le montage suivant :



Accès aux GPIOs avec C et C++

Accès aux GPIOs avec C et C++

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 auto main() -> int {
8     cout << "Running pigpio program" << endl;
9     gpioInitialise(); // you MUST initialize the library !
10    gpioSetMode(O_LED, PI_OUTPUT);
11    while (true) {
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 }

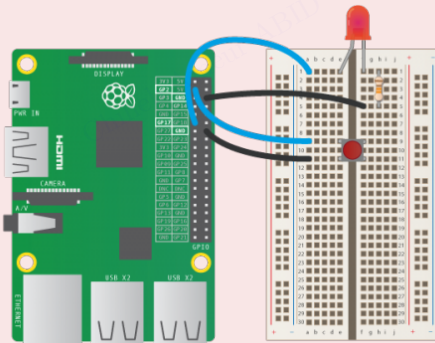
```



Accès aux GPIOs avec C et C++

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 auto main() -> int {
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 (true) {
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 }

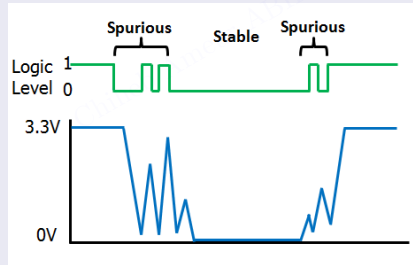
```



Accès aux GPIOs avec C et C++

Problème de parasitage des contacts

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



Accès aux GPIOs avec C et C++

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

- Entrées/Sorties GPIO
- Programmation concurrentielle
- Les interruptions

Programmation concurrentielle

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

Création d'un thread

- ↳ Un thread est créé à travers l'instanciation de la classe `std::thread` définie dans le fichier d'entête `<thread>`
- ⚠ Un objet de type `std::thread` ne peut pas être détruit avant la terminaison de son fil d'exécution
- 🔴 Pour attendre la fin de l'exécution d'un thread, on utilise la méthode `join()` de l'objet de type `std::thread()`

```

1 #include <iostream>
2 #include <thread>
3 auto hello() -> void
4 {
5     std::cout<<"Hello Concurrent World\n";
6 }
7 auto main() -> int
8 {
9     std::thread t {hello}; // Création du thread
10    // ... Traitements du thread principal
11    t.join(); // Attente de la fin du thread
12 }

```



Programmation concurrentielle

Section critique

- ↳ Lorsqu'un thread manipule une donnée (ressource) partagée avec d'autres, nous disons qu'il se trouve dans une section critique
- ↳ Une section critique est une 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



- ↳ Propriété d'exclusion mutuelle : garantir qu'un seul thread exécute sa section critique



Partage de données entre les threads

Partage des données

- ↳ Toute variable créée avant la création du thread et accessible depuis un ou plusieurs threads est une variable partagée
 - 🔴 Les variables globales sont des variables partagées
 - 🔴 Les variables locales à la fonction associée à un thread ne sont pas partagées

```

1 #include <iostream>
2 #include <thread>
3 int v_partagee {};
4 auto increment_counter() -> void {
5     int v_locale {};
6     ++v_locale;
7     ++v_partagee;
8     std::cout << " v_locale = " << v_locale << std::endl;
9     std::cout << " v_partagee = " << v_partagee << std::endl;
10 }
11 auto main() -> int{
12     std::thread t1{increment_counter};
13     std::thread t2{increment_counter};
14     t1.join();
15     t2.join();
16     return 0;
17 }

```



Les mutex

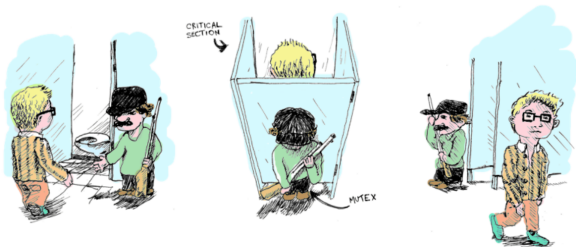
Coordination par les mutex

- ↳ 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



Utilisation d'un mutex

- ↳ L'utilisation typique d'un mutex pour assurer l'accès en exclusion mutuelle à une SC
 - Créer et initialiser une variable du mutex
 - Plusieurs threads tentent de verrouiller le mutex avant d'entrer dans la section critique
 - Seulement un thread réussit et prend le mutex, les autres sont bloqués ⇒ le mutex est verrouillé
 - Le thread propriétaire du mutex effectue un ensemble d'actions dans la section critique
 - Le thread propriétaire déverrouille le mutex en quittant la section critique
 - Un seul parmi les threads bloqués est réveillé, et il prend le contrôle mutex en répétant le processus



Les types atomiques standards

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
 - Implémentés ou non à l'aide des mutex

Programmation concurrentielle

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 auto add_to_list(int new_value) -> void // Thread 1
7 {
8     std::lock_guard guard {some_mutex};
9     some_list.push_back(new_value);
10 }
11 auto list_contains(int value_to_find) -> bool // Thread 2
12 {
13     std::lock_guard guard{some_mutex}; // Protéger la section critique
14     return std::find(some_list.begin(), some_list.end(), value_to_find) != some_list.end();
15 }

```



Le mutex est libéré à la destruction de sa garde

Les types atomiques standards

Les types atomiques standards

- ↳ Ces types peuvent être non lock-free

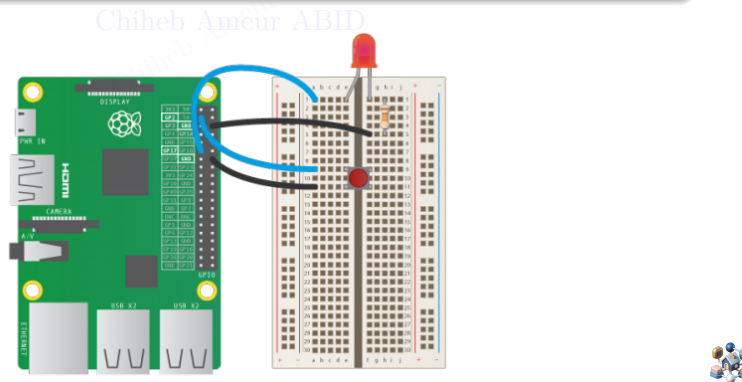
Type atomique	Spécialisation std::atomic<>
atomic_bool	std::atomic<bool>
atomic_char	std::atomic<char>
atomic_schar	std::atomic<signed char>
atomic_uchar	std::atomic<unsigned char>
atomic_int	std::atomic<int>
atomic_uint	std::atomic<unsigned>
atomic_short	std::atomic<short>
atomic_ushort	std::atomic<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 :

- 1 Un premier thread flashe une diode LED connectée à BCM17 à chaque seconde
- 2 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 auto main() -> int {
2   if (gpioInitialise() < 0) {
3     std::cout << "Echec d'initialisation...\n";
4     exit(-1);
5   }
6   gpioSetMode(0_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 (true);
12  gpioTerminate();
13  return 0;
14 }

```

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 not to optimize
9
10 auto flash() -> void { // Flash the LED
11   while (true)
12     if (flashEnabled) {
13       gpioWrite(O_LED, 1);
14       sleep(1);
15       gpioWrite(O_LED, 0);
16       sleep(1);
17     }
18 }
19
20 auto readButtonState() -> void { // Check whether the Button pressed
21   while (true)
22     if (gpioRead(I_BUTTON)==0) {
23       while (gpioRead(I_BUTTON)==0);
24       flashEnabled=!flashEnabled;
25     }
26 }

```

Plan

- 1 Entrées/Sorties GPIO
- 2 Programmation concurrentielle
- 3 Les interruptions

L'intérêt des interruptions

Vérifier une condition/évènement ?

Il existe trois approches pour la vérification d'un changement d'état sur une entrée

- 1 Attente active
- 2 Par scrutation (polling)
- 3 Interruption

L'intérêt des interruptions

Vérifier qu'une broche d'entrée change de 0 à 1!?

1 Attente active

```
1 while (true) {  
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
- ❌ Les autres évènements pendant l'attente sont ignorés

L'intérêt des interruptions

Interruptions

Vérifier qu'une broche d'entrée change de 0 à 1!?

2 Lecture par scrutation (Polling)

```
1 while (true) {  
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 d'ignorer d'autres évènements
- ❌ Consomme du temps processeur

Vérifier qu'une broche d'entrée change de 0 à 1!?

3 Les interruptions

```
1 /**  
2  * Programme principal  
3  **/  
4  while (true) {  
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

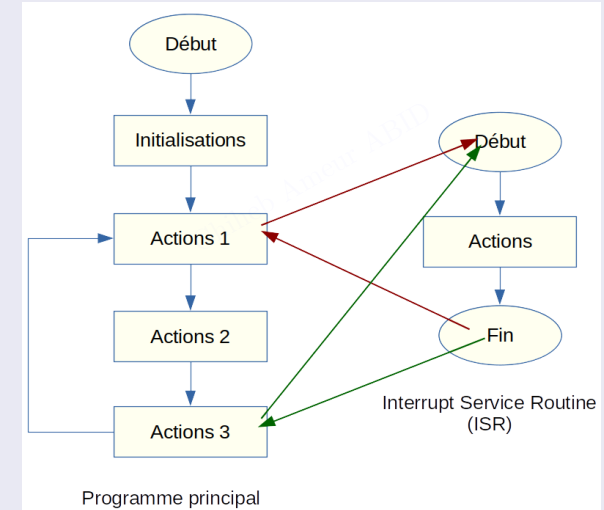
- ↳ Libérer le processeur en confiant la surveillance au contrôleur d'interruptions
- ↳ Lorsque un changement provient sur une entrée ou suite à un événement, le processeur est avisé à travers une demande d'interruption (Interrupt ReQuest) afin d'interrompre son travail actuel et exécuter le traitement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



↳ Raspberry OS n'est pas un système temps réel

Interruptions

Gestion des interruptions



Interruptions et GPIO

Gestion des interruptions avec pigpio

- ↳ Spécifier une fonction (callback) appelée suite à un changement d'état (demande d'interruption) d'une broche

```
auto gpioSetAlertFunc(unsigned user_gpio, gpioAlertFunc_t f) -> int;
```

- user_gpio : 0-31
- f : la routine de gestion d'interruption. f doit être écrite selon le type `void (*gpioAlertFunc_t)(int gpio, int level, uint32_t tick);`
- Retourne 0 si OK, sinon `PI_BAD_USER_GPIO`



- ↳ 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 auto cbPushButton(int gpio, int level, uint32_t tick) -> void {
9     if (level==0) gpioWrite(O_LED,!gpioRead(O_LED));
10 }
11 auto main() -> int {
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 (true) { }
22     gpioTerminate();
23     return 0;
24 }
  
```

Interruptions et GPIO

Gestion des interruptions avec pigpio

- Spécifier l'ISR relative à une interruption produite par une broche en précisant sur quel front l'interruption est générée

```
auto gpioSetISRFunc(unsigned user_gpio, unsigned edge, int timeout, gpioISRFunc_t f) -> int;
```

- 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 auto cbPushButton(int gpio, int level, uint32_t tick) -> void {
9     gpioWrite(O_LED, !gpioRead(O_LED));
10 }
11 auto main() -> int {
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 (true) { }
22     gpioTerminate();
23     return 0;
24 }
```



Merci pour votre attention



Questions?

