Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année

Systèmes Embarqués

Corrections TP n°1

Programmation avec Arduino

Broches GPIO

1 – Récupérez le « *schematic* » de la carte de développement, « *devboard* », de Muse Lab sur
`https://github.com/wuxx/nanoESP32-C3/blob/master/schematic/nanoESP32C3-v1.0.pdf`

a. Trouvez la broche correspondant à :
- la « guirlande » de LEDs RGB, de type WS2812 : *d'après le schematic, c'est la broche 14* ;
- le bouton "BOOT" : *broche 0* ;

b. D'après le « *schematic* », le bouton est-il :
- « *active low* », c-à-d quand on presse le bouton l'état logique est 0 ?
- « *active high* », c-à-d quand on presse le bouton l'état logique est 1 ?

D'après le schematic, le bouton connecte à la masse ou au « ground », il est donc « active low ».
Au niveau logique, on lira la valeur zéro lorsque le bouton est pressé et la valeur un quand il est relâché.

2 – a. Vous testerez le programme suivant :

```
const int buttonPin = 0;  // the pin of the pushbutton pin
int buttonState = 0;  // variable for reading the pushbutton status

void setup() {
  pinMode(buttonPin, INPUT);
}

void loop() {
  buttonState = digitalRead(buttonPin);

  if (buttonState == LOW) {
    printf("Low\n");
  }
}
```

Que fait-il ?
il affiche de nombreux « Low » mais finit par s'arrêter.
Si vous changez l'initialisation de la broche de `INPUT` à INPUT_PULLUP, que se passe-t-il ?
le comportement est similaire.
Expliquez le comportement.
Sur le schematic, on voit que le concepteur de la « board » a ajouté une résistance externe de Pullup, donc activer le pullup interne au microcontrôleur ne change rien.
Une pullup ramène la broche au niveau haut alors que le bouton l'amène au niveau bas.

b. Modifiez le programme précédent pour qu'il n'affiche qu'**une seule fois** le message lorsque l'on appuie sur le bouton, c-à-d que si on maintient appuyé le bouton, un seul affichage se fait.

```
const int buttonPin = 0;  // the number of the pushbutton pin
int buttonState = 0;  // variable for reading the pushbutton status
int afficher_passage_a_low = false;

void setup() {
  pinMode(buttonPin, INPUT_PULLUP);
  afficher_passage_a_low = false;
}

void loop() {
  buttonState = digitalRead(buttonPin);

  if ((buttonState == LOW) &&(afficher_passage_a_low == false )) {
    afficher_passage_a_low = true;
    printf("Low\n");
  }
  if (buttonState == HIGH)
  {
    afficher_passage_a_low = false;
  }
}
```

Est-ce que cela marche comme vous le voulez ?
Non, il peut en afficher plus d'un si on appui un peu rapidement sur le bouton.
Pourquoi ?
À cause du rebond mécanique.
Est-ce que votre programme détecte un changement d'état sur la broche associée au bouton ?
Non, il lit au moment où le CPU le choisit.

c. Dans les exemples fournis par l'IDE, chargez « *File>Examples>02.Digital>Debounce* ».

Expliquez ce qu'il fait ?
Il introduit un certain délai avant de valider le changement d'état du bouton.

Qu'est-ce que le « *debounce* » d'un bouton ?
Un procédé qui permet d'annuler les effets de rebond mécanique du bouton : par exemple lorsque l'on appui sur le bouton, la pression mécanique peut faire un contact, puis le rebond libère temporarirement ce contact pour le rétablir ensuite définitivement.
Ce rebond est très court, inférieur à $10ms$*, ce qui plus rapide que la perception humaine mais suffisamment long pour simuler l'appui rapide et répété du bouton du point de vue du CPU.*

Modifiez le **programme précédent** pour intégrer le « *debounce* ».

```
int buttonPin = 0;  // the number of the pushbutton pin
int buttonState = 0;  // variable for reading the pushbutton status
int lastButtonState = HIGH;
int lastDebounceTime = 0;
int lecture = HIGH;
int afficher_passage_a_low = false;

void setup() {
  pinMode(buttonPin, INPUT_PULLUP);
  afficher_passage_a_low = false;
}

void loop() {
  lecture = digitalRead(buttonPin);

  if (lecture != lastButtonState)
  {
    lastDebounceTime = millis();
  }

  if ((millis() - lastDebounceTime) > 50)
  {
    if (lecture != buttonState)
    {
      buttonState = lecture;
      if(buttonState == LOW)
      {
        afficher_passage_a_low = true;
      }
    }
  }
  if (afficher_passage_a_low == true ) {
    afficher_passage_a_low = false;
    printf("Low\n");
  }
  lastButtonState = lecture;
}
```

IRQ

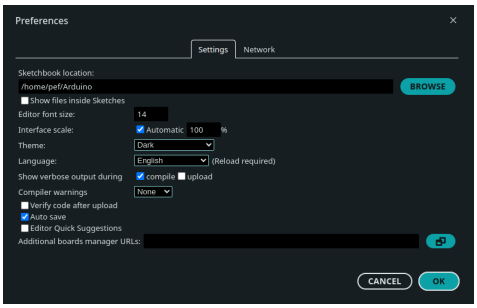
3 – Vous essaierez le programme suivant :

```
struct Button {
    uint8_t PIN;
    uint32_t numberKeyPresses;
    bool pressed;
};

Button button1 = {0, 0, false};

void IRAM_ATTR mon_isr(void* arg) {
    Button* s = (struct Button *) arg;
    s->numberKeyPresses += 1;
    s->pressed = true;
}

void setup() {
    pinMode(button1.PIN, INPUT_PULLUP);
    attachInterruptArg(digitalPinToInterrupt(button1.PIN), mon_isr, &button1, RI
SING);
}

void loop() {
    if (button1.pressed) {
        printf("Button 1 has been pressed %u times\n", button1.numberKeyPresses);
        button1.pressed = false;
    }
}
```

a. Pourquoi met-on `RISING` et non pas `FALLING` ?
Parce que le bouton est «active low» : l'état passe de haut à bas à l'appui, puis de bas à haut lorsqu'on le relâche.
Ici, on détecte le moment où il est relâché.

b. Que se passe-t-il si vous déplacez l'affichage du compteur d'appui dans la fonction `mon_isr` ?
Le programme peut planter si on appuie de manière rapide et répétée.
Pourquoi ?
Parce que l'interruption se déclenche et accède de nouveau au port série, avant que l'interruption précédente ait, elle-même fini d'y accéder, créant ainsi une corruption.

4 – On va regarder le placement de la fonction de gestion d'IRQ en mémoire.
Configurez dans les préférences «Show verbose output» :



Ainsi, après la compilation on devrait lire quelque chose de similaire à :

```
esptool v5.1.0
Wrote 0x400000 bytes to file '/home/pef/.cache/arduino/sketches/EE680B36CC6830B539E7D63362875653/sketch_nov13b.ino.merged.bin', ready to flash to offset 0x0.
/home/pef/.arduino15/packages/esp32/tools/esp-x32/2507/bin/xtensa-esp32s3-elf-size -A /home/pef/.cache/arduino/sketches/EE680B36CC6830B539E7D63362875653/sketch_nov13b.ino.elf
Sketch uses 275539 bytes (21%) of program storage space. Maximum is 1310720 bytes.
Global variables use 20840 bytes (6%) of dynamic memory, leaving 306840 bytes for local variables. Maximum is 327680 bytes.
```

On obtient le chemin d'accès au fichier «.elf», ici :
`/home/pef/.cache/arduino/sketches/EE680B36CC6830B539E7D63362875653/sketch_nov13b.ino.elf`

On va maintenant chercher l'adresse de la fonction «`mon_isr`» dans le «*firmware*» :

xterm

```
$ nm -C /home/pef/.cache/arduino/sketches/EE680B36.../sketch_nov13b.ino.elf |
grep mon_isr
4037799c T mon_isr(void*)
```

On obtient l'adresse de la fonction `mon_isr`
Si on supprime dans le code l'attribut de définition de la fonction :

```
void IRAM_ATTR mon_isr(void* arg) {
```

```
void mon_isr(void* arg) {
```

xterm

```
nm -C /home/pef/.cache/arduino/sketches/EE680B36.../sketch_nov13b.ino.elf | grep mon_isr
4201cba8 T mon_isr(void*)
```

Dans le fichier

`/.arduino15/packages/esp32/tools/esp32-arduino-libs/idf-release_v5.5-f1a1df9b-v3/esp32s3/ld/memory.ld`

on trouve la configuration du linker :

```
MEMORY
{
  /**
   *  All these values assume the flash cache is on, and have the blocks this uses subtracted from the length
   *  of the various regions. The 'data access port' dram/drom regions map to the same iram/irom regions but
   *  are connected to the data port of the CPU and eg allow byte-wise access.
   */
  /* IRAM for PRO CPU. */
  iram0_0_seg (RX) : org = (0x40370000 + 0x4000), len = (((0x403CB700 - (0x40378000 - 0x3FC88000)) - 0x3FC88000) + 0x8000 - 0x4000)
  /* Flash mapped instruction data */
  iram0_2_seg (RX) : org = 0x42000020, len = 0x800000-0x20
```

5 – Dans quelle zone mémoire se trouve la fonction `mon_isr` :

a. avec l'attribut « `IRAM_ATTR` » ?
4037799c -> iram0_0_seg

b. sans l'attribut « `IRAM_ATTR` » ?
4201cba8 -> iram0_2_seg

Expliquez pourquoi ?

Parce que l'attribut « IRAM_ATTR *» permet de localiser la fonction traitant l'IRQ en RAM.*
La LMA, « Load Memory Address », en flash, est différente de la VMA « Virtual Memory Address », en RAM.
L'accès aux instructions de traitement de l'IRQ sont chargées plus rapidement depuis la RAM que depuis la flash (accédée souvent par une interface série comme le bus SPI)⟹le traitement de l'IRQ est plus rapide.
Il est nécessaire au démarrage du microcontrôleur de copier les instructions de la flash vers la ram.

6 – a. Comment fonctionne une LED RGB WS2812 ?

`https://www.sdiplight.com/what-is-ws2812b-led-and-how-to-use-ws2812b-led/`

Elle dispose d'un microcontrôleur auquel on envoie l'information nécessaire à son état.
Vous installerez la bibliothèque permettant de piloter les LEDs WS2812 de Adafruit :

```
#define RGB_BRIGHTNESS 10 // ATTENTION ne pas dépasser sous peine de détruire l'ESP32
#define PIN_NEOPIXEL 14
void setup() {
  // No need to initialize the RGB LED
}
// the loop function runs over and over again forever
void loop() {
  neopixelWrite(PIN_NEOPIXEL, RGB_BRIGHTNESS, 0, 0);  // Red
  delay(1000);
  neopixelWrite(PIN_NEOPIXEL, 0, RGB_BRIGHTNESS, 0);  // Green
  delay(1000);
  neopixelWrite(PIN_NEOPIXEL, 0, 0, RGB_BRIGHTNESS);  // Blue
  delay(1000);
  neopixelWrite(PIN_NEOPIXEL, 0, 0, 0);  // Off / black
  delay(1000);
}
```

b. Vous combinerez ce programme avec le précédent pour bloquer l'arc-en-ciel lorsque l'on appui sur un bouton avec un traitement par IRQ.

```
#define RGB_BRIGHTNESS 10
#define PIN_NEOPIXEL 14

struct Button {
    uint8_t PIN;
    bool pressed;
};

Button button1 = {0, false};
bool activate = true;

void IRAM_ATTR mon_isr(void* arg) {
    Button* s = (struct Button *) arg;
```

```
    s->pressed = !digitalRead(s->PIN);
}

// the loop function runs over and over again forever
void loop() {
  if (!button1.pressed) {
    neopixelWrite(PIN_NEOPIXEL, RGB_BRIGHTNESS, 0, 0);  // Red
    delay(1000);
    neopixelWrite(PIN_NEOPIXEL, 0, RGB_BRIGHTNESS, 0);  // Green
    delay(1000);
    neopixelWrite(PIN_NEOPIXEL, 0, 0, RGB_BRIGHTNESS);  // Blue
    delay(1000);
    neopixelWrite(PIN_NEOPIXEL, 0, 0, 0);  // Off / black
    delay(1000);
  }
}

void setup() {
    pinMode(button1.PIN, INPUT_PULLUP);
    attachInterruptArg(digitalPinToInterrupt(button1.PIN), mon_isr, &button1,
CHANGE);
}
```

On utilise l'état `CHANGE`, pour déclencher l'interruption au moment de l'appui, mais aussi du relâchement du bouton.
On affecte la négation de la lecture du niveau sur la broche à l'état du bouton car il est « active low ».

7 – Que fait le programme suivant :

```
#include <Adafruit_NeoPixel.h>

#define RGB_Control_PIN   14
#define Matrix_Row        8
#define Matrix_Col        8
#define RGB_COUNT         64

uint8_t Matrix_Data[8][8];
Adafruit_NeoPixel pixels(RGB_COUNT, RGB_Control_PIN, NEO_RGB + NEO_KHZ800);

void Matrix_Init() {
  pixels.begin();
  pixels.setBrightness(10); // ATTENTION Ne pas dépasser cette valeur !!!
  memset(Matrix_Data, 1, sizeof(Matrix_Data));
}
void Matrix_zone(int zone, int color[3])
{
  int row_min = 0, row_max = 0;
  int col_min = 0, col_max = 0;
  switch(zone){
    case 0: row_min = 0; row_max = 3; col_min = 0; col_max = 3; break;
    case 1: row_min = 4; row_max = 7; col_min = 0; col_max = 3; break;
    case 2: row_min = 0; row_max = 3; col_min = 4; col_max = 7; break;
    case 3: row_min = 4; row_max = 7; col_min = 4; col_max = 7; break;
  }
  for (int row = row_min; row <= row_max; row++) {
    for (int col = col_min; col <= col_max; col++) {
      pixels.setPixelColor(row*8+col, pixels.Color(color[0],color[1],color[2]));
    }
  }
  pixels.show();
}
void setup()
{
  Matrix_Init();
}
//int x=0;
typedef enum {
    RED, GREEN, BLUE, BLACK, YELLOW, COLOR_COUNT
} Color;

int RGB[COLOR_COUNT][3] = {
    [RED]   = {255, 0,   0},
    [GREEN] = {0,   255, 0},
    [BLUE]  = {0,   0,   255},
    [BLACK] = {0,   0,   0},
    [YELLOW] = {255, 165, 0}
};

void loop()
{
  static int zone_choisie = 0;
  static int couleur_choisie = 0;
```

```
  Matrix_zone(zone_choisie, RGB[couleur_choisie]);
  delay(300);
  zone_choisie = (zone_choisie + 1) % 4;
  couleur_choisie = (couleur_choisie + 1 ) % COLOR_COUNT;
}
```

Il découpe la matrice de LEDs en 4 zones et fait alterner les couleurs de chaque zone.

Timers

8 – Vous essaierez le code suivant :

```
hw_timer_t *timer = NULL;
bool tic = false;
int caracteres = 0;

void ARDUINO_ISR_ATTR toc() {
    tic = true;
}

void setup() {
  Serial.begin(115200);
  while (!Serial) { delay(10); }
  timer = timerBegin(1000000);
  timerAttachInterrupt(timer, &toc);
  timerAlarm(timer, 1000000, true, 0);
}

void loop() {
    if (tic == true)
  {
    Serial.printf("*");
    tic = false;
    if (++caracteres == 10)
    {
      caracteres = 0;
      Serial.println();
    }
  }
}
```

a. À quelle vitesse s'effectue l'affichage ?
Toutes les secondes.
En quelle unité est défini le « *timer* » ?
En μs soit $10^{-6}s$

b. À l'aide de la fonction `millis()` affichez le temps mesurez entre chaque « *tick* » du timer.

```
#include "esp_system.h"

hw_timer_t *timer = NULL;
bool tick = false;
int caracteres = 0;
volatile unsigned int lastTimer = 0;
volatile unsigned int currentTimer = 0;
void ARDUINO_ISR_ATTR resetModule() {
  currentTimer = millis();
  tick = true;
}

void setup() {
  Serial.begin(115200);
  timer = timerBegin(1000000);
  timerAttachInterrupt(timer, &resetModule);  //attach callback
  timerAlarm(timer, 1000000, true, 0);  //set time in us
  currentTimer = millis();              //enable interrupt
}

void loop() {
  if (tick == true)
  {
    Serial.printf("Duree : %d\n",currentTimer-lastTimer);
    lastTimer = currentTimer;
    tick = false;
  }
}
```

c. Que se passe-t-il :

- si vous essayez de diminuer le temps de déclenchement du timer ? ***Attention*** *: on ne change que la valeur dans la fonction* `timerAlarm`.
  Arrivé à la valeur 10, en divisant par 10 successivement, on continue à avoir un affichage avec une durée de 0.
  Arrivé à 1, il n'y a plus d'affichage du tout... Les interruptions se produisent trop souvent et l'ESP32 plante.
- si vous mettez le calcul et l'affichage de la durée dans la fonction `tick` ?
  Si On reste à la valeur 1, l'ESP32 n'arrête pas de planter.
  Si on revient à 10, l'ESP32 continue de planter, et il faut passer à 100 pour obtenir de nouveau un affichage.

- Soit le code suivant :

```
portMUX_TYPE m = portMUX_INITIALIZER_UNLOCKED;

void ❶IRAM_ATTR toc()
{
portENTER_CRITICAL❷ISR(&m);
...
portEXIT_CRITICAL_ISR(&m);
}
```

Que fait le code ?
Il fait un mutex.
À quoi sert ❶ ?
à mettre le code dans la ram et pas dans la flash.
Qu'est-ce que veut dire ❷ ?
Que le mutex peut intervenir dans le code de gestion d'une interruption.
Reprenez votre code pour intégrer cette opération.

```
hw_timer_t *timer = NULL;
bool tick = false;
int caracteres = 0;
volatile unsigned int lastTimer = 0;
volatile unsigned int currentTimer = 0;
portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED;

void IRAM_ATTR resetModule() {
    portENTER_CRITICAL_ISR(&timerMux);
    currentTimer = millis();
    tick = true;
    //Serial.printf("Duree : %d\n",currentTimer-lastTimer);
    lastTimer = currentTimer;
    portEXIT_CRITICAL_ISR(&timerMux);
}

void setup() {
  Serial.begin(115200);
  while (!Serial) { delay(10); }
  timer = timerBegin(1000000);
  timerAttachInterrupt(timer, &resetModule);
  timerAlarm(timer, 10, true, 0);
  currentTimer = millis();
}

void loop() {
    if (tick == true)
  {
    Serial.printf("Duree : %d\n",currentTimer-lastTimer);
    lastTimer = currentTimer;
    tick = false;
  }
}
```

Si vous n'arrivez pas à programmer votre ESP32

1. Maintenir appuyé le bouton "BOOT" ;
2. Appuyer le bouton "RST" ;
3. Relâcher le bouton "BOOT" ;

`https://docs.espressif.com/projects/esptool/en/latest/esp32c3/advanced-topics/boot-mode-selection.html`