Programmation avancée

Les bases de la programmation arduino avec esp8266/esp32

Détails
Écrit par stéphane Domas
Catégorie : Programmation avancée (/index.php/menu-lpsil/objets-connectes)
Publication : 18 novembre 2009
Affichages : 1987

Préambule

La documentation sur les bibliothèques des composants natifs de l'esp32/8266 (PWM, WiFi, I2C, ...) , utilisables sous arduino sont accessibles via :

- pour l'esp32 : esp32 arduino docs (https://docs.espressif.com/projects/arduino-esp32/en/latest/index.html)
- pour l'esp8266 : esp8266 arduino docs (https://arduino-esp8266.readthedocs.io/en/latest/index.html#)

1°/ Structuration d'un sketch (= programme)

- un programme arduino est appelé un sketch.
- Il contient au moins 2 fonctions :
  - `void setup()` : initialise les variables globales, les communications en protocole série, senseurs, connexion wifi, ... Cette fonction est exécutée une seule fois après le boot,
  - `void loop()` : le point d'entrée d'exécution (= le main() d'un programme). A la fin de son exécution, le µC l'exécute de nouveau. loop() est donc appelée cycliquement, jusqu'à ce que le µC crashe, reboote, s'arrête, ...
- Le sketch peut bien entendu contenir d'autres fonctions.
- Comme dans un programme C, les variables peuvent être locales aux fonctions, ou bien globales à tout le sketch.
- Généralement, les variables globales sont déclarées au début du sketch, après les `#include` et `#define`.

2°/ Lire/écrire sur les pins GPIO

- le package (= enveloppe plastique) d'un µC expose des pins de connexion qui sont utilisées pour le connecter aux autres composants d'un circuit.
- Une partie de ces pins ont une fonction bien précise alors que d'autres peuvent être librement utilisées par un utilisateur pour être connectées à des circuits tels que des senseurs, des LEDs, des interrupteurs, etc.
- Ces pins sont appelées GPIO : Global Purpose Input/Output.
- Un µC n'étant quasi jamais utilisé directement mais intégré sur une carte de développement, seules une partie de ces GPIO sont réellement utilisables par le développeur.
- Par exemple, l'esp8266 a 33 pins au total, et 17 GPIOs. Mais seules 11 sont réellement utilisables puisque 6 sont toujours utilisées pour connecter le µC à une mémoire flash contenant le micro-code a exécuter.
- De plus, selon les constructeurs de la carte de développement, il peut y avoir encore moins que 11 GPIOs utilisables. Par exemple, les pins 1 et 3 sont généralement déjà connectées au convertisseur serie-usb qui permet à l'µC de communiquer avec un ordinateur au travers d'un cable USB.
- Enfin, même si les cartes de développement ont toutes une structure physique avec les GPIOs accessibles sur le pourtour de la carte, l'emplacement d'une GPIO précise peut varier d'une carte à une autre.
- En conclusion, pour un même projet, il peut être compliqué de l'adapter à différentes cartes de développement même si elles sont basées sur le même µC. C'est encore plus compliqué si on change de µC.

- Parmi les GPIOs, il existe des pins digitales et analogiques :
  - digital : permet de lire/écrire un signal numérique, c.a.d. un signal à 2 états : HIGH/LOW signal. HIGH correspond au voltage le plus haut (3.3V sur un esp8266, 5V sur un ATmega328), et LOW à 0V.

    - analogique : permet de lire/écrire un signal qui varie dans le temps, entre les niveaux haut et bas. Même si le signal physique varie de façon continue, ce n'est pas le cas du point de vue µC puisque ce dernier discrétise le signal afin d'obtenir une valeur numérique. La plage de variation acceptable, le nombre de pas, la vitesse d'acquisiation, ... dépendent du µC. Par exemple, un esp8266 accepte un signal variant entre 0 et 1V, et discrétise sur 1024 pas. (NB : 4096 pas pour un esp32)

- ATTENTION ! Une GPIO digitale peut être utilisée aussi bien en lecture qu'écriture, alors qu'en analogique, un seul sens est possible (pour des raisons électroniques)
- Sur une carte de développement basée sur un esp8266, il y a généralement une seule entrée analogique et aucune sortie, alors qu'avec un esp32, il y a généralement 3-4 entrées et 2 sorties analogiques.

- Pour simplifier la programmation des µC, les sketch n'utilise pas le nom technique ou bien le n° de pin physique du package du µC. Par exemple, la GPIO5 de l'esp8266 est la pin n° 24.
- On utilise plutôt des surnoms ou des numéros qui sont les mêmes quel que soit le constructeur de la carte de développement. ATTENTION, ces surnoms n'ont pas forcément de rapport avec le nom "officiel" de la GPIO. Par exemple, la GPIO5 de l'esp8266 est renommée D1. Mais parfois, cela correspond : la GPIO12 de l'esp32 est renommée 12.
- En conclusion, il faut toujours se baser sur les surnoms.

- Pour spécifier le sens d'utilisation d'une GPIO digitale, on utilise la fonction `pinMode()`.
- Généralement, ce sens ne varie pas au cours de l'exécution, donc on fait l'initialisation dans `setup()`. Au cas où le sens change, il suffit de réutiliser `pinMode()`.
- Exemple (sur esp8266) :

```
void setup() {
  pinMode(D3, OUTPUT);
  pinMode(D2, INPUT);
  ...
}
```

- Les fonctions pour lire et écrire changent selon le type de GPIO et s'il est disponible sur le µC.

Exemple (esp8266):

```
int val = analogRead(A0); // read value from 0 to 1023 (= 0V to 3.3V)
byte b = digitalRead(D2);
digitalWrite(D3,HIGH); // on an esp8266, D3 now outputs 3.3V
digitalWrite(D3,LOW); // on an esp8266, D3 now outputs 0V
```

Exemple (esp32) :

```
int val = analogRead(33); // read value from 0 to 4095 (= 0V to 3.3V)
byte b = digitalRead(8);
analogWrite(25, 1234); // write 1234 on pin 25 of an esp32, which corresponds to 3.3*1234/4096
digitalWrite(4,HIGH); // on an esp32, 4 now outputs 3.3V
digitalWrite(4,LOW); // on an esp32, 4 now outputs 0V
```

3°/ Communication série

- La plupart des cartes de développement ont un circuit qui permet d'envoyer un programme dans la mémoire flash liée au µC, au travers d'un câble USB.
- En fait, ce circuit permet de convertir les communications utilisant un protocole série en un protocole USB.
- Ce circuit est relié directement à 2 GPIO du µC, qui sont elles-mêmes reliées à une partie du µC capable d'envoyer recevoir des données via un protocole série.
- Il est donc possible d'envoyer recevoir des données entre le µC et un PC, en initialisant ce protocole série.
- Cela est particulièrement utile pour faire du débogage en envoyant des messages depuis le µC.
- Pour utiliser la communication série, le langage arduino fourni directement une classe nommée Serial.
- L'initialisation se fait généralement dans `setup()`, en donnant une vitesse en bauds.
- Ensuite, une méthode `println()` permet d'envoyer des lignes de texte, et read() permet de lire un octet (NB : pas de méthode pour lire des lignes !)
- Exemple(tout type µC) :

```
...
void setup() {
  Serial.begin(115200); // initialize serial connection at 115200 bauds
  Serial.println("hello");
  ...
}
```

Remarques :

- Les vitesse de communication maximales dépendent du µC. Par exemple, un ATmega328 (sur les cartes arduino) autorise seulement 9600 bauds.
- Certains modules, comme par exemple les GPS, utilisent également un protocole série pour communiquer avec le µC. Cela pose problème sur un esp8266 puisque le circuit série est déjà utilisé pour communiquer avec le PC. Fort heureusement, on peut faire de la communication série en utilisant n'importe quelles GPIOs digitales. Mais dans ce cas, il faut émuler le protocole série par du logiciel, ce qui est moins performant que de passer par l'électronique interne du µC.
- l'esp32 fourni 3 couples de GPIO capables nativement de faire des communications série.

4°/ Interruptions

- Certains µC sont capable de surveiller une GPIO digitale (pas possible pour analogique) et de vérifier s'il y a un changement d'état.
- Dans ce cas, le µC peut interrompre l'exécution en cours, et appeler une fonction callback nommée gestionnaire d'interruption.
- Après la fin de cette fonction, l'exécution reprend son cours là où elle avait été interrompue.
- Ce mécanisme est particulièrement utile pour détecter les appuis sur des boutons, les changements de position d'un switch, ... En effet, la façon basique de détecter un changement est de récupérer régulièrement l'état d'une GPIO, grâce à digitalRead(). Malheureusement, il est ainsi possible de rater un changement qui aurait lieu entre 2 lectures.
- Avec une interruption, on est assuré de réagir à tous les changements d'état.

Exemple : pin D3 d'une carte basée sur un esp8266

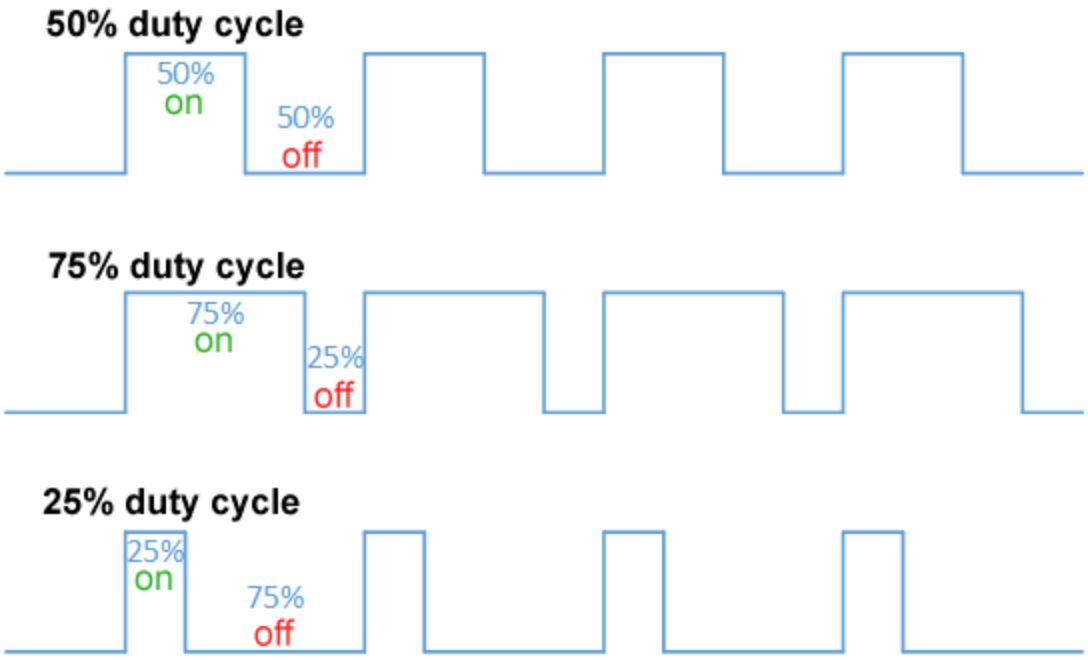
```
volatile byte state;
...
void ICACHE_RAM_ATTR mycallback() {
  state = digitalRead(D3);
  ...
}
...
void setup() {
  ...
  attachInterrupt(digitalPinToInterrupt(D3), mycallback, CHANGE);
  ...
}
```

Remarques:

- Il y a deux contraintes pour mettre en place un gestionnaire d'interruption :
  - le code de la fonction DOIT être en mémoire RAM et pas dans la mémoire flash (NB : comme c'est le cas par défaut), sinon le µC peut crasher. Pour forcer la mise en RAM, on utilise ICACHE_RAM_ATTR devant le nom de la fonction.
  - les variables globales manipulées par la fonction DOIVENT être également en RAM et pas en registre/cache. En effet, si un telle valeur est mise en registre, elle sera perdue puisqu'au moment du retour à l'exécution courante, les registres vont être restaurés avec leur valeur d'avant l'interruption. Pour ce faire, on utilise le mot-clé volatile devant le type de la variable.
- Le nom du gestionnaire d'interruption est libre, puisqu'il est donné en paramètre de `attachInterrupt()`.
- Le troisième paramètre de `attachInterrupt()` indique le type de changement détecté :
  - CHANGE détecte un changement détat du signal de LOW vers HIGH ou l'inverse.
  - RISING seulement de LOW vers HIGH,
  - FALLING seulement de HIGH vers LOW.
- `digitalPinToInterrupt()` est utilisé pour convertir le surnom de la GPIO en un numéro utilisé par le µC pour gérer les interruptions. Cette numérotation étant spécifique à chaque µC, on n'utilise jamais ce numéro dans le code mais on appelle `digitalPinToInterrupt()`.
- Le nombre de GPIO utilisables avec les interruptions dépend du type de µC. Sur esp8266/32, toutes les pins digitales peuvent être surveillées.

5°/ Pulse Width Modulation (PWM)

- PWM est un mécanisme qui utilise un GPIO digitale (pas possible pour analogique) pour émettre un signal cyclique composé d'impulsion plus ou moins large par rapport à la période du signal.
- Pendant la pulsation, le signal est à l'état haut et bas le reste du temps.
- Le ratio entre le temps passé à l'état haut et celui à l'état bas pour une même période est appelé le **duty cycle**. Il est généralement exprimé en %, comme le montre la figure ci-dessous avec 3 exemples de duty cycle.



- Le deuxième paramètre d'un signal PWM est la fréquence du signal, ce qui donne la durée de la période (= temps passé haut + temps passé bas).
- Par exemple, si la fréquence PWM est de 100Hz, la période est de 1/100 s, soit 10ms. Par conséquent, si le duty cycle est fixé à :
  - 10% : le signal est bas pendant 9ms puis haut pendant 1ms.
  - 50% : le signal est bas pendant 5ms puis haut pendant 5ms.
  - 80% : le signal est bas pendant 2ms puis haut pendant 8ms.

- Ce principe est utilisé pour alimenter des composants tels qu'un moteur pour faire varier la vitesse de rotation, une LED pour faire varier sa brillance.
- En effet, même si l'alimentation est effectivement faite par intermittence, si la fréquence est suffisamment élevée, un humain observera un phénomène continu mais moins "intense" qu'avec une alimentation en permanence à l'état haut.

- **Attention** : selon le µC, les principes de mise en place du PWM diffèrent un peu, notamment les fonctions à appeler. Ce qui suit donne les informations pour une esp8266. Pour un esp32, se référer à la documentation officielle : https://espressif-docs.readthedocs-hosted.com/projects/arduino-esp32/en/latest/api/ledc.html# (https://espressif-docs.readthedocs-hosted.com/projects/arduino-esp32/en/latest/api/ledc.html#)

- Les 3 fonctions utilisées pour mettre en place un signal PWM sur un esp8266 ont des noms commençant par `analog...()` bien que l'on utilise une GPIO digitale. Cela vient du fait que le PWM "simule" un signal analogique puisqu'on peut faire varier son niveau moyen en jouant sur la fréquence et le duty cycle.
- Ces fonctions sont :
  - `analogWriteRange(range)` : fixe le nombre de pas de discrétisation de la période, ce qui donne le nombre de valeur possibles pour le duty cycle (sans compter le 0%). Par exemple, si `range` = 2, il y a 2 pas de discrétisation donc les seules valeur possibles du duty cycle sont 0%, 50% et 100%. Si range = 100, il y a 100 pas, donc 101 valeurs possibles pour le duty cycle : 0%, 1%, 2%, ... 99% et 100%
  - `analogWriteFreq(freq)` : fixe la fréquence du signal Hertz. La plage de valeur de freq dépend du µC et de la valeur de range : plus range est petit plus feq peut être élevé. Par exemple, sur un esp8266, la fréquence peut aller jusqu'à 40KHz.
  - `analogWrite(pin, dc)` : démarre la génération PWM sur la GPIO pin. dc indique le nombre de pas de discrétisation de la période passés à l'état haut, donc permet de fixer indirectement le duty cycle. Par exemple ;
    - si range = 2 et dc = 1, alors on obtient un duty cycle de 50%

- si range = 10 et dc = 7, alors on obtient un duty cycle de 70%
- si range = 25 et dc = 10, alors on obitent un duty cycle de 40%
- ...

Exemple sur une esp8266/arduino :

```
analogWriteRange(100);
analogWirteFreq(10000); // 10KHz
analogWrite(D3,13); // duty cycle = 13/range = 13/100 =>  13%
```

Remarques :

- selon le µC, il y a plus ou moins de GPIOs digitales capables de produire un signal PWM. Sur l'esp8266/32, presque toutes en sont capables.
- sur l'esp32, les fonctions mentionnées ci-dessus n'existe pas et il faut utiliser la bibliothèque native ledc, qui fonctionne un peu différement.

6°/ A propos de la consommation électrique

- Utiliser le wifi (et/ou le bluetooth) sur un µC est ce qui consomme le plus d'énergie électrique.
- Par exemple, envoyer des données en Wifi sur un ESP32 soutire jusqu'à 250mA de la source d'alimentation, et environ 150mA en réception.
- Si le circuit Wifi est actif mais non communicant, le courant soutiré est entre 80 et 100mA.
- Ce ne sont pas des valeurs énormes mais elles sont tout de même suffisamment élevées pour poser des problèmes d'alimentation quand le µC fonctionne sur batterie/pile.
- Par exemple, une batterie 1000mAh est capable de délivrer 1A pendant une heure, ou bien 100mA pendant 10h.
- Cela veut dire qu'une telle batterie tiendrait seulement 1000/250 = 4 heures pour alimenter un µC constamment en train d'envoyer des données sur le Wifi.
- Et même si le µC n'envoie des données que quelques fois par jour, le fait d'avoir le wifi actif ne permettrait de tenir qu'au maximum 10h.
- Ce n'est évidemment pas applicable. Une solution de maison intelligente basée sur des boîtiers à µC, qu'il faut recharger toutes les 10h n'aurait pas beaucoup de succès.
- Il faut donc mettre en place des stratégies de réduction de la consommation.

- La première solution est de mettre régulièrement en pause le circuit wifi et de le réveiller juste le temps nécessaire pour vérifier s'il y a des données à recevoir depuis l'AP.
- Ce mode s'appelle "modem sleep" et c'est le mode par défaut de fonctionnement du circuit wifi sur un esp8266/32.
- Ce mode est possible grâce au fait qu'un AP émet régulièrement des trames dites de "balise" pour synchroniser le réseau (par ex tous les 100ms). Toutes les X trames de balise, la trame va contenir une information appelée DTIM (Delivery Traffic Information Map), qui permet à un client wifi de savoir s'il y a des données en attente sur l'AP pour lui. La trame contient également le temps avant la prochaine DTIM.
- Par conséquent, il suffit au µC de couper le circuit wifi entre deux trames contenant un DTIM et de se réveiller juste avant que la prochaine trame arrive.
- Avec cette "astuce", la consommation tombe en moyenne à 20mA, mais cela dépend du X (=nb balises entre 2 DTIM) qui est fixé par l'AP.
- En revanche, plus X est grand plus la latence de réception sera grande, ce qui peut poser problème à certaines applications où l'on a besoin d'une latence de communication la plus faible possible entre l'émetteur et le récepteur.
- Et si X est trop grand, il peut même y avoir des problèmes pour conserver la connexion wifi entre les client et l'AP.
- Malheureusement, même cette stratégie n'est pas très efficace puisqu'avec une batterie 1000mAh, il faudrait recharger au bout de 1000/20 = 50 heures. Pas terrible !

- Pour vraiment réduire la consommation, il faut passer à des modes beaucoup plus contraignants et qui coupent l'alimentation de plus ou moins de parties du µC.
- Ces modes ne sont utilisables que pour des applications où les µC envoient de façon sporadiques et qu'il n'y a quasi rien à faire entre deux communications. De plus, ce n'est pas adapté pour les cas où un µC doit pouvoir recevoir n'importe quand des données. C'est notamment le cas d'une solution où les µC se contentent de mesurer une grandeur, d'envoyer la valeur lue à un serveur, d'attendre sa réponse, et enfin s'endormir pour un temps fixe.
- Sur un esp8266/32, il y a principalement 3 modes :
  - light-sleep : le CPU, l'horloge, et le wifi sont en pause mais l'état du CPU est mis en RAM. Il est possible de réveiller le CPU via un changement d'état sur une GPIO, connectée par exemple à un bouton poussoir. Dans ce mode, la consommation tombe à environ 1mA, et l'exécution reprend là où elle s'était arrêtée. En revanche, le µC ne peut rien faire pendant l'attente.
  - deep-sleep : presque tout est arrêté excepté le circuit RTC (Real Time Clock) qui permet de compter le temps qui passe. Grâce à la RTC, le µC peut être programmé pour se réveiller au bout d'un temps paramétrable. On peut aussi le réveiller ĝrace à une GPIO. Cependant, contrairement au light-sleep, le µC reboote après le réveil et donc recommence entièrement le code du sketch. Dans ce mode, la consommation tombe à environ 10µA.
  - hibernation : tout est arrêté et le µC ne redémarre qu'avec un changement sur une GPIO précise (GPIO16 sur l'esp8266). Dans ce mode, la consommation tombe à 2µA.

- ATTENTION : les consommations annoncées ci-dessus ne tiennent compte que du µC seul. La consommation réelle est plus élevée, notamment à cause des composants actifs de la carte de développement : convertisseur série-usb, régulateur de tension, ... Par exemple, il est courant d'avoir une consommation réelle entre 50 et 100µA en deep-sleep.

- A part ces 3 modes, il est également possible de complètement couper le circuit wifi s'il n'est pas nécessaire avec : `WiFi.mode(WIFI_OFF)`
- Au contraire, pur qu'il soit toujours actif et avoir une latence minimale, on utilise : `WiFi.setSleepMode(WIFI_NONE_SLEEP)`

7°/ Mise en veille

- Comme dit en section 6, le mode deep sleep permet d'économiser la batterie.
- Généralement, ce mode s'arrête au bout d'un certain temps ou bien quand le µC reçoit un signal "externe" via une GPIO.
- La première solution est utilisée quand le µC doit faire des tâches régulières, et le second quand on attend un événement tel que l'appui sur un bouton ou bien une lumière suffisante.
- Dans les deux cas, l'inconvénient est que le µC reboot après sa sortie du deep-sleep, donc le contenu de la RAM est perdu. Néanmoins, il est possible de sauvegarder quelques données dans la RAM RTC : 512 octets sur l'esp8266 et 8192 sur l'esp32.
- Un autre inconvénient est que l'accès à la RAM RTC est différent entre les deux types de µC

7.1°/ sur l'esp8266

- Pour entrer en deep-sleep, il faut appeler la fonction `ESP.deepSleep(long time)`
- time est la durée du sommeil. Si = 0, il dure jusqu'à ce qu'un événement externe soit reçu.
- Pour réveiller le µC, il faut que la pin RST soit connectée à la masse (0V). Pour ce faire, il y a deux solutions, dépendantes du mode de réveil :
  - après un certain temps : la GPIO 16 (normalement, la pin D0) doit être connectée par un simple fil à la pin RST. Quand le temps est écoulé, la GPIO 16 passe automatiquement à l'état bas donc 0V, ce qui met également RST à 0V.
  - avec un signal externe : la pin RST est directement alimentée par un circuit qui émet un signal par défaut haut et qui passe à bas pour réveiller le µC. C'est par exemple un bouton avec une résistance de pull-up.
- Pour lire/écrire en RAM RTC: `ESP.rtcUserMemoryRead()` et `ESP.rtcUserMemoryWrite()`

Exemple (esp8266):

```
uint32_t bootcount;
void setup() {
  Serial.begin(115200);
  ...
  ESP.rtcUserMemoryRead(0, &bootcount, sizeof(bootcount));
  Serial.println(bootcount);
  ...
  bootcount++;
  ESP.rtcUserMemoryWrite(0, &bootcount, sizeof(bootcount));
  ...
}
void loop() {
  ...
  ESP.deepSleep(5000000); // sleep 5s, (=5000000µs)
}
```

7.2°/ sur l'esp32

- Le deep sleep est plus facile à mettre en place et plus souple :
  - aucune pin spécial pour réveiller
  - plusieurs pins (0, 2, 4, 12-15, 25-27, 32-39) peuvent être utilisée pour recevoir un signal externe
  - on peut mixer temps de sommeil + signal externe, au premier qui aura lieu
  - manipuler la RAM RTC se fait grâce à la macro RTC_DATA_ATTR devant les noms de variable.

Exemple (esp32):

```
RTC_DATA_ATTR int bootCount = 0;

// print the reason by which ESP32 has been awaken from sleep
void print_wakeup_reason(){
  esp_sleep_wakeup_cause_t wakeup_reason;
  wakeup_reason = esp_sleep_get_wakeup_cause();
  switch(wakeup_reason)  {
    case ESP_SLEEP_WAKEUP_EXT0 : Serial.println("Wakeup caused by external signal using RTC_IO"
    case ESP_SLEEP_WAKEUP_EXT1 : Serial.println("Wakeup caused by external signal using RTC_CN
    case ESP_SLEEP_WAKEUP_TIMER : Serial.println("Wakeup caused by timer"); break;
    case ESP_SLEEP_WAKEUP_TOUCHPAD : Serial.println("Wakeup caused by touchpad"); break;
    case ESP_SLEEP_WAKEUP_ULP : Serial.println("Wakeup caused by ULP program"); break;
    default : Serial.printf("Wakeup was not caused by deep sleep: %d\n",wakeup_reason); break;
  }
}

void setup() {
  Serial.begin(115200);
  ++bootCount;
  Serial.println(bootCount);
  print_wakeup_reason();
  // configure pin 33 as external wakeup source
  esp_sleep_enable_ext0_wakeup(GPIO_NUM_33,1); //1 = High, 0 = Low
  // configure deep sleep timer
  esp_sleep_enable_timer_wakeup(5000000); // time in µ-seconds
}

void loop() {
  delay(1000);
  Serial.println("Going to sleep now");
  esp_deep_sleep_start();
  Serial.println("This will never be printed");
}
```