Les Timers

Timers

Les SoC ESP32 contiennent de 2 à 4 minuteries matérielles. Ce sont tous des temporisateurs génériques 64 bits (54 bits pour ESP32-C3) basés sur des pré-scalers 16 bits et des compteurs croissants/décroissants 64 bits (54 bits pour ESP32-C3) capables d'être rechargés automatiquement.

SoC ESP32	Nombre de minuteries
ESP32	4
ESP32-S2	4
ESP32-C3	2
ESP32-S3	4

Arduino-ESP32 Timer API¶

Timers sur ESP32

Fondamentalement, un Timer (minuterie) est une horloge, qui est utilisée pour mesurer et contrôler les événements temporels. offrant une temporisation précise. La plupart des microcontrôleurs ont des minuteries intégrées. Les minuteries des microcontrôleurs ne sont pas seulement utilisées pour générer des retards, mais sont également utilisées comme compteur. Cette caractéristique du temporisateur est utilisée pour de nombreuses applications. Les minuteries du microcontrôleur sont contrôlées par des registres de fonctions spéciales qui sont affectés aux opérations de minuterie.

Une interruption est un événement externe qui interrompt le programme en cours et exécute une routine de service d'interruption (ISR).

Une fois l'ISR terminé, le programme en cours se poursuit avec l'instruction suivante. Et les interruptions du temporisateur sont les interruptions générées par le temporisateur. Voici l'exemple montrant comment configurer le temporisateur pour générer périodiquement une interruption et comment la gérer. ESP32 a deux groupes de minuteries, chacun avec deux minuteries matérielles à usage général. Tous les temporisateurs sont basés sur des compteurs 64 bits et des prédiviseurs 16 bits. Le prédiviseur est utilisé pour diviser la fréquence du signal de base (généralement 80 MHz), qui est ensuite utilisé pour incrémenter ou décrémenter le compteur de la minuterie. La variable de compteur sera partagée entre la boucle principale et l'ISR, puis elle doit être déclarée avec le mot-clé volatile .

```
volatile int interruptCounter ;
```

Nous aurons un compteur supplémentaire pour suivre le nombre d'interruptions déjà survenues.

```
 int totalInterruptCounter ;
```

Afin de configurer le timer, nous aurons besoin d'un pointeur vers une variable de type hw_timer_t .

```
 hw_timer_t * timer = NULL;
```

Enfin, nous devrons déclarer une variable de type portMUX_TYPE qui nous servira à nous occuper de la synchronisation entre la boucle principale et l'ISR.

```
portMUX_TYPE timerMux = portMUX_INITIALIZER_UNLOCKED;
```

Pour initialiser le timer à l'aide d'une fonction timerbegin , cette fonction reçoit le numéro du timer que l'on souhaite utiliser (de 0 à 3, puisque nous avons 4 timers matériels), la valeur du prescaler et un drapeau indiquant si le compteur doit compter (vrai) ou vers le bas (faux).

```
timer = timerBegin(0, 80, vrai);
```

Pour cet exemple, nous utiliserons la première minuterie et passerons vrai au dernier paramètre, donc le compteur compte la fréquence du signal de base utilisé par les compteurs ESP32 est de 80 MHz . Si nous divisons cette valeur par 80 (en utilisant 80 comme valeur du prescaler), nous obtiendrons un signal avec une fréquence de 1 MHz qui incrémentera le compteur de la minuterie 1 000 000 fois par seconde.

Avant d'activer le temporisateur, nous devons le lier à une fonction de traitement, qui sera exécutée lorsque l'interruption sera générée. Cela se fait avec un appel à la fonction timerAttachInterrupt .

```
timerAttachInterrupt(timer, &onTimer, true);
```

Cette fonction reçoit en entrée un pointeur vers le timer initialisé, que nous avons stocké dans notre variable globale, l'adresse de la fonction qui va gérer l'interruption et un drapeau indiquant si l'interruption à générer est front (true) ou level (false) . Pour cet exemple, nous passerons notre variable de minuterie globale en première entrée, en deuxième l'adresse d'une fonction appelée onTimer que nous spécifierons plus tard, et en troisième la valeur true , donc l'interruption générée est de type edge.

```
timerAlarmWrite(timer, 1000000, true);
```

fonction timerAlarmWrite pour spécifier la valeur du compteur dans laquelle l'interruption du temporisateur a été générée. Donc, pour cet exemple, on suppose que l'on veut générer une interruption chaque seconde, et on passe donc la valeur de 1 000 000 microsecondes, qui est égale à 1 seconde. Le troisième argument que nous passerons true , donc le compteur se rechargera et donc l'interruption sera périodiquement générée. Pour terminer la fonction de configuration en activant un appel à timerAlarmEnable(timer);

Main loop

La boucle principale (main loop) sera l'endroit où nous gérons réellement l'interruption de la minuterie, après qu'elle ait été signalée par l' ISR (routine de service d'interruption également appelée gestionnaire d'interruption) . Pour vérifier la valeur du compteur d'interruptions, nous allons donc vérifier si lavariable du compteur d'interruptions est supérieur à zéro et si c'est le cas, nous entrerons le code de gestion des interruptions. Là, la première chose que nous allons faire est de décrémenter ce compteur, signalant que l'interruption a été acquittée et sera traitée.

1.ino

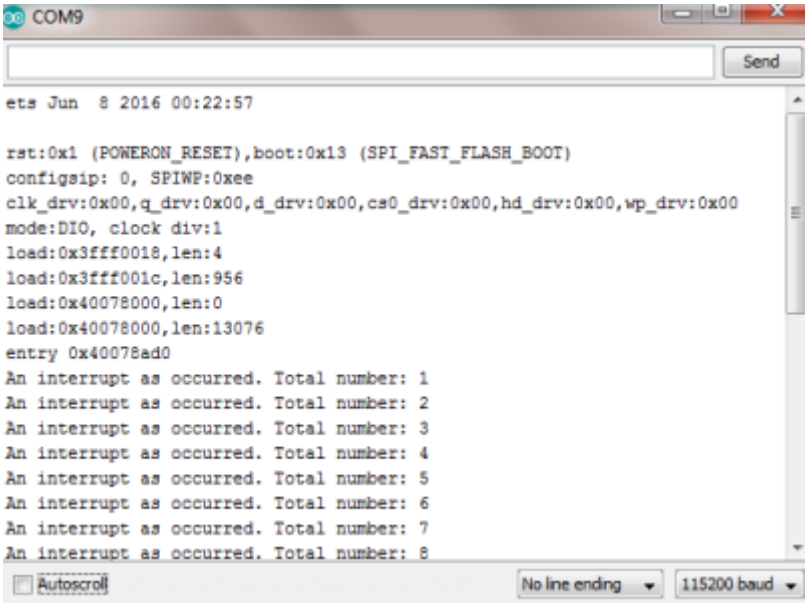
```
  if (interruptCounter > 0) {

    portENTER_CRITICAL(&timerMux);
    interruptCounter--;
    portEXIT_CRITICAL(&timerMux);

    totalInterruptCounter++;

    Serial.print("An interrupt as occurred. Total number: ");
    Serial.println(totalInterruptCounter);

  }
```

La fonction ISR doit être une fonction qui renvoie void et ne reçoit aucun argument. La routine de gestion des interruptions doit avoir l' attribut IRAM_ATTR , pour que le compilateur place le code dans IRAM. De plus, les routines de gestion des interruptions ne doivent appeler que les fonctions également placées dans l' IRAM.

2.ino

```
void IRAM_ATTR onTimer() {
 portENTER_CRITICAL_ISR(&timerMux);
 interruptCounter++;
 portEXIT_CRITICAL_ISR(&timerMux);
  }
```

Puisque cette variable est partagée avec l'ISR, nous le ferons dans une section critique, que nous spécifions en utilisant une macro portENTER_CRITICAL et une macro portEXIT_CRITICAL . Ces deux appels reçoivent en argument l'adresse de notre variable globale portMUX_TYPE . La gestion réelle des interruptions consistera simplement à incrémenter le compteur avec le nombre total d'interruptions survenues depuis le début du programme et à l'imprimer sur le port série.



ESP32TimerInterrupt Library

ESP32TimerInterrupt Library

Pourquoi avons-nous besoin de cette bibliothèque ESP32TimerInterrupt

Caractéristiques

Cette bibliothèque vous permet d'utiliser “Interrupt from Hardware Timers” sur une carte basée sur ESP32.

Comme les temporisateurs matériels sont des atouts rares et très précieux de n'importe quelle carte, cette bibliothèque vous permet désormais d'utiliser jusqu'à 16 temporisateurs basés sur ISR, tout en ne consommant qu'un seul temporisateur matériel . L'intervalle des temporisateurs est très long (ulong millisecs).

Désormais, avec ces nouveaux temporisateurs basés sur 16 ISR , l'intervalle maximal est pratiquement illimité (limité uniquement par de longues millisecondes non signées) tandis que la précision est presque parfaite par rapport aux temporisateurs logiciels.

La caractéristique la plus importante est qu'il s'agit de minuteries basées sur ISR. Par conséquent, leurs exécutions ne sont pas bloquées par des fonctions/tâches au mauvais comportement . Cette fonctionnalité importante est absolument nécessaire pour les tâches critiques.

L' exemple ISR_Timer_Complex démontrera la précision presque parfaite par rapport aux temporisateurs logiciels en imprimant les millisecondes écoulées réelles de chaque type de temporisateurs.

Étant des minuteries basées sur ISR, leurs exécutions ne sont pas bloquées par des fonctions/tâches qui se comportent mal, telles que la connexion aux services WiFi, Internet et Blynk. Vous pouvez également avoir plusieurs (up to 16)minuteries à utiliser.

Cette fonctionnalité importante non bloquée est absolument nécessaire pour les tâches critiques.

Vous verrez que le logiciel blynkTimer est bloqué pendant que le système se connecte au WiFi / Internet / Blynk, ainsi qu'en bloquant la tâche dans loop(), en utilisant la fonction delay() comme exemple. Le temps écoulé est alors très imprécis Pourquoi l'utilisation d'une interruption de minuterie matérielle basée sur ISR est préférable

Imaginez que vous ayez un système avec une fonction critique , mesurant le niveau d'eau et contrôlant la pompe de puisard ou faisant quelque chose de beaucoup plus important. Vous utilisez normalement une minuterie logicielle pour interroger, ou même placer la fonction dans loop(). Mais que se passe-t-il si une autre fonction bloque la boucle() ou la configuration().

Ainsi, votre fonction pourrait ne pas être exécutée et le résultat serait désastreux.

Vous préféreriez que votre fonction soit appelée, quoi qu'il arrive avec d'autres fonctions (boucle occupée, bogue, etc.).

Le bon choix consiste à utiliser une minuterie matérielle avec interruption pour appeler votre fonction.

Ces temporisateurs matériels, utilisant l'interruption, fonctionnent toujours même si d'autres fonctions bloquent. De plus, ils sont beaucoup plus précis (certainement en fonction de la précision de la fréquence d'horloge) que les autres temporisateurs logiciels utilisant millis() ou micros(). Cela est nécessaire si vous avez besoin de mesurer certaines données nécessitant une meilleure précision.

Les fonctions utilisant des minuteries logicielles normales, reposant sur loop() et appelant millis(), ne fonctionneront pas si loop() ou setup() est bloqué par certaines opérations. Par exemple, certaines fonctions se bloquent lors de la connexion au WiFi ou à certains services.

Le hic, c'est que votre fonction fait maintenant partie d'un ISR (Interrupt Service Routine), et doit être courte et rapide , et suivre certaines règles. Plus à lire sur :

HOWTO Attach Interrupt

Minuterie à usage général (GPTimer)

Minuterie à usage général (GPTimer)

Minuterie haute résolution

Minuterie haute résolution