Multitasking met de ESP32 (6)

event groups

Warren Gay (Canada)

Bij het ontwikkelen van applicaties die onder FreeRTOS draaien, is het vaak belangrijk om meerdere taken met elkaar te synchroniseren. In de vorige delen van deze serie hebben we semafoor- en taak-notificaties onderzocht. Maar daarmee kunnen we alleen events van een taak of interruptroutine (ISR) naar één enkele ontvangende taak sturen. Als u een event naar meerdere taken moet sturen, kunt u gebruik maken van een event group. In dit laatste artikel van deze serie onderzoeken we de mogelijkheden die dat biedt voor embedded ontwikkelaars.

Laten we beginnen met een praktisch voorbeeld. Een MIDI-drummachine kan via zijn seriële input opdracht krijgen om vier geluiden tegelijk te maken. Dan zouden alle vier geluid-genererende taken op exact hetzelfde moment moeten beginnen om te voorkomen dat het geluid voor het gehoor 'ongelijk' klinkt. We zouden dat kunnen proberen te doen met vier afzonderlijke semaforen en erop vertrouwen dat de CPU zo snel is dat het voor de toehoorder lijkt of de taken tegelijk starten. Maar die aanpak is onhandig en schaalt niet goed als het aantal ontvangende taken toeneemt. Event groups (event-groepen) zijn in FreeRTOS de aanbevolen methode voor het coördineren van meerdere taken vanuit één event-bron.

Event-vlaggen

FreeRTOS definieert voor elk gecreëerd event-groep-object 24 event-vlaggen (**figuur 1**). Deze vlaggen worden weergegeven in het C-datatype `EventBits_t`, een 32bit-datatype waarvan 24 bit beschikbaar zijn. Bit 0 is het minst significante bit (LSB) van de beschikbare vlaggen. De meest significante 8 bit zijn *gereserveerd* voor intern gebruik door FreeRTOS.

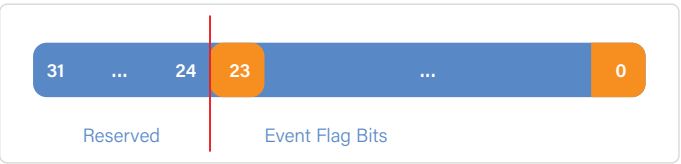
Hoe die 24 bit worden toegewezen en gebruikt door de applicatie kan de programmeur zelf beslissen. In deze demo genereert de taak `loop()` elke seconde een event dat twee taken opdracht geeft om synchroon te starten met een eigen knipperpatroon via een eigen LED. Bit 0 van de event group (waarde `0b0001` / decimale waarde 1) stuurt een melding naar de taak `blink2()`, die LED1 twee keer laat knipperen. Bit 1 van de event group (waarde `0b0010` / decimale waarde 2) stuurt de taak `blink3()` aan, die LED2 drie keer laat knipperen.

Demoprogramma

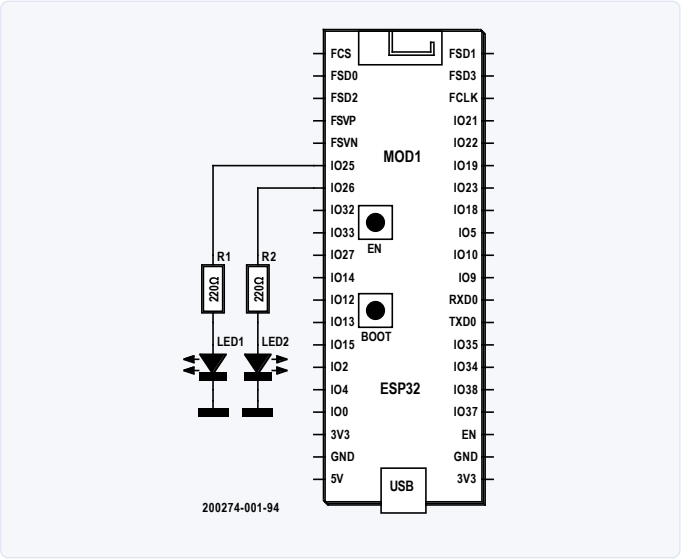
Laten we, voordat we in de code duiken, eens kijken wat ons demoprogramma moet gaan doen. Twee LED's worden aangestuurd met de GPIO-pennen 25 en 26 (regels 5 en 6) in een actief hoge configuratie (zie het schema in **figuur 2**). Deze GPIO's zijn in `setup()` geconfigureerd als uitgangen (regels 62 tot en met 65). Taak `blink2()` bestuurt LED1 door hem twee keer te laten knipperen en dan weer te wachten op het volgende event (regels 19...25). Taak `blink3()` bestuurt LED2 door hem drie keer te laten knipperen en dan weer te wachten op het volgende event (regels 41...47). Beide taken zijn identiek, met uitzondering van het aantal keren dat de knipperlus wordt uitgevoerd (regels 27 en 49) en de gebruikte vertragingstijden.

Synchronisatie

De synchronisatie wordt bereikt door de taakfuncties te laten blokkeren in een aanroep van `xEventGroupWaitBits()`. Het aanroepen van deze functie stopt de uitvoering van de taak totdat het event wordt



Figuur 1. Event-vlaggen in het C-datatype EventBits_t.



Figuur 2: Het schema voor het demoprogramma evtgrp.ino.

Listing 1. Het Arduino-programma evtgrp.ino met event groups [1].

```
0001: // evtgrp.ino
0002: // MIT License (see file LICENSE)
0003:
0004: // LED is active high
0005: #define GPIO_LED1     25
0006: #define GPIO_LED2     26
0007:
0008: #define EVBLK2        0b0001
0009: #define EVBLK3        0b0010
0010: #define EVALL         0b0011
0011:
0012: static EventGroupHandle_t hevt;
0013:
0014: void blink2(void *arg) {
0015:
0016:   for (;;) {
0017:     // Call blocks until EVBLK2 bit set,
0018:     // and same bit is cleared upon return
0019:     xEventGroupWaitBits(
0020:       hevt,
0021:       EVBLK2,
0022:       pdTRUE,
0023:       pdFALSE,
0024:       portMAX_DELAY
0025:     );
0026:     // Blink 2 times
0027:     for ( int x=0; x < 2; ++x ) {
0028:       digitalWrite(GPIO_LED1,HIGH);
0029:       delay(120);
0030:       digitalWrite(GPIO_LED1,LOW);
0031:       delay(120);
0032:     }
0033:   }
0034: }
0035:
0036: void blink3(void *arg) {
0037:
0038:   for (;;) {
0039:     // Call blocks until EVBLK3 bit set,
0040:     // and same bit is cleared upon return
0041:     xEventGroupWaitBits(
0042:       hevt,
0043:       EVBLK3,
0044:       pdTRUE,
0045:       pdFALSE,
0046:       portMAX_DELAY
0047:     );
0048:     // Blink 3 times
0049:     for ( int x=0; x < 3; ++x ) {
0050:       digitalWrite(GPIO_LED2,HIGH);
0051:       delay(75);
0052:       digitalWrite(GPIO_LED2,LOW);
0053:       delay(75);
0054:     }
0055:   }
0056: }
0057:
0058: void setup() {
0059:   int app_cpu = xPortGetCoreID();
0060:   BaseType_t rc;
0061:
0062:   pinMode(GPIO_LED1,OUTPUT);
0063:   pinMode(GPIO_LED2,OUTPUT);
0064:   digitalWrite(GPIO_LED1,LOW);
0065:   digitalWrite(GPIO_LED2,LOW);
0066:
0067:   delay(2000);
0068:
0069:   hevt = xEventGroupCreate();
0070:   assert(hevt);
0071:
0072:   rc = xTaskCreatePinnedToCore(
0073:     blink2,   // func
0074:     "blink2", // name
0075:     1024,     // stack bytes
0076:     nullptr,  // arg ptr
0077:     1,        // priority
0078:     nullptr,  // ptr to task handle
0079:     app_cpu   // cpu#
0080:   );
0081:   assert(rc == pdPASS);
0082:
0083:   rc = xTaskCreatePinnedToCore(
0084:     blink3,   // func
0085:     "blink3", // name
0086:     1024,     // stack bytes
0087:     nullptr,  // arg ptr
0088:     1,        // priority
0089:     nullptr,  // ptr to task handle
0090:     app_cpu   // cpu#
0091:   );
0092:   assert(rc == pdPASS);
0093: }
0094:
0095: void loop() {
0096:
0097:   delay(1000);
0098:   xEventGroupSetBits(
0099:     hevt,
0100:     EVBLK2|EVBLK3
0101:   );
0102: }
0103:
0104: // End evtgrp.ino
```

getriggerd. Zodra de event groep wordt getriggerd, wordt de uitvoering van de geblokkeerde taken hervat door terug te keren uit de aangeroepen functie. Het event wordt elke seconde getriggerd door een aanroep van `xEventGroupSetBits()` in de taak `loop()` (zie regels 98 tot 101 in **listing 1** [1]).

Event groups aanmaken

De event groep wordt gecreëerd door `xEventGroupCreate()` aan te roepen (regel 69). Deze functie wordt aangeroepen zonder parameters en hij geeft de handle terug van het toegewezen event-groep-object. Het geretourneerde datatype van deze handle is `EventGroupHandle_t`. Als de geretourneerde waarde nul is, was er geen ruimte in de heap beschikbaar (vandaar de controle in regel 70).

```
0069:   hevt = xEventGroupCreate();
0070:   assert(hevt);
```

Notificatie

Er zijn verschillende manieren om event-groepen te manipuleren. In dit artikel gebruiken we één van de eenvoudigere functies: `EventGroupSetBits()`.

```
EventBits_t xEventGroupSetBits(
  EventGroupHandle_t xEventGroup,    // Handle
  const EventBits_t uxBitsToSet      // Bits to set
);
```

De handle `xEventGroup` specificeert de event groep die moet worden aangestuurd en het argument `uxBitsToSet` geeft aan welke event bits atomair gezet moeten worden. De waarde die wordt geretourneerd bestaat uit de event-bits die actief waren *na* de aanroep van `xEventGroupSetBits()`. U moet er wel rekening mee houden dat de geretourneerde waarde niet altijd de bits bevat die u net hebt gezet, omdat de ontvangende taken ze misschien al bij ontvangst hebben gewist.

De taak `loop()` roept de functie `xEventGroupSetBits()` aan die bit 0 en bit 1 instelt met behulp van de macro-uitdrukking `EVBLK2|EVBLK3`.

```
0008: #define EVBLK2        0b0001
0009: #define EVBLK3        0b0010

0098:   xEventGroupSetBits(
0099:     hevt,
0100:     EVBLK2|EVBLK3
0101:   );
```

Uit deze aanroep blijkt dat de bits 0 en 1 atomair op '1' zijn gezet.

De taak loop()

De taak `loop()` wacht een seconde (regel 97) en stuurt dan een event notificatie (regels 98...101). Daarna eindigt de functie en wordt die herhaald, zoals te doen gebruikelijk bij Arduino. Dat heeft tot gevolg dat de events ongeveer één keer per seconde worden getriggerd.

Ontvangende taken

De taken `blink2()` en `blink3()` worden getriggerd door de event groep met behulp van de opgeslagen handle (regels 12 en 69). Zo wacht `blink2()` op de regels 19...25 totdat een event wordt getriggerd. Op dezelfde manier wacht `blink3()` op de regels 41 tot en met 47 totdat een event wordt getriggerd. Omdat de events atomair worden getriggerd in de regels 98...01, zullen beide taken tegelijkertijd worden hervat. Als de twee taken aan dezelfde CPU zijn toegewezen, zullen ze vervolgens na elkaar worden uitgevoerd. Als de twee taken aan verschillende CPU's zijn toegewezen, is het werkelijk mogelijk om beide taken tegelijkertijd te hervatten en uit te voeren.
Als de uitvoering van een taak wordt hervat, kan hij de bijbehorende LED op zijn eigen unieke manier laten knipperen. Zodra dat knipperen is voltooid, wordt de uitvoering bovenaan de takenlijst hervat door te wachten op de komst van het volgende event.

Events resetten

De aanroep van de functie `xEventGroupWaitBits()` is een beetje gecompliceerd. Hij biedt veel mogelijkheden, maar u moet wel blijven opletten. Laten we dat eens in detail bekijken:

```
EventBits_t xEventGroupWaitBits(
  const EventGroupHandle_t xEventGroup, // Event
   group handle
  const EventBits_t uxBitsToWaitFor,
  const BaseType_t xClearOnExit,
  const BaseType_t xWaitForAllBits,
  TickType_t xTicksToWait
);
```

Het eerste argument is de handle van de te gebruiken event groep. Dat is heel eenvoudig. Argument twee specificeert de event-bits waarop de functie wil wachten. In het demoprogramma hebben we de macro `EVBLK2` (regel 21) gespecificeerd voor de taak `blink2()` en de macro `EVBLK3` voor de taak `blink3()`. De reden om daar verschillende bits voor te gebruiken zal snel duidelijk worden.
Het derde argument `xClearOnExit` is het C-equivalent van een boolean. In de regels 22 en 44 krijgt die de waarde `pdTRUE`. Dit vertelt de functie dat hij de ontvangen event-bits moet resetten alvorens terug te keren. Dus de taak `blink2()` wacht tot bit 0 is gezet (macro `EVBLK2`), en de waarde `pdTRUE` van de derde parameter zegt hem dat hij het bit moet wissen bij ontvangst en terugkeer. De wacht-en-reset-operatie wordt atomair uitgevoerd.

In ons geval is het vierde argument van academisch belang, maar we zullen die toch bespreken. De parameter `xWaitForAllBits` is ook een boolean en geeft aan wanneer de functie moet terugkeren:

- wanneer *één* van de bits waarop wordt gewacht, is gezet (`pdFALSE`), *of*
- pas als *alle* bits waarop wordt gewacht zijn gezet (`pdTRUE`).

In ons voorbeeld wordt er maar op één enkel bit gewacht (bit 0 voor `blink2()` en bit 1 voor `blink3()`), dus dan maakt de waarde van deze parameter niet uit. Maar als u wilt wachten op een combinatie van bits, kan deze mogelijkheid heel nuttig zijn.

Event-besturing

Het zal nu wel duidelijk zijn, waarom er aparte event bits nodig waren: elke ontvangende taak wacht op zijn eigen event bit en wist het dan bij de aanroep van `xEventGroupWaitBits()`. Er moet dus een apart bit zijn voor elke taak. In de functie `loop()` worden beide taken tegelij-

WEBLINKS

[1] **Code voor evtgrp.ino**: https://bit.ly/35YrjCN
[2] **W. Gay, FreeRTOS for ESP32-Arduino, Elektor 2020**: https://bit.ly/2U2Yhg1
[3] **FreeRTOS-documentatie**: https://bit.ly/386HZL6

kertijd getriggerd door hun eigen event bits op hetzelfde moment in te stellen (regel 100).
Zoals u ziet, kunnen we met event group bits zeer complexe en creatieve applicatiecode creëren. Zie voor meer details over de extra functies het boek *FreeRTOS for ESP32-Arduino* [2] en de FreeRTOS-documentatie [3].

Uitvoeren van de demo

De demo maakt geen gebruik van de seriële monitor en kan worden uitgevoerd op zo ongeveer elke ESP32 die GPIO 25 en 26 beschikbaar heeft (of u kunt de regels 5 en 6 van **listing 1** aanpassen om andere uitgangen te gebruiken). Flash het programma evtgrp.ino in de ESP32 en sluit twee LED's aan met weerstanden van 220 Ω volgens het schema in **figuur 2**. Zodra de ESP32 begint met de uitvoering moeten de LED's gaan knipperen: één keer per seconde moet LED1 twee maal knipperen en dan wachten, terwijl LED2 drie keer (snel) moet knipperen en dan wachten. Het proces wordt elke seconde herhaald.

Synchronisatie voltooid

Deze demo illustreert hoe twee volledig onafhankelijke taken, `blink2()` en `blink3()`, gesynchroniseerd kunnen worden met behulp van één enkel event group-object. De taak `loop()` zendt zijn event vanaf regels 98...101. Deze aanpak kan indien nodig worden uitgebreid tot maximaal 24 taken (de limiet wordt bepaald door het aantal beschikbare event-vlaggen in een event groep). Dit artikel heeft één manier laten zien om event groups te gebruiken, maar er zijn nog een heleboel andere, meer geavanceerde, manieren om events te coördineren met behulp van andere FreeRTOS-functies.

200528-04

Vragen of opmerkingen?
Hebt u vragen of opmerkingen naar aanleiding van dit artikel? Stuur een e-mail naar de auteur of naar de redactie van Elektor, via redactie@elektor.com.

Een bijdrage van
Idee en tekst: **Warren Gay**
Schema en illustratie: **Patrick Wielders en Jack Jamar**
Redactie: **Stuart Cording**
Vertaling: **Evelien Snel**
Layout: **Giel Dols**

GERELATEERDE PRODUCTEN

- **W. Gay, FreeRTOS for ESP32-Arduino, Elektor 2020**
  www.elektor.nl/freertos-for-esp32-arduino