Multitasking met de ESP32 (5)

task event-notificaties

Warren Gay (Canada)

In FreeRTOS kunnen we gebruik maken van wachtrij-, semafoor- en mutexfuncties voor synchronisatie, maar soms is het daarmee toch behoorlijk ingewikkeld om vrij simpele dingen te doen. Daarom is in FreeRTOS versie V8.2.0 het concept *direct to task event notifications* geïntroduceerd. Dat biedt de programmeur een eenvoudige methode om te synchroniseren tussen taken.

Elke taak bevat een ingebouwde 32-bits *event notification*-variabele die op nul wordt geïnitialiseerd wanneer de taak wordt aangemaakt. Omdat die variabele in elke taak is ingebouwd, is er geen extra RAM nodig. Dit vraagt aanzienlijk minder geheugen dan bijvoorbeeld het gebruik van semafoorobjecten.

Beperkingen

Er zijn wel enkele beperkingen bij het gebruik van directe task-notificaties, zoals:

- het is niet mogelijk om een notificatie naar een ISR te sturen, want een ISR is geen taak. Een ISR kan zelf *wel* een notificatie sturen;
- een notificatie kan maar naar één taak tegelijk worden gestuurd (om meerdere taken aan te spreken, moeten event-groepen worden gebruikt);
- event-notificaties kunnen niet worden gebufferd zoals wachtrijen;
- de *notificerende* taak (of ISR) stopt niet om te wachten tot de ontvangende taak het event ontvangt. Het event wordt alleen gepost en de uitvoering van de aanroepende taak gaat ongehinderd door.

Als een van deze beperkingen niet past bij wat u wilt, moet u in plaats daarvan een alternatief FreeRTOS-mechanisme kiezen.

Wachten op een event

De eerste stap bij het gebruik van task event-notificaties is de ontvangende taak opdracht te geven om te wachten op een notificatie. Met andere woorden: de taak moet worden ingesteld om te pauzeren totdat een event wordt geactiveerd. We implementeren dat door gebruik te maken van een van de volgende functies:

- `ulTaskNotifyTake()`
- `xTaskNotifyWait()`

De functie `ulTaskNotifyTake()` is de eenvoudigste van de twee en die gaan we in dit artikel onderzoeken. `xTaskNotifyWait()` is geavanceerder en wordt in detail onderzocht in het Elektor-boek "FreeRTOS for ESP32-Arduino".

ulTaskNotifyTake()

De taak die een notificatie moet ontvangen, blokkeert zijn uitvoering door `ulTaskNotifyTake()` aan te roepen. Deze aanroep heeft twee argumenten nodig:

```
uint32_t ulTaskNotifyTake(
  BaseType_t xClearCountOnExit, // pdFALSE or pdTRUE
  TickType_t xTicksToWait
);
```

Het eerste argument werkt op het 32-bits task notificatie-woord (ook wel het *task event word* genoemd). Zie **tabel 1** voor de betekenis. De tweede parameter is de bekende time-out waarde in tikken (gebruik de macro `pdMAX_DELAY` als u geen time-out wilt). Bij alle aanroepen naar `ulTaskNotifyTake()` wordt de uitvoering van de aanroepende taak geblokkeerd zolang het task notificatie-woord op de waarde nul blijft staan. Als de waarde ongelijk nul wordt, bepaalt het argument `xClearCountOnExit` hoe het task notificatie-woord wordt bijgewerkt voordat de uitvoering van onze taak verder gaat.

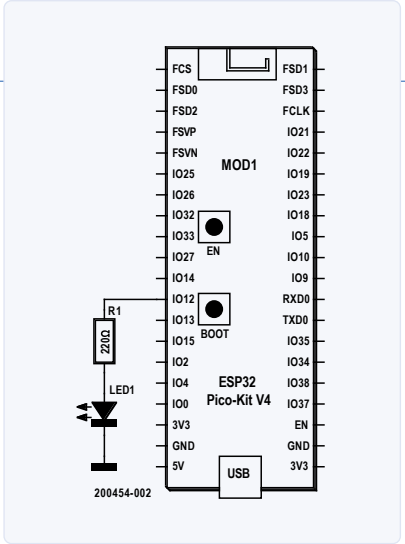
Tabel 1. De betekenis van de `xClearCountOnExit` argumenten voor `ulTaskNotifyTake()`.

Waarde van `xClearCountOnExit`	Effect op het 32-bits tasknotificatie-woord
`pdFALSE`	Decrementeer de waarde, blokkeer verdere verwerking als de waarde voor het decrementeren al nul was, alvorens de waarde voor het decrementeren te retourneren.
`pdTRUE`	Zet de waarde op 0, blokkeer verdere verwerking als de waarde al nul was, alvorens de waarde voor het wissen te retourneren.

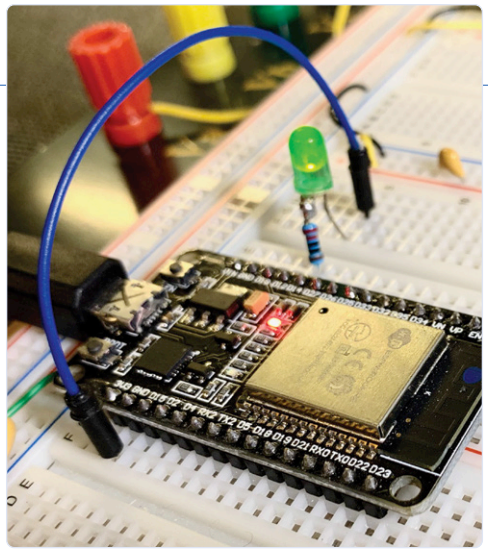
De retourwaarde van de functie is de waarde van het task notificatie-woord *voordat* het werd gewist of gedecrementeerd. Als er een *time-out* optreedt, is de retourwaarde ook nul.

Binaire notificatie

Als het argument `xClearCountOnExit` de waarde `pdTRUE` heeft, werkt de aanroep naar `ulTaskNotifyTake()` als de *take*-operatie van een binaire semafoor. Als het task notificatie-woord *al ongelijk is aan nul*, dan wordt de call onmiddellijk geretourneerd en wordt het event-woord op nul gezet (de eerdere waarde, ongelijk aan nul, wordt geretourneerd). Als het task notificatie-woord op het moment van de aanroep nul was, wordt de uitvoering van de taak *geblokkeerd* totdat er een notificatie wordt ontvangen (of totdat er een time-out optreedt).



Figuur 1. Schema voor de tasknfy.ino code.



Figuur 2. Bouw van de demoschakeling (aangesloten op GPIO12).

Als er een time-out optreedt, zal de retourwaarde altijd nul zijn (net als de waarde van het notificatie-woord op het moment van de terugkeer). Zo werkt de aanroep als een binaire semafoor. Als de taak twee meldingen krijgt voordat de ontvangende taak `ulTaskNotifyTake()` aanroept, dan zal de geretourneerde waarde 2 zijn. Maar het task notificatie-woord wordt bij terugkeer gewist naar *nul*. In dit geval is er in feite dus maar één notificatie-event, maar het werkelijke aantal meldingen kan gemakkelijk worden bepaald aan de hand van de geretourneerde waarde.

Tellende notificatie

Als het de bedoeling is dat meerdere meldingen er juist *wel* toe moeten leiden dat een taak meerdere keren ontwaakt, dan moet het argument `xClearCountOnExit` de waarde `pdFALSE` krijgen. In dat geval blokkeert de ontvangende taak zolang de waarde van het notificatiewoord nul is (net als eerst). Elke melding aan de taak incrementeert de waarde van het task notificatie-woord, maar het wordt slechts met *één* verlaagd voor elke aanroep naar `ulTaskNotifyTake()`. Als er dus bijvoorbeeld vier meldingen zijn geweest voordat de ontvangende taak `ulTaskNotifyTake()` aanroept, dan retourneert de aanroep onmiddellijk zonder te blokkeren totdat de waarde van het notificatie-woord weer tot nul is verlaagd. Met andere woorden, de ontvangende taak moet vier aanroepen naar `ulTaskNotifyTake()` doen voordat het notificatiewoord helemaal tot nul is verlaagd.

Notificatie geven

De ontvangende taak gebruikt dus de functie `ulTaskNotifyTake()`; de aanroepende taak gebruikt `xTaskNotifyGive()`:

```
BaseType_t xTaskNotifyGive(TaskHandle_t xTaskToNotify);
```

Deze functie heeft alleen de handle nodig van de taak die een melding

Listing 1. Het demoprogramma tasknfy1.ino voor xTaskNotifyGive() en ulTaskNotifyTake().

```
0001: // tasknfy1.ino
0002:
0003: #define GPIO_LED        12
0004:
0005: static TaskHandle_t htask1;
0006:
0007: static void task1(void *arg) {
0008:  uint32_t rv;
0009:
0010:  for (;;) {
0011:    rv = ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
0012:    digitalWrite(GPIO_LED, digitalRead(GPIO_LED)^HIGH);
0013:    printf("Task notified: rv=%u\n", unsigned(rv));
0014:  }
0015: }
0016:
0017: void setup() {
0018:  int app_cpu = 0;
0019:  BaseType_t rc;
0020:
0021:  app_cpu = xPortGetCoreID();
0022:  pinMode(GPIO_LED, OUTPUT);
0023:  digitalWrite(GPIO_LED, LOW);
0024:
0025:  delay(2000);    // Allow USB to connect
0026:   printf("tasknfy1.ino:\n");
0027:
0028:  rc = xTaskCreatePinnedToCore(
0029:    task1,        // Task function
0030:    "task1",      // Name
0031:    3000,         // Stack size
0032:    nullptr,      // Parameters
0033:    1,            // Priority
0034:    &htask1,      // handle
0035:    app_cpu       // CPU
0036:  );
0037:  assert(rc == pdPASS);
0038: }
0039:
0040: void loop() {
0041:  delay(1000);
0042:  xTaskNotifyGive(htask1);
0043: }
```

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.nl.

moet krijgen en retourneert altijd `pdTRUE` (volgens de FreeRTOS-handleiding is `xTaskNotifyGive()` gedefinieerd als een macro).

Demo

Een heel eenvoudige demo is te zien in **listing 1** [1]. Dit programma maakt gebruik van de seriële monitor, zodat u de gerapporteerde waarde van het notificatie-woord terug kunt zien. In **figuur 1** ziet u het schema voor het aansluiten van een optionele LED en in **figuur 2** de opbouw op een breadboard. Vrijwel elk ESP32-ontwikkelboard waarop GPIO12 beschikbaar is, kan hier worden gebruikt.
Het programma tasknfy1.ino gebruikt de `loopTask()` van Arduino om `task1()` (regel 42) herhaaldelijk een notificatie te sturen, met tussenpozen van een seconde. De functie `task1()` blokkeert dan (regel 11) tot de notificatie arriveert. Als dat is gebeurd, schakelt hij de LED in (regel 12). De functie `task1()` wordt geadresseerd met behulp van zijn handle `htask1` (regel 42). Die handle is aangemaakt in de functie `setup()` op regel 34.

```
0040: void loop() {
0041:   delay(1000);
0042:   xTaskNotifyGive(htask1);
0043: }
```

De code van `task1` is bijna net zo triviaal en draait in een eindeloze lus. Hij blijft in regel 11 wachten bij de aanroep van `ulTaskNotifyTake()` tot hij een notificatie ontvangt:

```
0007: static void task1(void *arg) {
0008:  uint32_t rv;
0009:
0010:  for (;;) {
0011:   rv = ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
0012:   digitalWrite(GPIO_LED, digitalRead(GPIO_LED)^HIGH);
0013:   printf("Task notified: rv=%u\n", unsigned(rv));
0014:   }
0015: }
```

De verdere uitvoering van `task1` blijft geblokkeerd totdat het task notificatie-event aankomt. De waarde die wordt toegewezen aan `rv` (regel 11) is de waarde van het notificatie-woord *voordat* het wordt gewist (vanwege het argument `pdTRUE`). Merk op hoe elegant en eenvoudig dit is: er zijn geen andere semafoorobjecten in het spel en dus ook geen extra handles.
De sessie moet er in de seriële monitor zo uitzien:

```
tasknfy1.ino:
Task notified: rv=1
Task notified: rv=1
...
```

Als u een melding wilt sturen vanuit een ISR, moet u de functie `xTaskNotifyGiveFromISR()` gebruiken. Het FreeRTOS-achtervoegsel 'FromISR' maakt het mogelijk om veilig te werken vanuit een ISR, iets waar vaak een speciale afhandeling voor nodig is.

```
void vTaskNotifyGiveFromISR(
  TaskHandle_t xTaskToNotify,
   BaseType_t *pxHigherPriorityTaskWoken
);
```

Het tweede argument is het adres van de ISR-wakeup-flag, die aangeeft of de scheduler al dan niet moet worden aangeroepen (gebruik `NULL`/`nullptr` wanneer dat niet nodig is). Deze task notificatie-functie is een zeer handige manier voor een ISR-routine om een notificatie naar een taak te sturen.

Geavanceerde notificatie

In dit artikel hebben we alleen het eenvoudigste tweetal functies xTaskNotifyGive() (of xTaskNotifyGiveFromISR()) en ulTaskNotifyTake() besproken.
Er zijn nog andere, meer geavanceerde, functies beschikbaar, waaronder de volgende:

- `xTaskNotify()` (of `xTaskNotifyFromISR()`)
- `xTaskNotifyAndQuery()` (of `xTaskNotifyAndQueryFromISR()`)
- `xTaskNotifyWait()`
- `xTaskNotifyStateClear()`
- `xTaskNotifyValueClear()`

Deze functies werken op een vergelijkbare manier, maar ondersteunen de verwerking van events op bitniveau. Ze bieden ook meer selectieve vormen om events te testen en te resetten. U kunt hierover lezen in de FreeRTOS-referentiehandleiding [2] of online [3].

Conclusie

De FreeRTOS task notificatie-API biedt de applicatie-ontwikkelaar een eenvoudig event notificatie-mechanisme. Dit is vooral nuttig voor het implementeren van notificaties vanuit ISR's, omdat het de interruptverwerking kort houdt, terwijl het zware werk in de aangesproken taak kan worden uitgevoerd. Zelfs wanneer er geen interrupts worden gebruikt, spaart dit mechanisme RAM en vereenvoudigt het het gebruik, doordat er geen extra synchronisatieobjecten zoals semaforen nodig zijn.

200454-04

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

WEBLINKS

[1] **Code van tasknfy.ino**: https://github.com/ve3wwg/FreeRTOS_for_ESP32/blob/master/tasknfy1/tasknfy1.ino
[2] **FreeRTOS documentatie**: www.freertos.org/Documentation/RTOS_book.html
[3] **FreeRTOS documentatie voor Task Notification**: www.freertos.org/RTOS-task-notification-API.html