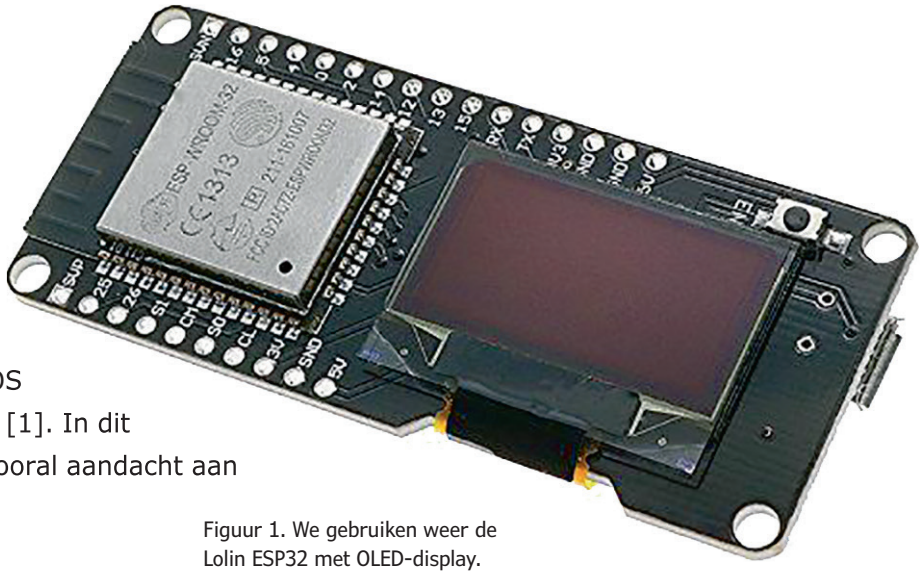
Multitasking met de ESP32 (2)
taakprioriteiten

Warren Gay (Canada)



In microcontrollerprojecten worden ontwikkelaars vaak geconfronteerd met het probleem dat er veel processortaken tegelijk moeten worden uitgevoerd. ESP32 en Arduino IDE maken het programmeren van meerdere taken eenvoudig, omdat het populaire FreeRTOS al is geïntegreerd in de kernbibliotheken [1]. In dit tweede deel van de serie besteden we vooral aandacht aan taakprioriteiten.

Figuur 1. We gebruiken weer de Lolin ESP32 met OLED-display.

In de ESP32-implementatie van de FreeRTOS scheduler worden taken uitgevoerd volgens hun prioriteit. De prioriteit wordt toegekend wanneer de taak wordt gecreëerd, maar kan later worden gewijzigd. Taken met een hoge prioriteit komen als eerste aan de beurt voor de CPU, en taken met een prioriteit 0 als laatste. Taakprioriteiten zijn een bekend begrip, maar de real-time scheduler van FreeRTOS werkt toch anders dan u misschien gewend bent van Linux of Windows. In dit artikel wordt aan de hand van een demonstratie het verschil onderzocht.
De ESP32-implementatie onderscheidt maximaal 25 prioriteitsniveaus, variërend van 0 tot 24. De Arduino-functies `setup()` en `loop()` draaien standaard op prioriteitsniveau 1 (zoals we in het vorige deel hebben gezien, worden deze twee functies uitgevoerd door dezelfde hoofdtaak [1]).

Vive la différence

Hoe anders kan de taakplanning zijn? Op bijvoorbeeld een Linux-systeem heeft de prioriteit invloed op de relatieve urgentie van het proces of de thread. Zelfs een proces met lage prioriteit krijgt wat CPU-tijd; normaal gesproken draait het dus langzamer. Maar het proces wordt uiteindelijk altijd uitgevoerd. En daar ligt het verschil.
In een real-time systeem zoals FreeRTOS garandeert de scheduler **niet** dat taken met een lagere prioriteit ooit zullen worden uitgevoerd. Als u bijvoorbeeld taken hebt met een prioriteit van 9 die altijd *klaar zijn om uit te voeren*, dan zal er geen taak met prioriteit 8 of lager worden ingepland op diezelfde CPU. Met andere woorden: de taken met prioriteit 9 zullen de taken met een lagere prioriteit volledig verdringen.

Klaar om uit te voeren

Het is belangrijk om te begrijpen wat ‘klaar’ betekent voor FreeRTOS. Een taak is *klaar* als hij niet geblokkeerd is omdat hij op iets staat te wachten. Hij kan bijvoorbeeld wachten op een event, een bericht dat in een *queue* wordt geplaatst of een *mutex* die ontgrendeld moet worden (het begrip *mutex* komt later in deze serie nog aan de orde). Een taak die klaar is om uit te voeren wordt in de *ready list* van de planner ingevoegd op basis van zijn prioriteit en wordt uitgevoerd wanneer hij aan de beurt is. Omdat de lijst op prioriteit gesorteerd is, komen de taken met de hoogste prioriteit als eerste aan de beurt.
Taken met gelijke prioriteit worden gepland met behulp van een *Round-Robin*-strategie. Drie ‘ready’ taken met prioriteit 9 (a, b en c) worden om beurten uitgevoerd:

- taak9a
- taak9b
- taak9c
- taak9a
- taak9b
- enzovoort

Dit blijft doorgaan, zolang ze niet geblokkeerd raken. Alleen een taak met een hogere prioriteit kan dit doorbreken. Bijvoorbeeld de ESP32-taak *idc1* (voor CPU 1) heeft een hoge prioriteit en kan uw taken met prioriteit 9 verdringen. Zodra de *idc1*-taak weer *niet klaar* is worden de prioriteit-9-taken hervat waar ze gebleven waren. Enkele voorbeelden van *niet klaar*:

- slaap of vertraging (wachtend op een timer);
- wachtend op een mutex of semafoor;
- wachtend op een bericht uit een lege wachtrij;
- wachtend om een bericht in te voegen in een volle wachtrij;
- wachtend op een FreeRTOS-event of event-groep;
- wachtend op de voltooiing van een I/O;
- suspended (hetzij door de taak zelf, hetzij door een andere taak).

Eén van de manieren om *geblokkeerd* te raken is wachten op een bericht uit een lege wachtrij. Als de wachtrij leeg is, is er

niets te doen voor de taak. De planner verwijdert die taak dan uit de lijst met gereedstaande taken en zoekt naar andere taken om uit te voeren. Alleen taken die op de 'ready'-lijst staan, komen in aanmerking. Als er geen taken worden gevonden, wordt in plaats daarvan de *idle*-taak gedraaid.
Aanroepen van de functie `taskYIELD()` maakt dus niet dat een taak niet meer *ready* is. Als het tijdslot van een taak op is, of als hij vrijwillig stopt door `taskYIELD()` aan te roepen, gaat de controle terug naar de FreeRTOS scheduler, zodat deze een andere taak kan kiezen om te draaien voor de volgende slice. In deze gevallen wordt de taak niet geblokkeerd: hij blijft klaar om uitgevoerd te worden als hij de volgende keer weer aan de beurt komt.

ESP-IDF FreeRTOS SMP wijzigingen

FreeRTOS is ontworpen voor microcontrollers met één CPU. Omdat de ESP32 beschikt over twee CPU's (behalve de ESP32-S2), heeft Espressif de scheduler aangepast. De volgende ESP32-CPU's zijn aanwezig:

- CPU 0 bekend als de PRO_CPU (Protocol CPU);
- CPU 1 bekend als de APP_CPU (Application CPU);

Espressif zegt dat de "twee kernen in de praktijk identiek zijn en hetzelfde geheugen delen".
Om symmetrische multiprocessing (SMP) te ondersteunen, stellen ze dat de "scheduler taken zal overslaan bij het uitvoeren van de Round-Robin-scheduling tussen meerdere taken in de ready-toestand die dezelfde prioriteit hebben". Dit komt door de beperking van het gebruik van een 'ready'-lijst die is ontworpen voor een enkele CPU, op een platform met twee CPU's [2]. Het probleem waar ze mee te maken hadden was dat wanneer een CPU een context switch zou moeten uitvoeren (om aan de volgende gereedstaande taak te beginnen), de CPU slechts één lijst met gereedstaande taken had om te doorzoeken. Dus als de lijst-index wijst naar een gereedstaande taak voor de andere CPU, dan moet die taak worden overgeslagen, net zolang tot een taak voor de juiste CPU wordt gevonden. Dat kan de planning van de Round-Robin verstoren.
Het komt er voor de ontwikkelaar op neer dat de Round-Robin-scheduling niet helemaal eerlijk is in de dual-CPU ESP32. Voor veel projecten zal dat niet opvallen, maar als het toch problematisch wordt, zijn er manieren om er 'omheen' te programmeren. Wees u daarvan bewust bij de planning van de taken.

Demonstratie

Voor de Lolin ESP32 OLED Display Module is een Arduino-demonstratieprogramma beschikbaar (**figuur 1**). Door een paar macro's in het programma te veranderen, kunt u de taakprioriteiten van vier verschillende taken binnen het programma wijzigen. Het programma is ontworpen om drie wormen weer te geven, die horizontaal heen en weer bewegen op OLED-display. Elk van de wormen zal alleen bewegen als de taak die hem aanstuurt CPU-tijd krijgt.
Elke worm wordt aangestuurd door een taak die CPU-tijd verbruikt en vervolgens een bericht naar de vierde taak stuurt. Deze vierde taak zorgt voor de weergave van de worm.
De code voor het tekenen en beheren van de toestand van de worm is gedefinieerd in de klasse `InchWorm` (hier niet weergegeven). Voor dit artikel richten we ons simpelweg op het effect van de methode `InchWorm::draw()` voor elke worm. Elke instantie van de klasse InchWorm beheert zijn eigen toestand en beweging. De weergave- en worm-instanties worden in het programma als volgt gedeclareerd:

```
static Display oled;
static InchWorm worm1(oled,1);
static InchWorm worm2(oled,2);
static InchWorm worm3(oled,3);
```

Elke worm krijgt een C++-referentie (vergelijkbaar met een pointer in C) naar de weergaveklasse als eerste argument en het nummer van de worm als tweede. De referentie naar het display maakt toekomstige uitbreiding mogelijk, zoals ondersteuning van meerdere schermen. Het wormnummer bepaalt waar op de OLED de worm wordt weergegeven (1, 2 en 3 zijn respectievelijk de bovenste, de middelste en de onderste regel). De taak achter elke worm bestaat gewoon uit een lus die CPU-tijd verspilt en een aanroep om een bericht te verzenden:

```
void worm_task(void *arg) {
  InchWorm *worm = (InchWorm*)arg;

  for (;;) {
    for ( int x=0; x<800000; ++x )
      __asm__ __volatile__("nop");
    xQueueSendToBack(qh,&worm,0);
    // vTaskDelay(10);
  }
}
```

Het is belangrijk om de aanroep van `vTaskDelay()` voorlopig uitgecommentarieerd te laten. Die gaan we gebruiken in een volgend experiment.
Dezelfde taakfunctie wordt gebruikt voor alle drie de inchworm-taken, waarbij het argument `arg` aangeeft welke instantie van de worm we willen bewegen. Het adres van de worm wordt geconverteerd van een void-pointer en opgeslagen in de lokale variabele `worm`. Hij wordt alleen binnen deze taak gebruikt om hem als een bericht naar de displaytaak (hoofdtaak) te verzenden, om aan te geven welke worm moet worden bewogen. Als `xQueSendToBack()` wordt aangeroepen in deze demonstratie, vullen we 0 in voor de time-to-wait-parameter (het derde argument). Dat geeft FreeRTOS opdracht om het bericht naar de wachtrij te sturen als dat mogelijk is, maar onmiddellijk op te geven als de wachtrij vol is. Dat is met opzet zo gedaan, want we willen niet dat onze wormtaak blokkeert als de wachtrij vol raakt. De taak mag bij deze demonstratie de CPU niet vrijgeven, zodat hij de CPU echt kan monopoliseren.
De buitenste `for`-lus zorgt ervoor dat de taak altijd blijft doorlopen. De binnenste `for`-lus dient alleen om CPU-tijd te verstoken en voert 800.000 keer een *no operation* (`nop`) operatie uit. Het keyword `__volatile__` voorkomt dat de compiler deze lus wegoptimaliseert. Na afloop van de tijdverspillende lus sturen we het adres van de te bewegen worm naar de wachtrij die met de handle `qh` wordt geïdentificeerd. Als het bericht wordt ontvangen door de displaytaak, zal die ervoor zorgen dat onze worm wordt verplaatst en weergegeven.
De Arduino-hoofdtaak `loop()` wordt gebruikt als displaytaak om de worm te bewegen:

```
void loop() {
  InchWorm *worm = nullptr;
```

```
  if ( xQueueReceive(qh,&worm,portMAX_DELAY) )
    worm->draw();
}
```

Deze lus blokkeert zichzelf totdat één van de andere taken het adres van een te tekenen worm stuurt. Zodra die class-pointer is ontvangen, wordt de methode `InchWorm::draw()` aangeroepen om de worm te tekenen en te verplaatsen.
In de functie `setup()` in **listing 1** zien we hoe de drie wormtaken en de wachtrij worden gecreëerd.

Veranderen van de prioriteit

In FreeRTOS kan een taak zijn eigen prioriteit of die van een andere taak wijzigen met de functie `vTaskPrioritySet()`. Standaard draait de taak die `setup()` en `loop()` aanroept op prioriteitsniveau 1 (deze functies worden door dezelfde hoofdtaak aangeroepen). Voor deze demonstratie moet die prioriteit hoger zijn dan die van de drie wormtaken. De functie `setup()` wijzigt de prioriteit van zijn eigen taak als volgt:

Listing 1. De setup()-functie.

```
void setup() {
  TaskHandle_t h = xTaskGetCurrentTaskHandle();

  app_cpu = xPortGetCoreID(); // Which CPU?
  oled.init();
  vTaskPrioritySet(h,MAIN_TASK_PRIORITY);
  qh = xQueueCreate(4,sizeof(InchWorm*));

  // Draw at least one worm each:
  worm1.draw();
  worm2.draw();
  worm3.draw();

  xTaskCreatePinnedToCore(
    worm_task,  // Function
    “worm1”,    // Task name
    3000,       // Stack size
    &worm1,     // Argument
    WORM1_TASK_PRIORITY,
    nullptr,    // No handle returned
    app_cpu);

  xTaskCreatePinnedToCore(
    worm_task,  // Function
    “worm2”,    // Task name
    3000,       // Stack size
    &worm2,     // Argument
    WORM2_TASK_PRIORITY,
    nullptr,    // No handle returned
    app_cpu);

  xTaskCreatePinnedToCore(
    worm_task,  // Function
    “worm3”,    // Task name
    3000,       // Stack size
    &worm3,     // Argument
    WORM3_TASK_PRIORITY,
    nullptr,    // No handle returned
    app_cpu);
}
```

```
static int app_cpu = 0; // Updated by setup()
...
void setup() {
  TaskHandle_t h = xTaskGetCurrentTaskHandle();

  app_cpu = xPortGetCoreID(); // Which CPU?
  ...
  vTaskPrioritySet(h,MAIN_TASK_PRIORITY);
```

Zoals u ziet, haalt `setup()` zijn eigen taakhandle op door `xTaskGetCurrentTaskHandle()` aan te roepen en op te slaan in `h`. Door de prioriteit van de hoofdtaak te wijzigen met `vTaskPrioritySet()`, wordt ook de taakprioriteit voor `loop()` beïnvloed. Zo kunnen taakprioriteiten worden aangepast.

In het eerste experiment krijgen de wormtaken de prioriteiten 9, 8 en 7 toegewezen. Dat betekent dat de display(hoofd)taak prioriteit 9 of hoger moet krijgen (we gebruiken 10). Als dat niet gebeurt, zal de hoofdtaak `loop()` verhongeren en niet in staat zijn om de inchwormen te animeren.

Welke CPU?

In het getoonde deel van `setup()` werd een ESP32 API-functie met de naam `xPortGetCoreID()` gebruikt om uit te vinden op welke CPU de applicatie draait. Deze informatie wordt toegewezen aan de statische variabele `app_cpu`, zodat de code weet voor welke CPU hij nieuwe taken moet maken. Voor de dual-core ESP32 zal de waarde van `app_cpu` 1 zijn (draaien op CPU 1 in een dual-core configuratie). Voor single-CPU-platformen wordt `app_cpu` op 0 gezet. Door het op deze manier te coderen kan de code zowel draaien op platforms met één als met twee CPU’s. Deze specifieke demonstratie zal niet goed functioneren op een single-CPU-platform vanwege de manier waarop de CPU wordt gemonopoliseerd. Dat zou de watchdog-timer activeren en een reset veroorzaken. Maar de techniek van het gebruik van `xPortGetCoreID()` illustreert wel hoe portabiliteit voor andere toepassingen kan worden bereikt.

Democonfiguratie

De broncode voor deze demonstratie is beschikbaar op [3]. Bovenaan het demonstratieprogramma staan de macrodefinities, waarmee elk experiment wordt geconfigureerd:

```
// Wormtaak prioriteiten
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 8
#define WORM3_TASK_PRIORITY 7

// loop() moet de hoogste prioriteit hebben
#define MAIN_TASK_PRIORITY  10
```

Laat die in eerste instantie zoals getoond bij het eerste experiment.

Aangepast OLED-display

Als u de aanbevolen Lolin ESP32 met zijn ingebouwde OLED niet gebruikt, kunt u hier uw aangepaste weergave-instellingen herconfigureren:

```
Display(
   int width=128,
   int height=64,
   int addr=0x3C,
   int sda=5,
   int scl=4);
```

Als uw instellingen correct zijn geconfigureerd, moet de OLED bij de initialisatie van het programma meteen wit worden. Anders moet u de verbindingen en instellingen opnieuw controleren.

Demonstratie 1

Met de gedownloade code kunt u gewoon de applicatie compileren, flashen en uitvoeren. Uw OLED moet onmiddellijk wit worden, met drie zwarte inchwormen erop (zie **figuur 2**).
De configuratie voor dit experiment is:

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 8
#define WORM3_TASK_PRIORITY 7
#define MAIN_TASK_PRIORITY  10
```

Deze configuratie zorgt ervoor dat de bovenste worm zich een weg baant over de bovenste regel, terwijl de onderste twee stil blijven staan. De vraag is: waarom bewegen de middelste en onderste wormen niet?

```
                    _¯_
_¯_
_¯_
```

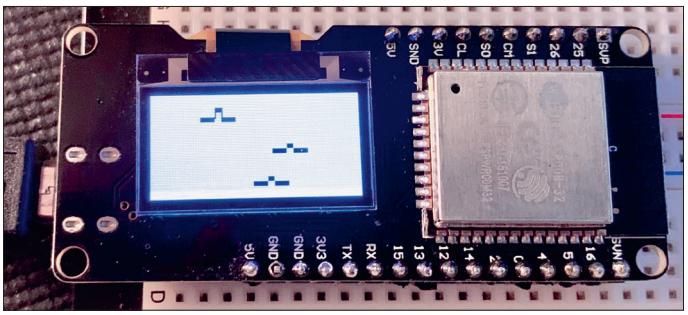
Bedenk dat we de hoofdtaak prioriteit 10 hebben gegeven. Die heeft dus de hoogste prioriteit van al onze taken. De eerste worm, die op de bovenste regel van de OLED wordt weergegeven, kon vooruitgaan omdat het de enige CPU-verslindende taak was die kon worden uitgevoerd. Deze taak met prioriteit 9 kan draaien omdat de displaytaak die op niveau 10 draait naar de OLED schrijft en dan **wacht** tot er berichten in de wachtrij komen (hij wordt dus geblokkeerd). Wanneer de displaytaak is geblokkeerd, kunnen taken met een lagere prioriteit worden ingepland. De taken met prioriteit 8 en 7 (voor middelste en onderste wormen) krijgen geen CPU-tijd toegewezen en worden nooit uitgevoerd omdat de taak met prioriteit 9 de CPU volledig monopoliseert. Zo werkt de real-time planning binnen FreeRTOS. In tegenstelling tot Linux of Windows krijgen taken met een lagere prioriteit geen kans om te draaien.

Demonstratie 2

Bij het tweede experiment veranderen we de configuratie en geven we de drie wormen allemaal dezelfde prioriteit. De hoofd(display)taak laten we op prioriteit 10 staan.

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 9
#define WORM3_TASK_PRIORITY 9
#define MAIN_TASK_PRIORITY  10
```

Wat zag u, toen u de code compileerde en de ESP32 opnieuw flashte?



Figuur 2. Als de taken worden uitgevoerd, bewegen de demowormen.

```
          _¯_
          _¯_
          _¯_
```

De drie wormen lopen nu allemaal (bijna) even snel. Als de demonstratie lang genoeg doorgaat, kunnen sommige wormen een kleine voorsprong opbouwen.

Demonstratie 3

Bij dit experiment, krijgen de drie wormen weer allemaal dezelfde prioriteit (net als bij demonstratie 2), maar nu krijgt ook de hoofdtaak dezelfde prioriteit. Ik gebruik prioriteit 9 voor alle taken:

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 9
#define WORM3_TASK_PRIORITY 9
#define MAIN_TASK_PRIORITY  9
```

Wat zag u deze keer? Was er een verschil?

```
      _¯_
            _¯_
                        _¯_
```

Als ik dit doe, lijkt de onderste worm de meeste CPU-tijd te krijgen (hij beweegt dus het snelst). De bovenste worm beweegt het langzaamst. Hier blijkt de oneerlijkheid van de Round-Robin-scheduling waar Espressif op wijst. In het ideale geval zou de displaytaak alleen een beetje CPU moeten stelen tijdens het tekenen van de worm. De resterende CPU-tijd zou gelijkelijk moeten worden verdeeld over de drie andere taken.
Toch zien we dat de planning onevenwichtig is. Beide CPU's reageren op timer- en andere interrupts. De gebrekkige scheduler is verantwoordelijk voor het verstoren van de eerlijkheid van de Round-Robin-scheduling.

Demonstratie 4

Tot nu toe hebben de wormtaken bij elke demonstratie zoveel CPU-tijd gebruikt als ze maar konden krijgen. Hoe verandert het gedrag als we een kleine (blokkerende) vertraging invoeren binnen de lus? We zetten de hoofdtaak weer op prioriteit 10, en de wormtaken op prioriteit 9, 8 en 7:

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 8
#define WORM3_TASK_PRIORITY 7
#define MAIN_TASK_PRIORITY  10
```

We halen nu het commentaarteken weg bij `vTaskDelay()`, zodat de taken er nu zo uitzien:

```
void worm_task(void *arg) {
  InchWorm *worm = (InchWorm*)arg;

  for (;;) {
    ...
    for ( int x=0; x<800000; ++x )
      __asm__ __volatile__("nop");
    xQueueSendToBack(qh,&worm,0);
    vTaskDelay(10); // Uncommented
  }
}
```

Nu zal elke wormtaak CPU consumeren, proberen een worm in de wachtrij te zetten en dan 10 milliseconden blokkeren. Compileer dit voorbeeld, flash het en voer het uit. Wat ziet u? De bovenste worm zal het snelst bewegen en de onderste worm het langzaamst. De bovenste worm met prioriteit 9 krijgt de eerste kans om de CPU te gebruiken vanwege zijn hoge prioriteit (terwijl de displaytaak wordt geblokkeerd). Als de wormtaak wordt geblokkeerd in de call naar `vTaskDelay(10)`, krijgt de volgende taak met lagere prioriteit (de middelste worm) de kans om wat CPU te verbruiken en roept uiteindelijk ook `vTaskDelay(10)` aan. Dan krijg zelfs de taak met de laagste prioriteit de kans om wat CPU-tijd te gebruiken. Zo wordt de aandacht van de CPU verdeeld over de verschillende niveaus. Maar de taken met prioriteit 8 en 7 worden wel onderbroken als de taak met prioriteit 9 weer klaar is. Daarom beweegt de bovenste worm het snelst. De middelste worm heeft betere kansen dan de onderste, dus hij zal sneller zijn dan de onderste.

Meer experimenten

Wat gebeurt er als we de `vTaskDelay()` tijd veel langer maken dan 10 milliseconden? Probeer het u voor te stellen en voer het dan uit. Waarom kreeg u dit resultaat? Wat gebeurt er als u de vertragingstijd verkort tot 1 milliseconde? Deze experimenten laten we over aan de lezer.

Prioriteitsconfiguratie

Hoewel we het gebruik van de interrupts in de ESP32 nog niet hebben behandeld, moet u zich bewust zijn van het header-bestand *FreeRTOSConfig.h*, dat de prioriteiten voor het platform configureert. Het is hier te vinden:

```
$IDF_PATH/components/freertos/include/freertos/
FreeRTOSConfig.h
```

De header definieert de volgende prioriteiten. We tonen de gecompileerde waarden:

```
configMAX_PRIORITIES = 25
configKERNEL_INTERRUPT_PRIORITY = 1
configMAX_SYSCALL_INTERRUPT_PRIORITY = 3
```

De eerste macro definieert het aantal beschikbare prioriteiten. Dit betekent dat geldige prioriteitsnummers variëren van 0 tot 24.
De tweede macro definieert de prioriteit die de kernel zelf gebruikt voor *interrupts*. De derde macro, die de hoogste prioriteit geeft aan kernel-interrupts hangt daarmee samen. Elke FreeRTOS API-call die vanuit een Interrupt Service Routine (ISR) wordt gedaan**,** mag alleen FreeRTOS API-functies aanroepen met namen die eindigen op `FromISR()`. Verder kunnen deze functies, met de getoonde waarden, alleen worden opgeroepen vanaf de interruptietaakprioriteiten 1 tot en met 3. Als er geen `FromISR()`–calls worden gedaan, kan de ISR vrijelijk werken op de prioriteiten 4 tot en met 24.

Samenvatting

Wat kunnen we concluderen uit deze experimenten? Wat misschien een eenvoudig concept van prioriteit leek, was toch niet zo eenvoudig. Het gevolg is dat u, als u uw taakprioriteiten niet goed hebt gepland, voor verrassingen kunt komen te staan. Sommige taken krijgen misschien helemaal geen CPU-tijd toebedeeld. We hebben het nog niet gehad over watchdog-timers, maar die kunnen hier ook door worden beïnvloed. Als bijvoorbeeld de watchdog-timer in CPU 0 wordt geactiveerd, dan wordt uw ESP32 gereset en opnieuw opgestart.
Voor de dual-core ESP32 is er het bijkomende probleem dat de Round-Robin-scheduling op gelijk prioriteitsniveau mogelijk geen eerlijke verdeling van CPU-tijd geeft. Dat kan in sommige toepassingen een probleem zijn en in andere niet. Het ligt aan de aard van uw ‘systeem’.
Voor veel toepassingen kunt u gewoon taken aanmaken die op prioriteit 1 draaien. Dat is de prioriteit die is geconfigureerd voor de Arduino-taken `setup()` en `loop()`. Taken met een hogere prioriteit kunnen veilig worden gebruikt als ze blokkeren in een wachtrij, semafoor of een ander event. Als een taak wordt geblokkeerd of opgeschort, wordt de CPU gedeeld met andere taken met gelijke of lagere prioriteit. Een applicatie met goed geconfigureerde taakprioriteiten werkt als een goed geoliede machine. ⏮

(191195-04)

IN DE STORE

→ Lolin ESP32 OLED-displaymodule
www.elektor.nl/lolin-esp32-oled-module-with-wifi

Weblinks

[1] Multitasking met de ESP32, Elektor januari/februari 2020: www.elektormagazine.nl/190182-04

[2] Symmetric Multiprocessing: https://thc420.xyz/esp-idf/file/docs/en/api-guides/freertos-smp.rst.html

[3] Broncode van dit project: https://github.com/ve3wwg/esp32_freertos/blob/master/priority-worms1/priority-worms1.ino