Multitasking met de ESP32

taakgeoriënteerd programmeren met FreeRTOS en de Arduino IDE

Wanneer een microcontroller als hart van een project wordt gebruikt, hebben ontwikkelaars vaak het probleem dat er meer dan één taak tegelijk moet worden uitgevoerd: het scannen van sensoren, het aansturen van actuatoren, het visualiseren van apparaatstatussen en/of het wachten op invoer van een menselijke gebruiker. Gelukkig kan dit probleem op een uiterst elegante manier worden opgelost met taakgeoriënteerd programmeren op basis van 'lichtgewicht' embedded besturingssystemen. FreeRTOS is een veelgebruikt open source-besturingssysteem, beschikbaar voor veel microcontroller-platforms. Het terecht populaire duo ESP32 en Arduino IDE maken het wel bijzonder gemakkelijk om FreeRTOS te gebruiken voor het programmeren van taken, omdat het al geïntegreerd is in de kernbibliotheken. Mogelijkerwijs hebt u FreeRTOS altijd al gebruikt zonder het te beseffen!

Warren Gay (Canada)

Het ESP32-platform van Espressif biedt opwindende mogelijkheden voor maker-projecten. De hardwaremogelijkheden, de uitgebreide software en de betaalbaarheid maken het voor velen tot een 'must have'.
Espressif levert zowel een Arduino-compatibele als een 'eigen' ontwikkelomgeving. De eigen omgeving heet officieel ESP-IDF (*Espressif IoT Development Framework*). Voor dit artikel gebruiken we echter de bekende Arduino-omgeving. Het grootste deel van de API (*Application Programming Interface*) is voor beide omgevingen beschikbaar [1]. Sommige van onze lezers weten wellicht dat Espressif gebruik maakt van het populaire open source embedded operating system FreeRTOS om de ESP32-bibliotheekfuncties voor WiFi, Bluetooth en nog veel meer features van de microcontroller te implementeren. In dit artikel zullen we zien dat FreeRTOS [2] en taakgeoriënteerd programmeren (*task programming*) ook gebruikt worden voor eenvoudige Arduino `setup()` en `loop()` functies.

Eerst behandelen we een eenvoudig ESP32-demoproject [3]. Daarna nemen we een kijkje onder de motorkap; later zullen we onze eigen taken voor de demo schrijven.

De toepassing

Voor deze aflevering ontwikkelen we een applicatie die de analoog/digitaal-omzetter (ADC) uitleest en zo de stand van een potmeter in de vorm van een staafdiagramvorm weergeeft op een OLED. Daarnaast dimmen we zo een LED met behulp van pulsbreedtemodulatie (PWM).
Voor dit artikel gebruiken we het Lolin ESP32-board met geïntegreerd OLED-display (**figuur 1**) [4]. Als u een ander ESP32-board zonder SSD1306-compatibele OLED hebt, kunt u de displayroutines in het programma uitschakelen en alleen de helderheid van de LED regelen.
De LED en de potentiometer worden aangesloten volgens het schema van **figuur 2**. Als het programma actief is, wordt door verdraaien van de potentiometer de spanning op ingang GPIO36 gevarieerd. Deze spanning wordt ingelezen als een 12-bit waarde

(dus van 0 tot 4095). Als de potmeter helemaal rechtsom wordt gedraaid, wordt de waarde 4095 geretourneerd en zal de LED met maximale helderheid branden. Als precies het omgekeerde gebeurt, moet u de 'buitenste' aansluitingen van de potmeter omwisselen. De middelste aansluiting is de loper waarop een variabele spanning van 0 V tot +3,3 V staat. Let erop dat u de potmeter niet aansluit op de +5V-voeding, omdat dan de maximale GPIO-ingangsspanning kan worden overschreden.

Programmaconfiguratie

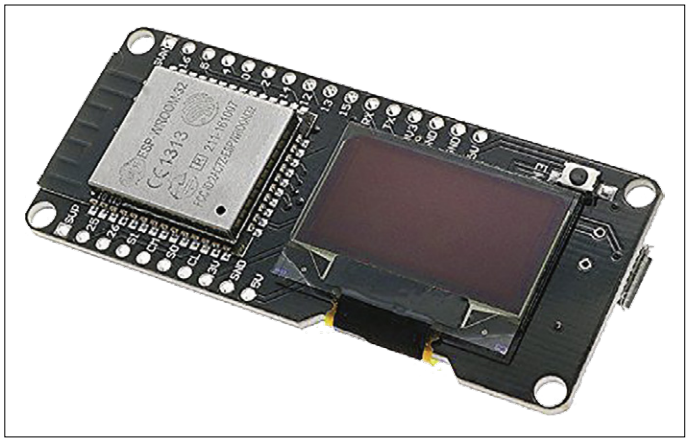
Nu is de software aan de beurt. In het programma zijn C-macro's gedefinieerd zodat u de configuratie gemakkelijk zelf kunt aanpassen. U ziet deze in **listing 1**. Maak de macro `CFG_OLED` nul als u geen display gebruikt (op deze manier worden het adres en de I²C-macro's genegeerd).
De waarde van macro `CFG_ADC_GPIO` is zo gekozen dat ADC1 op kanaal 0 (GPIO36) wordt gebruikt. Tenslotte is de LED via `CFG_LED_GPIO` geconfigureerd voor GPIO13.
De volledige broncode is beschikbaar op github in de *freertos-tasks1*-subdirectory [3].

Initialisatie

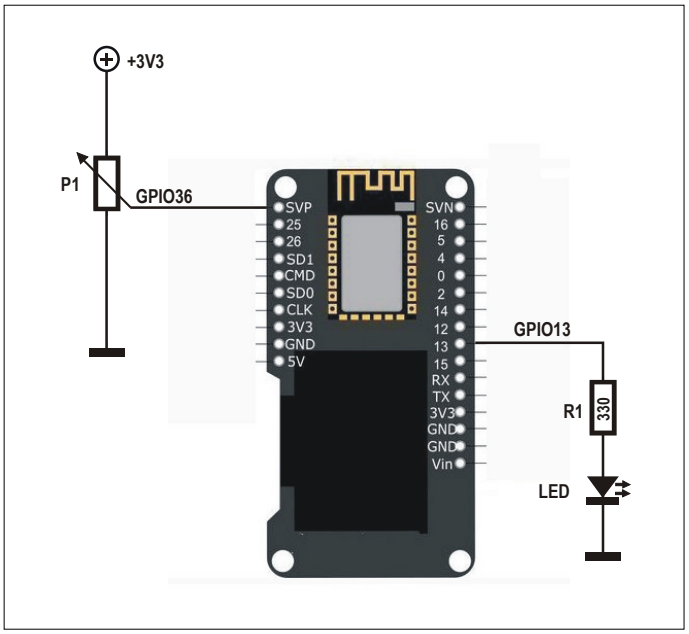
Wanneer we de taken even buiten beschouwing laten, ziet de `setup()`-functie voor onze applicatie er uit als geschetst in **listing 2**. Dit is eigenlijk een typische initialisatie binnen het Arduino-framework.
Het display wordt alleen geïnitialiseerd als het bij de compilatie is meegenomen (dus geconfigureerd is). Daarna wordt met behulp van de Arduino-routine `analogReadResolution()` de ADC geconfigureerd om 12-bit waarden te lezen. De functie `analogSetAttenuation()` wordt aangeroepen om een waarde van 0 tot +3,3 V in te lezen. De ADC bevat een interne versterker, dus is het belangrijk om deze voor het juiste bereik te configureren.

Listing 1. Configuratiemogelijkheden.

```
// Set to zero if NOT using SSD1306
#define CFG_OLED            1

// I2C address of SSD1306 display
#define CFG_OLED_ADDRESS    0x3C

// GPIO for display I2C SDA
#define CFG_OLED_SDA        5

// GPIO for display I2C SCL
#define CFG_OLED_SCL        4

// Pixel width of display
#define CFG_OLED_WIDTH      128

// Pixel height of display
#define CFG_OLED_HEIGHT     64

// GPIO for ADC input
#define CFG_ADC_GPIO        36

// GPIO for PWM LED
#define CFG_LED_GPIO        13
```



Figuur 1. De Lolin-ESP32 met OLED-display.



Figuur 2. De Lolin ESP32-module met potentiometer en LED.

Listing 2. De setup()-functie.

```
void setup() {

#if CFG_OLED
  display.init();
  display.clear();
  display.setColor(WHITE);
  display.display();
#endif

  analogReadResolution(12);
  analogSetAttenuation(ADC_11db);

  pinMode(gpio_led,OUTPUT);
  ledcAttachPin(gpio_led,0);
  ledcSetup(0,5000,8);
}
```

Listing 3. Het staafdiagram.

```
#include "SSD1306.h"
...
void barGraph(unsigned v) {
  char buf[20];
  unsigned width, w;

  snprintf(buf,sizeof buf,"ADC %u",v);
  width = disp_width-2;
  w = v * width / 4095;
  display.fillRect(1,38,w,disp_height-2);
  display.setColor(BLACK);
  display.
fillRect(w,38,disp_width-2,disp_height-2);
  display.fillRect(1,1,disp_width-2,37);
  display.setColor(WHITE);
  display.drawLine(0,38,disp_width-2,38);
  display.
drawRect(0,0,disp_width-1,disp_height-1);
  display.setTextAlignment(TEXT_ALIGN_CENTER);
  display.setFont(ArialMT_Plain_24);
  display.drawString(64,5,buf);
  display.display();
}
```

Daarna wordt de GPIO voor de LED (`gpio_led`) geconfigureerd als uitgang, en geconfigureerd voor PWM door aanroepen van `ledcAttachPin()` en `ledcSetup()`.

Staafdiagram

Met behulp van de SSD1306 OLED tekent onze applicatie een staafdiagram om de ADC-waarde weergeven (zie **listing 3**). De functie `barGraph()` neemt een invoerwaarde `v` (de ADC-waarde) en formatteert die in een korte message met behulp van de array `buf` en met `snprintf()`. Rechthoeken worden in zwart en in wit getekend om het staafdiagram vorm te geven. De methode `display.drawString()` zet de geformatteerde tekst ook in het display. Tenslotte wordt de methode `display.display()` aangeroepen om de display-afbeelding in het geheugen in de OLED-controller van het display te laden.

De functie loop()

Als we een normaal Arduino programma zouden schrijven, dan zouden we een lus gebruiken zoals in **listing 4**.

Listing 4. Typische Arduino loop-code.

```
void loop() {
  uint adc = analogRead(ADC_CH);

#if CFG_OLED
  barGraph(adc);
#endif

  printf("ADC %u\n",adc);
  ledcWrite(0,adc*255/4095);
  delay(50);
}
```

Tabel 1. Taken uitgevoerd bij het opstarten van Arduino.

Taaknaam	Taak #	Prioriteit	Stack	CPU
loopTask	12	1	5188	1
Tmr Svc	8	1	1468	0
IDLE1	7	0	592	1
IDLE0	6	0	396	0
ipc1	3	24	480	1
ipc0	2	24	604	0
esp_timer	1	22	4180	0

De diverse stappen in deze code zijn eenvoudig:

- Lees de 12-bit ADC-waarde in (0...4095).
- Geef die weer als staafdiagram (indien het display is geconfigureerd).
- Print “ADC” en de waarde naar de seriële monitor.
- Stel de helderheid van de LED in door de PWM-parameter te wijzigen.
- En wacht 50 ticks.

We hebben dit programma bewust eenvoudig gehouden. U zou een gecompliceerde applicatie eigenlijk liever in kleinere ‘taken’ wille opsplitsen.

Wat zijn taken?

Wanneer u een dual-core ESP32-CPU hebt, is het mogelijk om twee programma’s *tegelijk* te laten draaien. Dit is spannend bij een microcontroller! Elke CPU gebruikt een eigen programmateller en andere registers om instructies uit te voeren. Dit staat voor twee besturings-*threads*. Een single-core CPU kan daarentegen op elk moment slechts één enkele thread uitvoeren. Om meer dan twee programma’s *tegelijk* uit te voeren, wordt een *scheduling*-truc gebruikt om bepaalde programma’s op te schorten terwijl andere hervat worden. Dit is de belangrijkste taak van FreeRTOS. Deze scheduling (‘planning’) gaat zo snel dat het lijkt alsof er meerdere programma’s tegelijk draaien. Maar op elk moment voert de dual-core CPU niet meer dan twee programma’s tegelijk uit. U kunt de ‘losjes’ gebruikte term *programma* in dit verband als een *taak* beschouwen.
Om het uitvoeren van meerdere gelijktijdige taken te managen, slaat de FreeRTOS-scheduler de registers van de huidige taak op en herstelt hij de registers van de volgende uit te voeren taak. Op deze manier kunnen meerdere taken onafhankelijk van elkaar worden uitgevoerd. Naast het programmatellerregister voor elke taak moet ook de stackpointer worden opgeslagen en teruggezet. Dit is nodig omdat C/C++ programma’s variabelen en retouradressen op de stack opslaan. Elke taak moet zijn eigen privé-stack hebben.
Bij de ESP32 draaien alle taken in dezelfde geheugenruimte (in tegenstelling tot bijvoorbeeld de Raspberry Pi). Daarom moeten taken zorgvuldig geprogrammeerd worden, zodat ze het geheugen dat door andere taken gebruikt wordt niet corrumperen. Daar komen bij gebruik van een multi-core CPU nog andere overwegingen bij, maar laten we dat voor een andere keer bewaren.

Arduino-taken

Wanneer uw ESP32 Arduino-programma begint, is het dan een taak? Jazeker! Naast uw eigen taak worden er bij het opstarten nog andere FreeRTOS-taken uitgevoerd. Sommige daarvan zijn verantwoordelijk voor services zoals timers, TCP/IP,

Bluetooth, enzovoort. En er kunnen nog meer taken worden gestart wanneer u services van de ESP32 aanroept.
Tabel 1 toont bij wijze van voorbeeld de FreeRTOS-taken die worden uitgevoerd wanneer de Arduino-functies `setup()` en `loop()` worden aangeroepen. Uw Arduino-hoofdtaak is die met de naam loopTask.
De kolom *Stack* toont de door FreeRTOS niet gebruikte stackbytes. Hoe u de stackgrootte kunt plannen, komt later aan de orde. Het overzicht staat in omgekeerde chronologische volgorde, gesorteerd op taaknummer. Uw hoofdtaak (loopTask) is de laatste taak die is aangemaakt. Ontbrekende taaknummers suggereren dat sommige taken werden aangemaakt en na voltooiing zijn beëindigd. De prioriteitskolom verduidelijkt de prioriteit die aan elke taak is toegekend, waarbij nul de laagste prioriteit is. Voor dit moment is het voldoende als u beseft dat prioriteiten bij FreeRTOS iets anders werken in FreeRTOS dan bijvoorbeeld bij Linux.
Tot slot worden de taken verdeeld tussen CPU 0 en CPU 1. Espressif dirigeert de ondersteunende taken naar CPU 0, terwijl de applicatietaken CPU 1 gebruiken. Hierdoor blijven de services voor TCP/IP en Bluetooth enzovoort soepel functioneren zonder dat uw programma daar aandacht aan hoeft te besteden. Maar ondanks deze conventie kunt u altijd extra taken in een van beide CPU's aanmaken.

Opstarten van de Arduino

Het is goed om te weten hoe ESP32 Arduino-programma's worden geïnitialiseerd voordat `setup()` wordt aangeroepen. Bekijk als voorbeeld het vereenvoudigde programmafragment in **listing 5** (de elementen van de watchdog-timer zijn weggelaten). We zien hier diverse interessante zaken:

- U ziet dat dit een C++ startup is (vandaar de externe 'C'-declaratie van `app_main()`).
- Een deel van de Arduino-initialisatie wordt uitgevoerd door `initArduino()`.
- De loopTask wordt gemaakt en uitgevoerd door de aanroep van `xTaskCreatePinnedToCore()`.

De loopTask wordt gecreëerd door `xTaskCreatePinnedToCore()` met een reeks argumenten aan te roepen. Het eerste argument is het adres van de functie die voor de taak (*loopTask*) moet worden uitgevoerd. Wanneer de taak is aangemaakt, voert de functie `loopTask()` eerst `setup()` uit en vervolgens `loop()` vanuit een eeuwige for-lus.
Het tweede argument is een string met de naam van de taak in de vorm van een C-string ("`loopTask`"). Argument drie specificeert een stackgroottevan 8192 *bytes*. Merk op dat dit afwijkt van standaard FreeRTOS op andere platforms, waar hiervoor 4-byte-woorden worden gebruikt. Wanneer de taak wordt aangemaakt, worden deze bytes toegewezen aan de taak voordat de functie wordt aangeroepen. Dit is een aanzienlijke portie SRAM die verspild zou zijn als de hoofdtaak niet werd gebruikt. Alle FreeRTOS-taken accepteren één lege pointer als argument. In dit geval wordt de waarde niet gebruikt en heeft deze de waarde NULL. Het vijfde argument is de prioriteit die FreeRTOS aan de taak moet toewijzen; in dit voorbeeld is dat 1. Hier moeten we het nog over hebben; gebruik de waarde 1 tot we de FreeRTOS-prioriteiten hebben besproken.
Na het prioriteitsargument komt een pointer naar een variabele die de *handle* van de taak ontvangt. Hier wordt die niet gebruikt; we hadden NULL kunnen schrijven. Het laatste argument specificeert op welke CPU-core de taak moet worden uitgevoerd. Dit kan 0 of 1 zijn bij een dual-core ESP32. CPU 1 wordt gebruikt om de hoofdtaak te starten. Voor single-core apparaten kan dit alleen maar nul zijn.

Listing 5. Vereenvoudigde ESP32 Arduino-startup.

```
void loopTask(void *pvParameters) {
  setup();
  for (;;) {
    loop();
  }
}

extern "C" void app_main() {
  initArduino();
  xTaskCreatePinnedToCore(
    loopTask,       // function to run
    "loopTask",     // Name of the task
    8192,           // Stack size (bytes!)
    NULL,           // No parameters
    1,              // Priority
    &loopTaskHandle, // Task Handle
    1);             // ARDUINO_RUNNING_CORE
}
```

Een taak creëren

Laten we nu eens aannemen dat onze applicatie bijzonder ingewikkeld is en dat we de code in twee taken moeten opsplitsen. We hebben al de `loopTask()`, die `loop()` steeds opnieuw aanroept. Om de stackruimte die er aan is toegewezen optimaal te benutten, zouden we normaal gesproken ons meest 'stack-intensieve' deel van het programma daar coderen.
Om het gebruik van taken te demonstreren, laten we de `loop()`-functie het volgende doen:

- Lees de 12-bit waarde van de ADC.
- Print de waarde naar de seriële monitor.
- Stuur de ADC-waarde naar de tweede taak.
- Wacht 50 ticks en keer terug (uit `loop()`).

Dan blijft voor de tweede taak over:

- Haal de ADC waarde op uit de `loop()`-functie (in de *loopTask*).
- Toon de ADC-waarde als staafdiagram (indien geconfigureerd).
- Pas de PWM-regeling voor de LED-helderheid aan.

De enige slimme plaats om die tweede taak te creëren is in de `setup()`-taak omdat dit maar één keer hoeft te worden gedaan. Dus laten we dit toevoegen aan onze `setup()`-taak:

```
void setup() {

#if CFG_OLED
  display.init();
  display.clear();
  display.setColor(WHITE);
  display.display();
#endif
```

```
  analogReadResolution(12);
  analogSetAttenuation(ADC_11db);

  pinMode (gpio_led,OUTPUT);
  ledcAttachPin(gpio_led,0);
  ledcSetup(0,5000,8);

...

  xTaskCreatePinnedToCore(
    dispTask,   // Display task
    "dispTask", // Task name
    2048,       // Stack size (bytes)
    NULL,       // No parameters
    1,          // Priority
    NULL,       // No handle returned
    1);         // CPU 1
}
```

Deze aanroep start een nieuwe taak met de naam *dispTask*, met een stack van 2048 bytes. We wijzen de taak toe aan CPU 1, met een prioriteit 1. Het zal de functie `dispTask()` uitvoeren (die nog gedefinieerd moet worden). Aangezien de taak op dezelfde CPU en met dezelfde prioriteit als onze `loopTask()` draait, zal de CPU gedeeld worden met de `loopTask()` en alle andere taken die met prioriteit 1 op die CPU worden uitgevoerd.

Wachtrijen

Er ontbreekt nog één ingrediënt – hoe sturen we de ADC-waarde van de ene taak naar de andere? U zou kunnen proberen om dat op de een of andere manier via het geheugen te te doen – dat lukt mogelijk bij taken die op dezelfde CPU draaien. Maar het wordt ingewikkelder bij het doorgeven van informatie van de ene CPU naar de andere. De FreeRTOS-wachtrij (*queue*) biedt de oplossing voor dit probleem.
De wachtrij geeft één taak toestemming om er een item op 'atomaire' manier op te zetten (*push*). Evenzo mag de ontvanger dat item op dzelfde wijze ophalen (*fetch*). Met *atomair* wordt bedoeld dat het betreffende item ondeelbaar is (zoals men vroeger van atomen dacht) en dus in zijn geheel wordt verwerkt. Bij een dual-core CPU met twee instructies die gelijktijdig worden uitgevoerd, let de timing erg nauw en bestaat het gevaar van *race conditions*. Het wachtrij-mechanisme neemt speciale maatregelen om dit atomair te laten gebeuren.

Een FreeRTOS-wachtrij wordt als volgt aangemaakt:

```
static QueueHandle_t qh = 0;
...
  qh = xQueueCreate(8,sizeof(uint));
```

Het eerste argument is de diepte van de wachtrij (het maximum aantal items in de wachtrij dat mogelijk is). Het tweede argument specificeert de grootte van elk item in de wachtrij. In dit geval is het de grootte van een `uint`-waarde in bytes. De retourwaarde is de *handle* van de wachtrij. Deze stap moet direct vóór het aanmaken van de dispTask in `setup()` worden uitgevoerd, zodat die meteen beschikbaar is.
Nu moet de wachtrij nog 'gevoerd' worden:

```
void loop() {
  uint adc = analogRead(ADC_CH);
  printf("ADC %u\n",adc);
  xQueueSendToBack(qh,&adc,portMAX_DELAY);
  delay(50);
}
```

Het actualiseren van het staafdiagram en de LED doen we in onze nieuwe functie `dispTask()`, die nog moet worden behandeld. De `loopTask()` leest echter de ADC-waarde in adc en drukt deze vervolgens af naar de seriële monitor. Daarna wordt hij aan de achterkant van de wachtrij geschoven met de functie `xQueSendToBack()`. We specificeren de *queue handle* in het eerste argument. De waarde die in de wachtrij moet worden geplaatst, wordt geleverd met behulp van een pointer in het tweede argument. Het laatste argument geeft aan hoe lang moet worden gewacht als de wachtrij vol is. Met de specificatie van de macro `portMAX_DELAY` geven we aan dat we willen dat de uitvoering geblokkeerd wordt tot er ruimte is in de wachtrij.

De weergavetaak

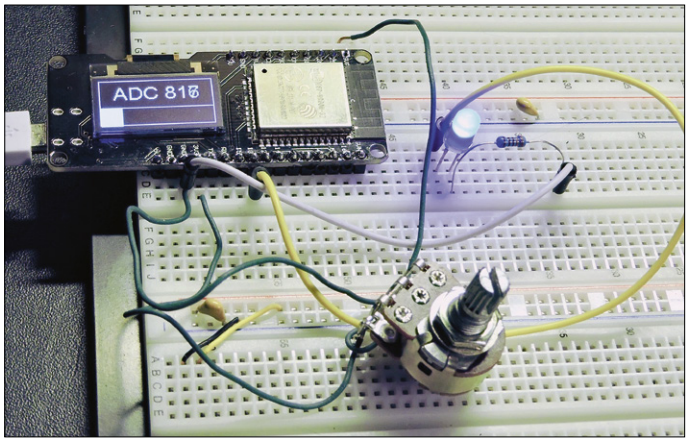
Laten we nu de weergavetaak in de onderstaande code nader bekijken:

```
void dispTask(void *arg) {
  uint adc;

  for (;;) {
    xQueueReceive(qh,&adc,portMAX_DELAY);
    barGraph(adc);
    ledcWrite(0,adc*255/4095);
  }
}
```

Deze functie krijgt, zoals alle taakfuncties, een pointer-argument wanneer hij begint. Hier wordt die niet gebruikt en is NULL vanwege de manier waarop de taak is gemaakt.
De hart van de taak wordt gevormd door een for-lus die waarden uit de wachtrij ontvangt. Hier retourneert `xQueueReceive()` de waarde in de wachtrij, maar wacht anders voortdurend als

Weblinks

[1] De ESP32 API: https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/
[2] FreeRTOS homepage: www.freertos.org
[3] Broncode van het project: https://github.com/ve3wwg/esp32_freertos/blob/master/freertos-tasks1/freertos-tasks1.ino
[4] ESP32 Lolin Board met OLED: www.elektor.com/lolin-esp32-oled-module-with-wifi



Figuur 3. Uitvoeren van de demo.

de wachtrij leeg is (het derde argument is `portMAX_DELAY`). Zodra een waarde uit de wachtrij is ontvangen, wordt het staafdiagram geactualiseerd en de LED PWM-waarde aangepast. Merk op dat het niet nodig is is om `delay()` aan te roepen in deze weergavetaak. De reden is dat de taak automatisch zal blijven wachten in de functie `xQueueReceive()` wanneer de wachtrij leeg is. Het uitvoeringstempo wordt bepaald door de verzendtaak.

Uitvoeren van de demo

Nadat alles is aangesloten en het programma is geladen, moet u het staafdiagram zien terwijl de LED oplicht (zie **figuur 3**). Als de LED geen licht geeft is, moet u de potmeter rechtsom draaien om meer licht te krijgen. Als u het programma zonder display hebt gecompileerd, start dan de seriële monitor op en let op uitvoer in de vorm van “ADC 3420” enzovoort. Door de potmeter linksom te draaien wordt de PWD-LED gedimd; rechtsom raaien resulteert in een helderder LED. De op de monitor getoonde ADC-waarden moeten zich op dezelfde manier gedragen (hogere waarde: meer licht; lagere waarde: minder licht).

Samenvatting

We hebben in dit artikel een heleboel zaken behandeld. De ESP32 opstartprocedure van `app_main()` tot `setup()` en `loop()` is aan de orde gekomen. Ook het aanmaken van de hoofdtaak met de naam `loopTask` is behandeld. In de demo hebben we onze eigen extra taak gecreëerd om OLED en PWM-LED aan te sturen, en hebben we de gegevens overgedragen via een wachtrij. Deze code draait aangenaam onafhankelijk van de hoofdtaak.

Daarnaast hebben we gebruik gemaakt van de hoofdtaak in de wetenschap dat daar een behoorlijke hoeveelheid stackruimte aan is toegewezen (vanuit de *heap*). In een volgende aflevering zullen we bespreken hoe we het gebruik van de stack kunnen bepalen voor nieuwe taken die u wilt aanmaken.

Dit voorbeeld was triviaal in termen van complexiteit. Maar het zal duidelijk zijn dat taken ingewikkelde toepassingen eenvoudiger maken door een probleem op te splitsen. Zo zou een MIDI-controller kunnen worden onderverdeeld in zend-, ontvangst- en besturingstaken. De ontvangsttaak kan dan ongestoord de inkomende seriële data ontvangen en decoderen naar *events* die de controletaak kan uitvoeren. Sommige van die events kunnen op hun beurt seriële data via de verzendtaak doen versturen. Zo wordt het werk op een keurige manier verdeeld. Dit maakt het onderhoud van de code niet alleen aangenamer, maar maakt het ook waarschijnlijker dat het programma correct zal werken. ⏮

(190182-04)