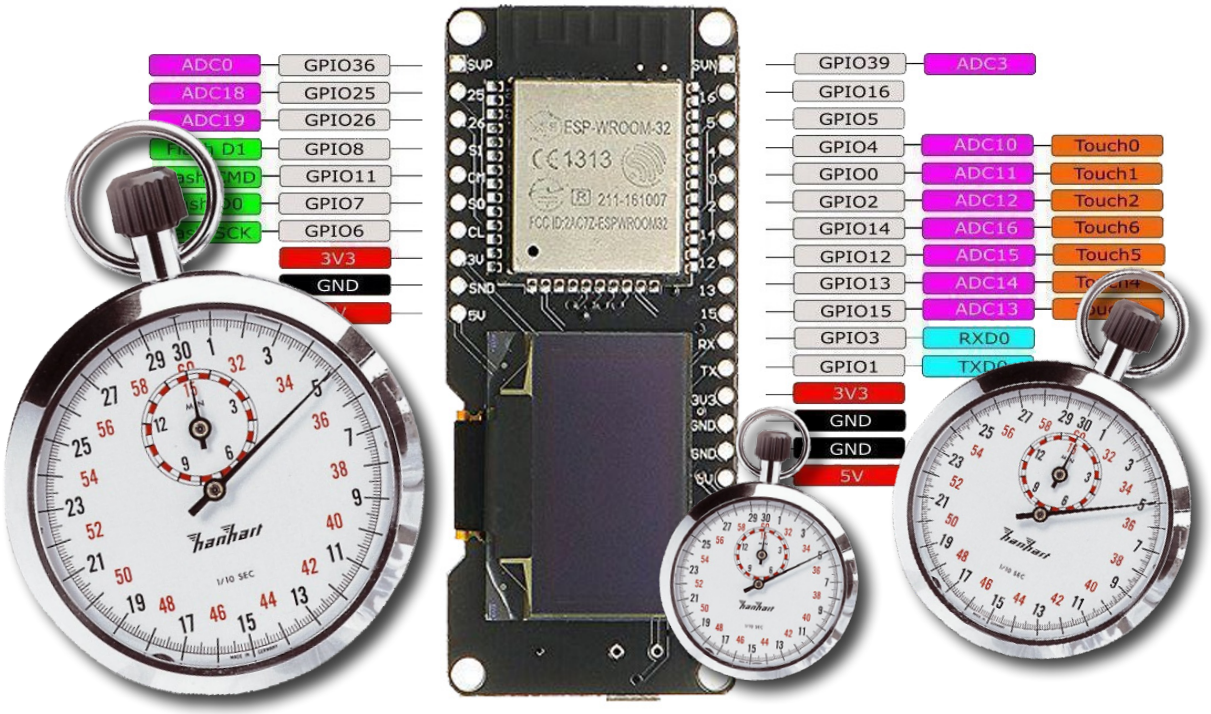
Multitasking met de ESP32 (3)

software-timers

Warren Gay (Canada)

Microcontroller-ontwikkelaars hebben vaak timers nodig, bijvoorbeeld om data opnieuw te verzenden na een time-out, voor het ontdenderen van toetsaanslagen of gewoon voor een knipperende LED. De ESP32 en de Arduino IDE bieden software-timers die gebruik maken van de FreeRTOS-bibliotheek. In dit derde deel van de serie richten we ons op het gebruik van die timers.



Om te laten zien hoe FreeRTOS-timers werken, bouwen we een demoprogramma met een klasse voor waarschuwings-LED's. Dat is niet zomaar een knipperende LED: als hij wordt geactiveerd, knippert hij snel vijf keer achter elkaar. Daarna blijft hij een tijdje uit, en dan gaat hij opnieuw knipperen. Het is de bedoeling om op die manier de aandacht te trekken voor een alarm, bijvoorbeeld als visuele indicatie voor het rinkelen van de telefoon.

De klasse `AlertLED` gebruikt intern één FreeRTOS-softwaretimer voor elke LED-driver. Het is een klasse, dus u kunt zoveel objecten instantiëren als u wilt zonder het geheugen van de ESP32 te belasten. In **listing 1** ziet u het volledige programma [2]. Op regel 100 is te zien hoe een AlertLED wordt geïnstantieerd met een simpele C++ declaratie:

```
static AlertLED alert1(GPIO_LED,1000);
```

De naam van de instantie is hier `alert1`. Hij maakt gebruik van de gespecificeerde LED (GPIO 12 zoals is aangegeven in regel 5) en de flitsperiode is 1000 milliseconden. We kunnen gemakkelijk een tweede instantiëren voor GPIO 13 door simpelweg de volgende regel toe te voegen:

```
static AlertLED alert2(GPIO_LED,1000);
```

Voor onze demonstratie beperken we ons tot één indicatie-LED, maar doordat we een C++-klasse hebben gemaakt, is het gemakkelijk om die meerdere keren te gebruiken.

De klasse AlertLED

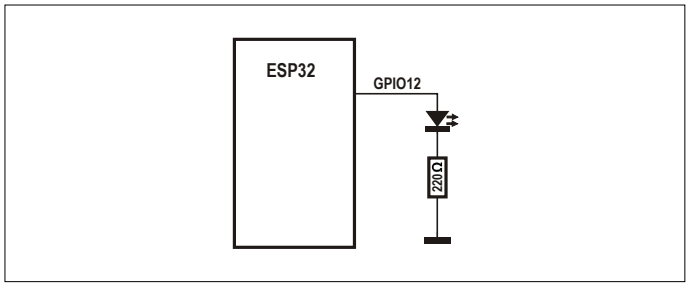
We hebben de klassedefinitie hieronder voor het gemak gekopieerd. De eerste klassevariabele is `thandle` (regel 11), die dient als handle voor de FreeRTOS timer (type `TimerHandle_t`). In eerste instantie krijgt die de waarde `nullptr` (hetzelfde als NULL in de taal C). De klassevariabele `state` (regel 12) gaat parallel lopen met de toestand van de LED. Hij is actief hoog (`true` = aan). De klassevariabele `count` (regel 13) gaan we in de callback gebruiken als een subtoestand, we komen daar later nog op terug. De variabelen `period_ms` en `gpio` bepalen respectievelijk de knipperperiode in milliseconden en de GPIO waarmee de LED wordt aangestuurd (regel 14 en 15).

```
0010: class AlertLED {
0011:    TimerHandle_t      thandle = nullptr;
```

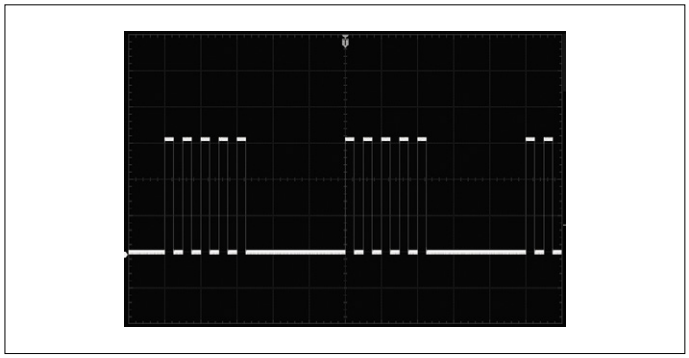
Listing 1. Het programma alertled.ino [2]

```
0001: // alertled.ino
0002: // MIT License (see file LICENSE)
0003:
0004: // LED is active high
0005: #define GPIO_LED      12
0006:
0007: //
0008: // AlertLED class to drive LED
0009: //
0010: class AlertLED {
0011:   TimerHandle_t     thandle = nullptr;
0012:   volatile bool     state;
0013:   volatile unsigned count;
0014:   unsigned          period_ms;
0015:   int               gpio;
0016:
0017:   void reset(bool s);
0018:
0019: public:
0020:   AlertLED(int gpio,unsigned period_ms=1000);
0021:   void alert();
0022:   void cancel();
0023:
0024:   static void callback(TimerHandle_t th);
0025: };
0026:
0027: //
0028: // Constructor:
0029: //  gpio        GPIO pin to drive LED on
0030: //  period_ms   Overall period in ms
0031: //
0032: AlertLED::AlertLED(int gpio,unsigned period_
      ms) {
0033:   this->gpio = gpio;
0034:   this->period_ms = period_ms;
0035:   pinMode(this->gpio,OUTPUT);
0036:   digitalWrite(this->gpio,LOW);
0037: }
0038:
0039: //
0040: // Internal method to reset values
0041: //
0042: void AlertLED::reset(bool s) {
0043:   state = s;
0044:   count = 0;
0045:   digitalWrite(this->gpio,s?HIGH:LOW);
0046: }
0047:
0048: //
0049: // Method to start the alert:
0050: //
0051: void AlertLED::alert() {
0052:
0053:   if ( !thandle ) {
0054:     thandle = xTimerCreate(
0055:       "alert_tmr",
0056:       pdMS_TO_TICKS(period_ms/20),
0057:       pdTRUE,
0058:       this,
0059:       AlertLED::callback);
0060:     assert(thandle);
0061:   }
0062:   reset(true);
0063:   xTimerStart(thandle,portMAX_DELAY);
0064: }
0065:
0066: //
0067: // Method to stop an alert:
0068: //
0069: void AlertLED::cancel() {
0070:   if ( thandle ) {
0071:     xTimerStop(thandle,portMAX_DELAY);
0072:     digitalWrite(gpio,LOW);
0073:   }
0074: }
0075:
0076: // static method, acting as the
0077: // timer callback:
0078: //
0079: void AlertLED::callback(TimerHandle_t th) {
0080:   AlertLED *obj = (AlertLED*)
       pvTimerGetTimerID(th);
0081:
0082:   assert(obj->thandle == th);
0083:   obj->state ^= true;
0084:
      digitalWrite(obj->gpio,obj->state?HIGH:LOW);
0085:
0086:   if ( ++obj->count >= 5 * 2 ) {
0087:     obj->reset(true);
0088:     xTimerChangePeriod(th,pdMS_TO_TICKS
       (obj->period_
         ms/20),portMAX_DELAY);
0089:   } else if ( obj->count == 5 * 2 - 1 ) {
0090:     xTimerChangePeriod(th,
0091:       pdMS_TO_TICKS(obj->period_ms/20+
          obj->period_
           ms/2),
0092:       portMAX_DELAY);
0093:     assert(!obj->state);
0094:   }
0095: }
0096:
0097: //
0098: // Global objects
0099: //
0100: static AlertLED alert1(GPIO_LED,1000);
0101: static unsigned loop_count = 0;
0102:
0103: //
0104: // Initialization:
0105: //
0106: void setup() {
0107:   // delay(2000); // Allow USB to connect
0108:   alert1.alert();
0109: }
0110:
0111: void loop() {
0112:   if ( loop_count >= 70 ) {
0113:     alert1.alert();
0114:     loop_count = 0;
0115:   }
0116:
0117:   delay(100);
0118:
0119:   if ( ++loop_count >= 50 )
0120:     alert1.cancel();
0121: }
```

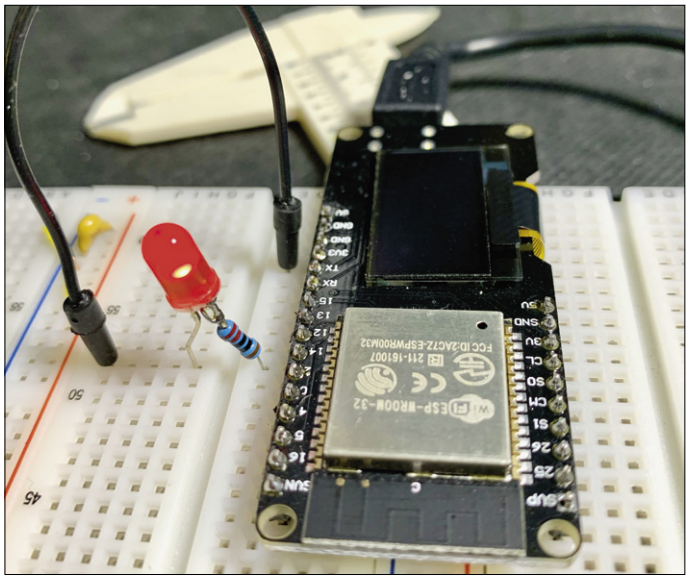
```
0012:   volatile bool     state;
0013:   volatile unsigned count;
0014:   unsigned          period_ms;
0015:   int               gpio;
0016:
0017:   void reset(bool s);
0018:
0019: public:
0020:   AlertLED(int gpio,unsigned period_ms=1000);
```



Figuur 1. De bedrading voor het demonstratieprogramma *alertled.ino*.



Figuur 2. Het LED-signaal voor alertled.ino, tijdbasis: 200 ms/div.



Figuur 3. ESP32, LED en 220Ω-weerstand aangesloten voor de demo *alertled.ino*.

```
0021:   void alert();
0022:   void cancel();
0023:
0024:   static void callback(TimerHandle_t th);
0025: };
```

De klasse heeft ook enkele publieke methodes, waaronder:
- `alert()` - activeer de LED-indicator (regel 21)
- `cancel()` - schakel de LED-indicator uit (regel 22)

Er is ook een C++ `static`-methode met de naam `callback()` op regel 24. Ook daar komen we nog op terug.

De methode AlertLED::alert()

De methoden `alert()` en `callback()` zijn de interessantste onderdelen van de klasse. We zagen al dat de handle werd geïnitialiseerd als `nullptr`. Als de methode `alert()` voor de eerste keer wordt aangeroepen, vindt hij dus op regel 53 de waarde nul. Zo weet hij dat er een nieuwe timer moet worden gecreëerd.
De software-timer wordt aangemaakt met behulp van `xTimerCreate()` in de regels 54 tot 59, waarbij de handle van de nieuw gemaakte timer wordt teruggegeven. De `assert()` macro op regel 60 test of het aanmaken is gelukt. De parameters voor `xTimerCreate()` zijn:

1. Een string als gebruikersvriendelijke naam voor onze timer (hier is het gewoon “`alert_tmr`”). Deze wordt niet gebruikt door FreeRTOS, behalve voor bepaalde functies voor het verzamelen van statistieken. Normaal gesproken zou de naam uniek moeten zijn, maar dat hoeft niet als u niet op naam gaat zoeken.
2. De periodetijd in *system ticks* (op regel 56 omgerekend van milliseconden met behulp van de macro `pdMS_TO_TICKS()`).
3. Dit geeft aan of het gaat om een “one shot”- (waarde `pdFALSE`) of een “auto-reload”-timer (waarde `pdTRUE`). In ons voorbeeld maken we een auto-reload timer (regel 57).
4. Dit is een “Timer ID” in FreeRTOS-terminologie. Dit moet worden gezien als een “gebruikersdata-parameter”. Het is een void-pointer, hier het adres van het klasse-object met behulp van het C++-sleutelwoord `this` (regel 58).
5. Het adres van de callback-functie. In regel 59 geven we de naam van de statische methode `AlertLED::callback()`. Dit is de functie die wordt aangeroepen als de timer afloopt.

Sommige dataleden worden gereset door het aanroepen van interne methode `reset()` op regel 62. Daarna wordt de timer met behulp van `xTimerStart()` gestart op regel 63. Alle timers worden ‘slapend’ gemaakt totdat ze worden opgestart. Eenmaal gestart, komen ze in de toestand ‘running’.

De statische methode AlertLED::callback()

De klasse AlertLED definieert een *statische* methode met de naam `callback()`. Voor wie niet bekend is met C++: dit betekent dat de methode slechts een functie is. Hij heeft geen impliciete objectpointer (geen ‘this’-pointer). Het is hetzelfde als een C-functie met uitzondering van de rare C++-naam en het feit dat hij speciale toegang heeft tot de internals van het klasse-object.
De software-timer-callback heeft één argument van het type `TimerHandle_t` (regel 79). Dit geeft ons toegang tot de handle van de timer die op dat moment afloopt, maar we hebben meer informatie nodig: in dat geval het adres van de LED-instantie

die moet worden aangestuurd. In regel 80 wordt die informatie opgehaald met behulp van de 'Timer ID' waarde – het adres van de (LED) klasse-instantie (`alert1`) met de aanroep `pvTimerGetTimerID()`, die de timer-handle nodig heeft. Deze functie haalt de waarde op die we hebben gegeven in regel 58, toen de timer werd aangemaakt. We kunnen nu toegang krijgen tot het LED-object en zijn members via de variabele `obj`. Regels 83 en 84 schakelen de voor de LED geconfigureerde GPIO-lijn. Dan wordt de waarde van de teller verhoogd (regel 86), en totdat de teller de waarde 9 bereikt, keren we gewoon terug uit de callback. Zo gaat de LED snel knipperen tijdens de eerste helft van de alarmperiode.
Als de teller de waarde 9 bereikt, worden de regels 90 tot 93 uitgevoerd. De functie `xTimerChangePeriod()` wijzigt het interval dat onze timer gebruikt. Merk op dat period_ms/20 wordt opgeteld bij period_ms/2, ter compensatie van de laatste donkere helft van de snelle knippertijd.
Ten slotte worden bij de tiende aanroep van de callback de regels 87 tot en met 88 uitgevoerd. De objectvariabelen worden gereset door het oproepen van interne methode `reset()` (regel 87), en de timerperiode wordt opnieuw ingesteld voor snel knipperen op regel 88. Omdat de objectvariabele `count` op nul wordt gezet, wordt de hele cyclus herhaald. Omdat dit een auto-reload-timer is, zal het patroon zich blijven herhalen totdat `AlertLED::cancel()` wordt aangeroepen.

Methode AlertLED::cancel()

Om het alarm te stoppen gebruiken we de methode `AlertLED::cancel()`. Hiertoe wordt `xTimerStop()` aangeroepen (regel 71) en wordt de LED geforceerd uitgeschakeld door LOW naar de GPIO-lijn van de LED te schrijven in regel 72.

Waarom volatile?

De oplettende lezer zal gemerkt hebben dat we het C-sleutelwoord `volatile` hebben gebruikt in regel 12 en 13 bij de definitie van de klasse. Waarom is dat nodig? Het vertelt de compiler dat hij voor de waarde van de variabelen `state` of `count` altijd naar het geheugen van de klasse-instantie moet gaan (`alert1` in onze demo) in plaats van te vertrouwen op een waarde die misschien nog in een register staat. Er zijn twee taken die interactie hebben met onze klasse:

- De daemon-klasse van FreeRTOS (via de callback)
- Onze eigen taak die gebruik maakt van de methoden `alert()` of `cancel()` van de klasse.

Als deze waarden niet als volatile waren aangemerkt, zouden de beide taken waarden kunnen gebruiken die nog in een register staan, omdat dat normaal gesproken de snelste manier is om de waarde te vinden. Maar de waarden in het geheugen kunnen intussen door de *andere* taak zijn veranderd. Daarom zorgt het sleutelwoord volatile ervoor dat de compiler deze mogelijke optimalisatie buiten beschouwing laat en code genereert die de waarden altijd uit het geheugen haalt.

De demo

Voor een eenvoudige demo *alertled.ino* kunnen we weer gebruik maken van de Lolin ESP32 OLED Display Module. De bedrading is weergegeven in **figuur 1**. De LED is actief-hoog bedraad.
Om te bewijzen dat de `AlertLED`-klasse kan in- en uitschakelen, bevat de functie `loop()` een teller met de naam `loop_count` (regel 101). De functie `setup()` activeert in eerste instantie de LED, maar wanneer loop_count de waarde 50 bereikt, wordt de waarschuwing geannuleerd op regel 120. Wanneer de teller later de waarde 70 bereikt, wordt de waarschuwing opnieuw geactiveerd op regel 113 en wordt de teller gereset.
Als het programma in de ESP32 is geflasht en gestart, moet u de alarm-LED zien knipperen op een manier die de aandacht trekt. Even later gaat hij uit, om vervolgens weer te hervatten. In **figuur 2** ziet u het stuursignaal van de LED.
Figuur 3 toont de opbouw, die bestaat uit een weerstand, een LED en een jumperdraad naar GND.

Beperkingen van FreeRTOS

Tot nu toe hebben we de software-timer API toegepast zonder al te veel te zeggen over enkele belangrijke beperkingen van de callback. Deze beperkingen zijn onder andere:

- Voer nooit een blokkerende oproep uit binnen de callback (zoals `delay()` of een blokkerende push naar een wachtrij).
- Vermijd lange uitvoeringstijden (dat zal de API verstoren).
- Vermijd het gebruik van te veel stackruimte (ESP32 Arduino is waarschijnlijk beperkt tot ongeveer 1500 bytes). Functies zoals `printf()` en `snprintf()` hebben vaak veel stackruimte nodig en kunnen daarom beter niet worden gebruikt in de callback.
- Sommige van de FreeRTOS API-functies werken via een interne wachtrij. Daarom is in ons demoprogramma in de regels 88 en 92 gebruik gemaakt van `portMAX_DELAY`. De diepte van de timer-wachtrij voor de ESP32 Arduino is 10; die zal waarschijnlijk niet vol raken tenzij er veel timers tegelijk worden gebruikt.

Samenvattend

U vraagt zich misschien af waarom we niet gewoon een taak gebruiken voor elke waarschuwings-LED afzonderlijk? De belangrijkste reden hiervoor is dat voor elke taak een stack en een task control block (TCB) nodig zijn. Als u 32 LED-indicatoren zou willen gebruiken, zou dat 32 x (356 + 1500) bytes in beslag nemen, in totaal 59.392 bytes! Bij onze aanpak met FreeRTOS-timers heeft u voor elke timer 40 bytes nodig, zodat het totaal uitkomt op 1200 bytes (het objecttype `StaticTimer_t` is 40 bytes groot). De stack voor alle timers wordt gedeeld met de FreeRTOS daemon-taak (voorheen bekend als de *timer service*-taak).
Soms worden in een toepassing verschillende benaderingen gebruikt om te voorzien in speciale behoeften, maar vaak voldoet een FreeRTOS-timer aan de eisen. ⏮

200071-02

Weblinks

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

[2] Broncode van het project: https://github.com/ve3wwg/esp32_freertos/blob/master/alertled/alertled.ino