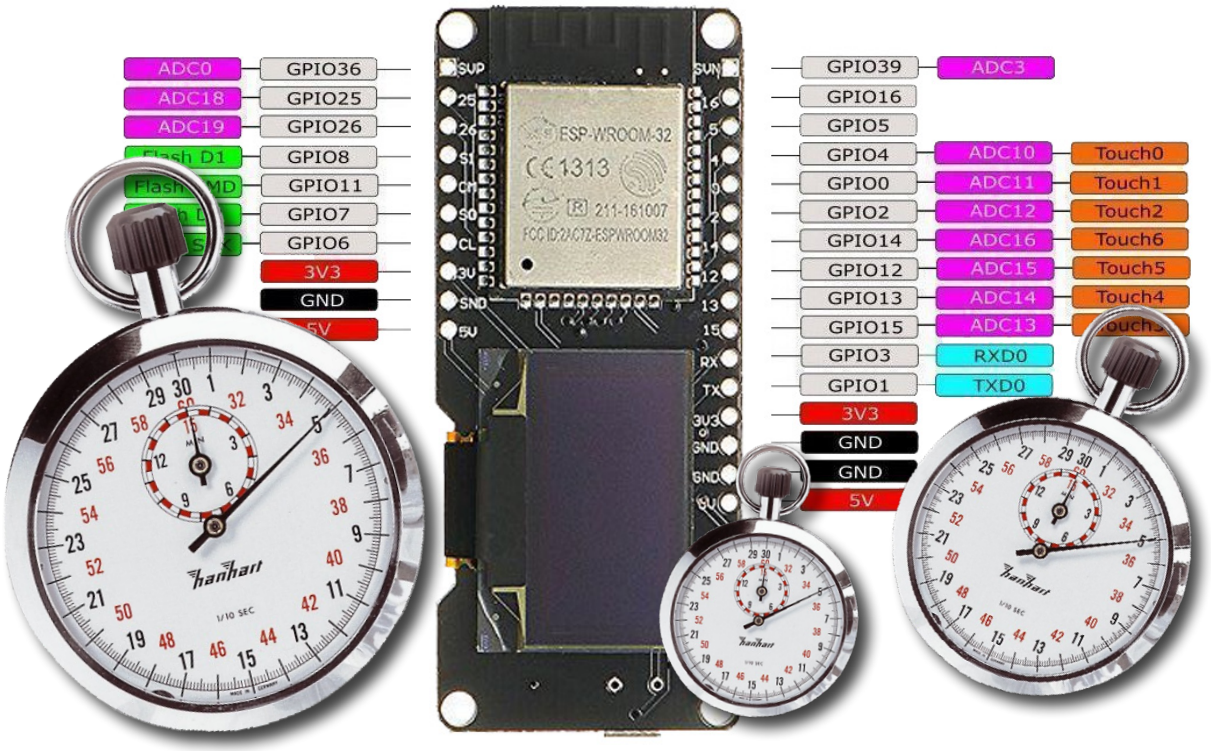
Practical ESP32 Multitasking (3)

Software timers

By **Warren Gay** (Canada)

Microcontroller developers often need timers, to perform retransmissions and other error recovery procedures, for debouncing of button presses, or just for a blinking LED indicator. The ESP32 device and the Arduino IDE provide software timers using the included FreeRTOS library. In this third part of the series we focus on the timer facility.



In order to illustrate how FreeRTOS timers work, we'll work with a demo assignment to construct an alert LED class. This is not your average blinking LED however. When activated, it quickly blinks five times and then is off for the rest of the period, and repeats. It will visually appear as an attention getting alert or as a visual phone ring alert.

The `AlertLED` class will use one FreeRTOS software timer internally for each LED driver. As a class, you will be able to add as many as you need, without burdening the memory utilization of the ESP32. **Listing 1** illustrates the full program [2]. Line 100 shows the creation of an alert LED instance with a simple C++ declaration statement:

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

The class instance name here is `alert1`, using the specified LED (GPIO 12 from line 5), and the flash period of 1000 milliseconds. We could easily add another one for GPIO 13, by simply adding the line:

```
static AlertLED alert2(13,1000);
```

Our demonstration will focus on one indicator but the C++ class packaging makes for convenient use.

The AlertLED class

The class definition is copied below for convenience. The class data members include the member `thandle` (line 11), which is a handle to the FreeRTOS timer (type `TimerHandle_t`). Initially it is assigned the value of `nullptr` (same as the C language NULL). The data member named `state` will track the LED state (line 12), which is active high (`true` = on). The data member `count` (line 13) will be used in the callback as a substate, which is explained later. Finally members `period_ms` and `gpio` configure the blink period in milliseconds and the GPIO to drive for the LED (lines 14 and 15).

Listing 1: The alertled.ino program [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: }
```

```
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:
```

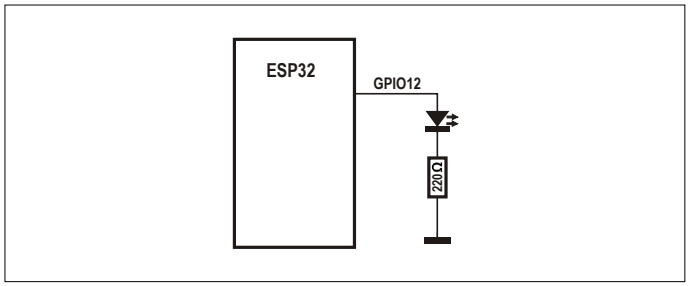


Figure 1. The wiring of the demonstration program *alertled.ino*.

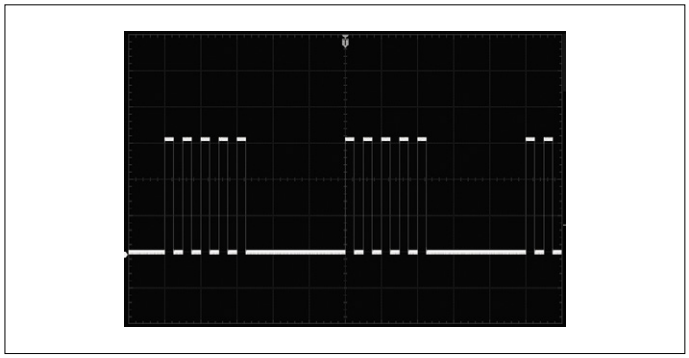


Figure 2. The LED drive signal for *alertled.ino*, horizontal is 200 ms / div.

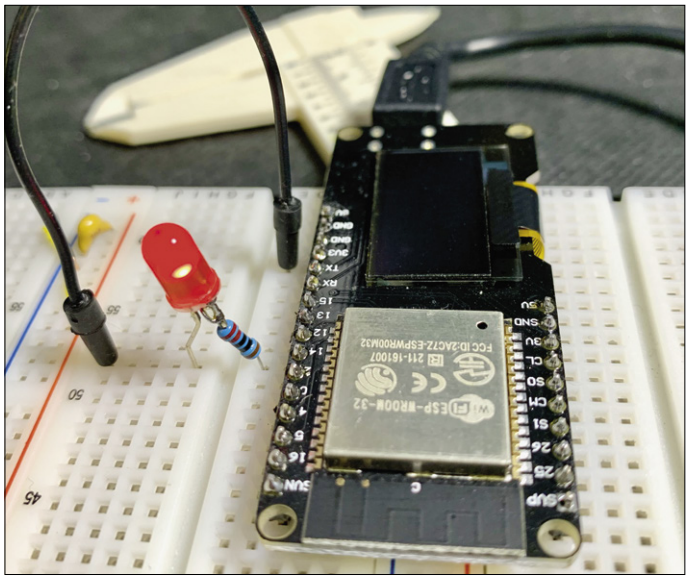


Figure 3. ESP32, LED and 220 ohm resistor wired for the *alertled.ino* demonstration.

```
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: };
```

The class also includes some public method calls including:

- `alert()` - activate the LED indicator (line 21)
- `cancel()` - disable the LED indicator (line 22)

There is also a C++ `static` method named `callback()` in line 24. More will be said about this later.

Method AlertLED::alert()

The method `alert()` and `callback()` are the most interesting parts of the class. We noted that the handle was initialized as `nullptr`. When the `alert()` method is called for the first time, it will find the value to be null in line 53, causing the timer to be created.
The software timer is created by the use of `xTimerCreate()` in lines 54 to 59, returning the handle of the newly created timer. The `assert()` macro in line 60 tests for successful creation. The arguments to `xTimerCreate()` are:

1. A user friendly string name of our timer (here it is simply "`alert_tmr`"). This is not used by FreeRTOS other than for some statistic gathering functions. It normally should be unique but does not have to be if you don't perform lookups by name.
2. A timer period in *system ticks* (converted from milliseconds in line 56 using the `pdMS_TO_TICKS()` macro).
3. This indicates whether the timer is "one shot" (value `pdFALSE`) or an "auto-reload" timer (value `pdTRUE`). Our example creates an auto-reload timer (line 57).
4. This is a "Timer ID" in FreeRTOS terminology. This should be thought of as a "user data parameter". It is a void pointer, which in this example, is the address of the class object using the C++ reserved keyword `this` (line 58).
5. The address of the callback function. In line 59, we supply the name of the static method `AlertLED::callback()`. This is the function that will be called when the timer is triggered.

Some data members are reset by calling internal method `reset()` in line 62, and followed by starting the timer in line 63 using `xTimerStart()`. All timers are created "dormant" until they are started/restarted. Once started, they enter the "running" state.

Static method AlertLED::callback()

The AlertLED class defines a *static* method call named `callback()`. For those unfamiliar with C++, this means that the method is just a function. As such, it has no implied object pointer (no "this" pointer). It is the same as a C function except for the funky C++ name and the fact that it enjoys special access to the internals of the class object.
The software timer callback requires one argument of type `TimerHandle_t` (line 79). This gives us access to the timer handle that is being triggered but we need more informa-

tion - in that case the LED instance which should be driven. Line 80 obtains this information using the "Timer ID" value - the address of the (LED) class instance (`alert1`) using the `pvTimerGetTimerID()` call, which needs the timer's handle. This retrieves the value we supplied in line 58, when the timer was created. We can now get access to the LED object and its members by the variable `obj`.
Lines 83 and 84 toggles the configured GPIO for the LED. Then the value of count is incremented (line 86), and until the count reaches 9, we simply return from the callback. This performs the simple on/off fast blinkling for the first half of the alert period. When the count reaches 9, lines 90 to 93 are executed. The function `xTimerChangePeriod()` alters the time period used by the timer we created. Notice that period_ms/20 is added to the period_ms/2, to compensate for the last dark half of the fast blink time.
Finally, on the 10th time into the callback, lines 87 to 88 are executed. The data members are reset by calling internal method `reset()` (line 87), and the timer period re-established for a fast blink in line 88. Because the data member `count` is cleared to zero, the whole cycle repeats. The auto-reload nature of the timer will keep the pattern repeating until `AlertLED::cancel()` is called.

Method AlertLED::cancel()

To stop the alert from blinking, `AlertLED::cancel()` method is used. To accomplish this, `xTimerStop()` is called (line 71), and the LED is forced off by writing LOW to the LED's GPIO in line 72.

The need for volatile

The alert reader will have noticed that we used the `volatile` C keyword in lines 12 and 13 of the class definition. Why is this necessary? This informs the compiler that whenever it reaches for the data member values `state` or `count`, that it should go directly to the memory of the class instance (`alert1` in our demo) rather than depend upon a value that might still exist in a register. There are two tasks interacting with your class:

- The FreeRTOS daemon class (via the callback)
- Your task that invokes the `alert()` or `cancel()` methods of the class.

If these values were not marked as volatile, either or both tasks might use values that are still in a register, since this is normally the optimal thing to do. However, the values in memory may have changed by the *other* task. For this reason the keyword volatile causes the compiler to disregard this possible optimization and generates code that fetches the values directly from memory instead.

The demonstration

For a simple demo *alertled.ino*, we can again use the Lolin ESP32 OLED Display Module. The wiring is shown in **Figure 1**. The LED is wired according to the active-high configuration.

To prove that the `AlertLED` class can activate and cancel, the `loop()` function maintains a count variable named `loop_count` (line 101). Function `setup()` initially activates the LED, but when the loop count reaches 50, the alert is canceled in line 120. When the loop count later reaches 70, the alert is reactivated in line 113, and the counter reset.
When the program is flashed to the ESP32 and running, you should see the alert LED flashing in an attention getting manner. A little while later it will turn off, only to resume again. **Figure 2** illustrates the LED drive signal.
Figure 3 shows the setup, using a resistor LED combination and a Dupont wire to GND.

FreeRTOS limitations

So far, we have applied the software timer API, without saying too much about some important limitations of the callback. These limitations include:

- Never perform a blocking call within the callback (like `delay()` or a blocking queue push for example).
- Avoid long execution times (this will disrupt the API).
- Avoid using too much stack space (ESP32 Arduino is probably limited to about 1500 bytes). Functions like `printf()` and `snprintf()` often require considerable stack space and thus should probably be avoided in the callback.
- Some of the FreeRTOS API functions operate through an internal queue. This is why our demo program used `portMAX_DELAY` in lines 88 and 92. The timer queue depth for the ESP32 Arduino is 10, which is unlikely to get full, unless many timers are involved at once.

Summary

You might wonder why not simply use a task to manage each alert LED? The main reason for this is that each task requires a stack and a task control block (TCB). If you needed to allocate for 32 LED indicators, this would require 32 x (356 + 1500) bytes, totalling 59,392 bytes! Contrast that to the FreeRTOS timer approach requiring 40 bytes for each timer, for a total of 1200 bytes (object type `StaticTimer_t` is 40 bytes). The stack is shared with the FreeRTOS daemon task for all timers (previously known as the *timer service* task).
Applications might mix and match approaches for special needs but many times a FreeRTOS timer satisfies the requirements. ⏮

200071-01

Web Links

[1] Practical ESP32 Multitasking, Elektor Magazine 1/2020: www.elektormagazine.com/190182-01

[2] Project source code: https://github.com/ve3wwg/esp32_freertos/blob/master/alertled/alertled.ino