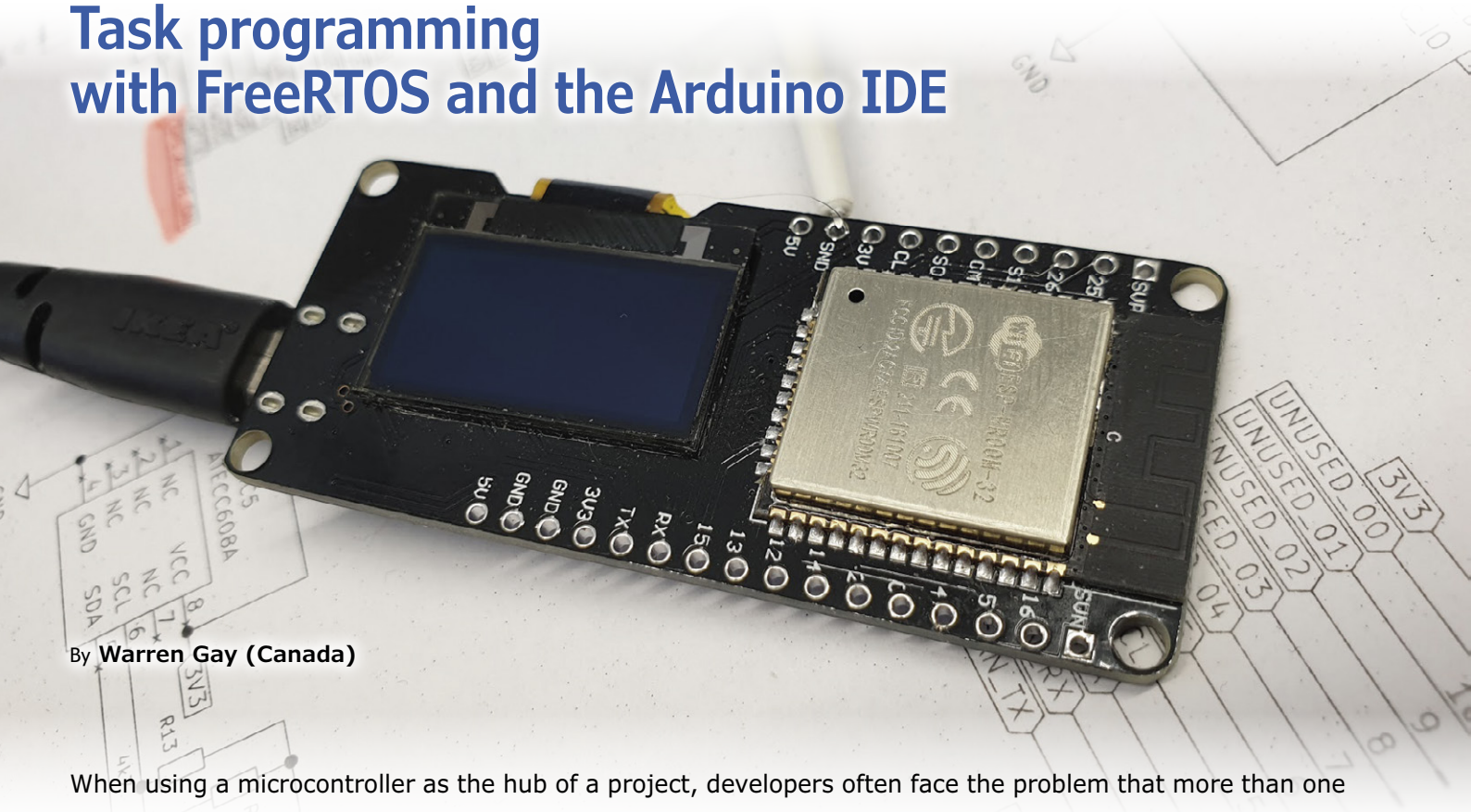
Practical ESP32 Multitasking

Task programming with FreeRTOS and the Arduino IDE



By **Warren Gay (Canada)**

When using a microcontroller as the hub of a project, developers often face the problem that more than one task needs to be performed at a time: Scanning sensor values, controlling actuators, visualizing device states and/or waiting for a human user to enter. Fortunately, this problem can be solved in a very elegant way with task programming based on lightweight embedded operating systems. FreeRTOS is an open source and widely used operating system, available for many microcontroller platforms. The very popular tandem ESP32 and Arduino IDE makes it especially easy to use FreeRTOS for task programming, as it is already integrated into the core libraries. In fact, you may have always used FreeRTOS without knowing it!

The Espressif ESP32 platform is an exciting microcontroller for maker projects. Its hardware capabilities, extensive software and affordability place it in the "must have" category for many. Espressif has provided both Arduino compatible and native development environments. Their native mode is officially known as ESP-IDF (Espressif IoT Development Framework). For this article, the familiar Arduino environment will be used. Most of the API (Application Programming Interface) is available to both [1]. Some of our readers may know that Espressif uses the popular open source embedded operation system FreeRTOS, to implement the ESP32 library functions for WiFi, Bluetooth and many more features of the microcontroller. In this article, we'll see that FreeRTOS [2] and task programming is also used for the simple Arduino `setup()` and `loop()` functions. First, we will describe a simple ESP32 demo project [3]. Then we will have a look under the hood; later we will establish our own tasks for the demo.

Application

For this installment in the series we'll develop an application that reads the analogue to digital converter (ADC) from a potentiometer and display it on an OLED in bar graph form. In addition we'll illuminate an LED using pulsewidth modulation (PWM), varying it from dim to bright.

This article will use the Lolin ESP32 board with the builtin OLED (**Figure 1**) [4]. If you have another ESP32 board but lack the SSD1306 compatible OLED, you can disable the OLED display in the program and rely on the brightness of the LED instead. The LED and the potentiometer will be wired up according to the schematic in **Figure 2**.

As the application runs, turning the potentiometer will vary the input voltage seen on GPIO36. This voltage will be measured as a 12-bit value, returning a value from 0 to 4095. Turning the potentiometer (or "pot") fully clockwise will return a value of 4095 which will cause full brightness of the LED. If the opposite

happens, then reverse the outer lead connections on the pot. The centre connection is the pot's wiper arm and will vary in voltage from ground potential up to the full +3.3 V. Be careful not to wire the pot to the +5 V supply, as that exceeds the GPIO input maximum voltage.

Program configuration

Let's examine the software now. The program has C macros defined to make it easy for you to reconfigure it. These are shown in **Listing 1**. Set macro `CFG_OLED` to zero, if you are not using a display (this causes the address and I²C macros to be ignored).
Macro `CFG_ADC_GPIO` value has been chosen to use ADC1 on channel 0 (GPIO36). Finally, the LED has been configured by `CFG_LED_GPIO` to use GPIO13.
The full source code is available from github in the *freertos-tasks1* subdirectory [3].

Initialization

Ignoring tasks for the moment, the device `setup()` for our application is as shown in **Listing 2**. This would be a typical initialization in the Arduino framework.
The display is only initialized when it is compiled in (configured). After that, using the Arduino routine `analogReadResolution()`, the ADC is configured to read 12-bit values. The function `analogSetAttenuation()` is called so that it will read a value from 0 to +3.3 V. The ADC includes an internal amplifier, so it is important to configure it to use the correct range.
After that, the GPIO for the LED (`gpio_led`) is configured as an output, and configured for PWM by calls to `ledcAttachPin()` and `ledcSetup()`.

Listing 1. Configuration Options.

```
// 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
```

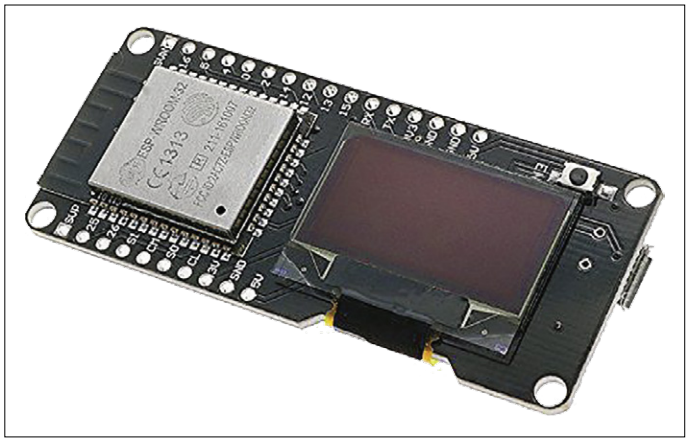


Figure 1: The Lolin ESP32 with OLED Display.

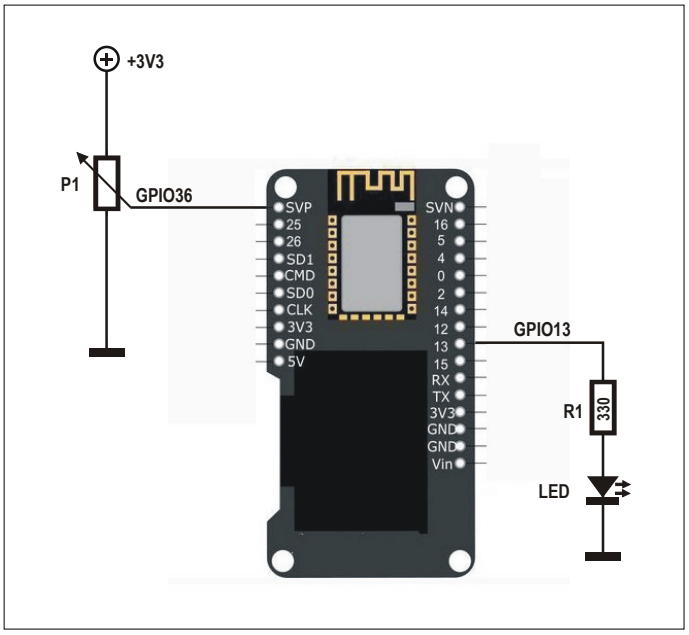


Figure 2: Lolin ESP32 module equipped with potentiometer and LED.

Listing 2. The setup() Function.

```
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. The Bar Graph Display Function.

```
#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();
}
```

Bar graph display

Using the SSD1306 OLED device, the application will draw a bar graph on it and display the ADC value (see **Listing 3**). The `barGraph()` function accepts an input value `v` which is the ADC value, and formats that into a short message using the array buf, using `snprintf()`. The some rectangles are drawn in black and in others in white to represent the bar graph. The `display.drawString()` method call places the formatted text in the display also. At the end, the `display.display()` method is called to put the in-memory image of the display into the OLED controller for the device.

The loop() function

If we were writing a normal Arduino program, we would code a loop like the one in **Listing 4**.

Listing 4. Typical 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);
}
```

Table 1. Tasks running at Arduino Startup.

Task Name	Task #	Priority	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

In this code, the steps are simple:

- Read the 12-bit ADC value (gets a value from 0 to 4095).
- Display it on the bar graph (when configured).
- Print “ADC” and the value to the serial monitor.
- Set the LED brightness by changing the PWM parameter.
- And delay for 50 ticks.

This is an intentionally simple program. But if this application was complicated, you would want to break it up into smaller “tasks”.

What are tasks?

When you have a dual core ESP32 CPU, it is possible to have two programs running **simultaneously**. This is exciting in a micro-controller! Each CPU operates with its own program counter and other registers to carry out instructions. This represents two **threads** of control. A single core CPU on the other hand, would only perform one thread of control at any given instant. To allow more than two programs to execute **concurrently**, a scheduling trick is used to suspend programs while resuming others. This is the main job of FreeRTOS. This scheduling is done so quickly that it has the **appearance** of running several programs at once. But at any given instant in time, the dual core CPU has no more than two programs executing simultaneously. Think of this loosely used term “program” as a task. To manage running several concurrent tasks, the FreeRTOS scheduler saves the registers of the current task and restores the registers of the next task to be run. In this way, multiple tasks can run independently. In addition to the program counter register for each task, its **stack pointer** must also be saved and restored. This is necessary because C/C++ programs have variables and return function addresses saved on the stack. Each task must have its own private stack.
On the ESP32, all tasks run within the same memory space (unlike the Raspberry Pi, for example). For this reason, tasks must be programmed carefully so that they do not corrupt the memory used by other tasks. Additional considerations apply when you have a multi-CPU core operating, but let’s save that discussion for another time.

Arduino tasks

When your ESP32 Arduino program begins, is it in a task? Absolutely! In addition to your own task at startup, there are other FreeRTOS tasks executing. Some of these tasks provide services such as timers, TCP/IP, Bluetooth etc. Additional tasks may be started as you request services from from the ESP32. **Table 1** shows an example of FreeRTOS tasks running when

Arduino functions `setup()` and `loop()` are invoked. The task named *loopTask* is your main Arduino task.
The column labeled *Stack* represents FreeRTOS unused stack bytes. How to plan for stack sizes will be covered at a later time. The chart is sorted in reverse chronological order, by task number. Your main task (*loopTask*) is the last task created. Missing task numbers suggest that some tasks were created and ended when their job was completed. The priority column illustrates the task priorities assigned, with zero representing lowest priority. For now, just know that priorities operate a little differently in FreeRTOS than they do in Linux for example. Finally, the tasks are divided between CPU 0 and CPU 1. Espressif places their support tasks in CPU 0, while application tasks use CPU 1. This keeps services for TCP/IP and Bluetooth etc. running smoothly without any special consideration from your application. Despite this convention, it is still possible for you to create additional tasks in either CPU.

Arduino startup

It's good to know how ESP32 Arduino programs initialize before `setup()` is called. Consider the simplified program snippet in **Listing 5** (the watchdog timer elements have been left out). From this example, we can note some interesting things:

- Note that this is a C++ startup (hence the extern "C" declaration of `app_main()`).
- Some Arduino initialization is performed by `initArduino()`.
- The *loopTask* is created and run by the call to `xTaskCreatePinnedToCore()`.

The loopTask is created by calling `xTaskCreatePinnedToCore()`, with a number of arguments. The first argument is the address of the function to be executed for the task (*loopTask*). When the task is created, the function `loopTask()` then runs invoking `setup()` first and then `loop()` from a forever "for" loop.
The second argument is a string describing the name of the task as a C string (`"loopTask"`). The stack size argument three is specified as 8192 *bytes*. Note that this differs from the stock FreeRTOS on other platforms, which use 4-byte words instead. When the task is created, this is allocated from the heap and assigned to the task before the function is called. This is a fair bit of SRAM that would be wasted if the main task was not used.
All FreeRTOS tasks accept one void pointer as an argument. In this case the value is not used and is supplied as NULL. The fifth argument is the FreeRTOS priority to assign to the task, which is 1 in this example. This requires more discussion, so use 1 until FreeRTOS priorities can be explained.
After the priority argument is a pointer to a variable that will receive the handle to the task. In this case, it is not used and could have been supplied as NULL. The last argument specifies which CPU core for it to run on. It can be 0 or 1, for the dual ESP32. CPU 1 is used to start the main task. For single core devices, this can only be zero.

Creating a task

Let's pretend our application is complicated and that we need to break the code into two tasks. We already have the `loopTask()`, which calls `loop()` over and over. To take advantage of the stack space allocated to it, we would normally code our most stack intensive part of the program there.

Listing 5. Simplified 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
}
```

For our task demonstration, let's perform the following in the `loop()` function:

- Read the 12-bit value from the ADC.
- Print the value to the serial monitor.
- Send the ADC value to the second task.
- Delay 50 ticks and return (from `loop()`).

This leaves the second task to do:

- Get the ADC value from the `loop()` function (in the *loopTask*).
- Display the ADC value on the bar graph (when configured).
- Modify the PWM control for LED brightness.

The only sensible place to create the second task is in the `setup()` task because this creation only needs to be performed once. So let's add that to our `setup()` task:

```
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
}
```

This call will start a new task named *dispTask*, with a stack of 2048 bytes. We have assigned it to CPU 1, with a priority of 1. It will be executing function `dispTask()` (yet to be defined). Since the task will run on the same CPU and at the same priority as our `loopTask()`, the CPU will be shared with the `loopTask()` and any other priority 1 tasks on that CPU.

Queues

There is still one missing ingredient — how do we send the ADC value from one task to another? You might try to design something in memory, which might work for tasks on the same CPU. But things get more complicated when passing information from one CPU to another. The FreeRTOS queue is the solution to our problem.

The queue allows one task to push an item into it in an "atomic" fashion. It likewise allows the receiver to fetch that item in an atomic way. By atomic, I mean that the item cannot be partly pushed or partly received. With a dual core CPU with two instructions executing simultaneously, there are timing issues and race conditions. The queue mechanism takes special measures to make this happen atomically.

To create a FreeRTOS queue, see the listing below:

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

The first argument is the depth of the queue (the maximum number of queued entries that it can hold). The second argument specifies the size of each queued item. In this case it is the size of a `uint` value in bytes. The return value is the queue's handle. This step should be performed immediately before the creation of the *dispTask* in `setup()`, so that it is immediately available for it.

Now let's turn our attention to feeding that queue:

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

The bar graph and LED updates will be done in our new function `dispTask()`, yet to be described. The `loopTask()` however, reads the ADC value into adc and then prints it to the serial monitor. After that is done, it is pushed on to the *back* of the queue using the function `xQueueSendToBack()`. We specify the queue handle in the first argument. The value to be queued is provided using a pointer in the second argument. The last argument indicates how long to wait if the queue is full. By specifying the macro `portMAX_DELAY`, we indicate that we want the execution to block until there is room in the queue.

The display task

Now lets examine the display task in the following code:

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

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

This function, like all task functions receives a pointer argument when it starts. Here it is unused and is NULL because of the way that the task was created.

The main body of the task is a for loop receiving from the queue. Here `xQueueReceive()` returns the queued data value, but otherwise waits forever if the queue is empty (the third argument value is `portMAX_DELAY`). Once a value is received from the queue, the bar graph display is updated and the LED PWM value is changed.

Note that there is no need to call `delay()` in this display task. The reason is that the task will automatically stall waiting in function `xQueueReceive()` when the queue is empty. The pace of the execution will be set by the sending task.

Running the demo

When wired up and flashed, you should see the bar graph display and the LED light (see **Figure 3**). If the LED is dark, turning the pot clockwise should brighten it. If you compiled the program without a display, then startup the serial monitor

Web Links

[1] ESP32 API reference: https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/

[2] FreeRTOS homepage: www.freertos.org

[3] Project source code: https://github.com/ve3wwg/esp32_freertos/blob/master/freertos-tasks1/freertos-tasks1.ino

[4] ESP32 Lolin Board with OLED: www.elektor.com/lolin-esp32-oled-module-with-wifi

and look for lines of output of the form “ADC 3420” etc. Turning the pot counter clockwise should result in the dimming of the PWD LED and clockwise should brighten it. Likewise, counter clockwise should display reduced ADC values and clockwise increased values.

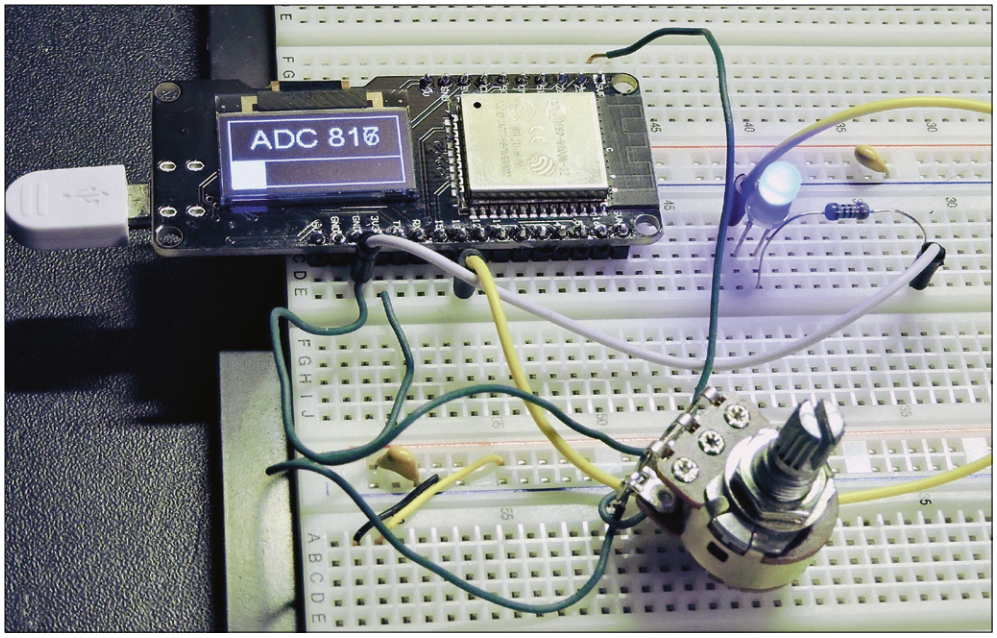


Figure 3: Running the demo.

Summary

We’ve covered a lot of ground in this article. You’ve seen the ESP32 startup procedure from `app_main()`, to `setup()` and `loop()`. The creation of the main task named `loopTask` was covered. In the demo we created our own additional task for the purpose of controlling the OLED and PWM LED, and communicated data to it using a queue. This code runs nicely independent from the main task.

Additionally, we’ve made use of the main task knowing that a fair amount of stack space is allocated to it (from the heap). In a future instalment, we’ll discuss how to determine stack usage for new tasks that you want to create. This example was trivial in terms of complexity. But recognize that tasks make complicated applications simpler by subdividing the problem. Consider how a MIDI controller might be divided into sending, receiving and control tasks. The receiving task is then free to receive incoming serial data and decode them into events for the control task to execute. Some control events may in turn result in sending serial data from the sending task. This is a neat subdivision of labour. Not only does this make the maintenance of the code a joy but it improves upon program correctness. ⏮

190182-01