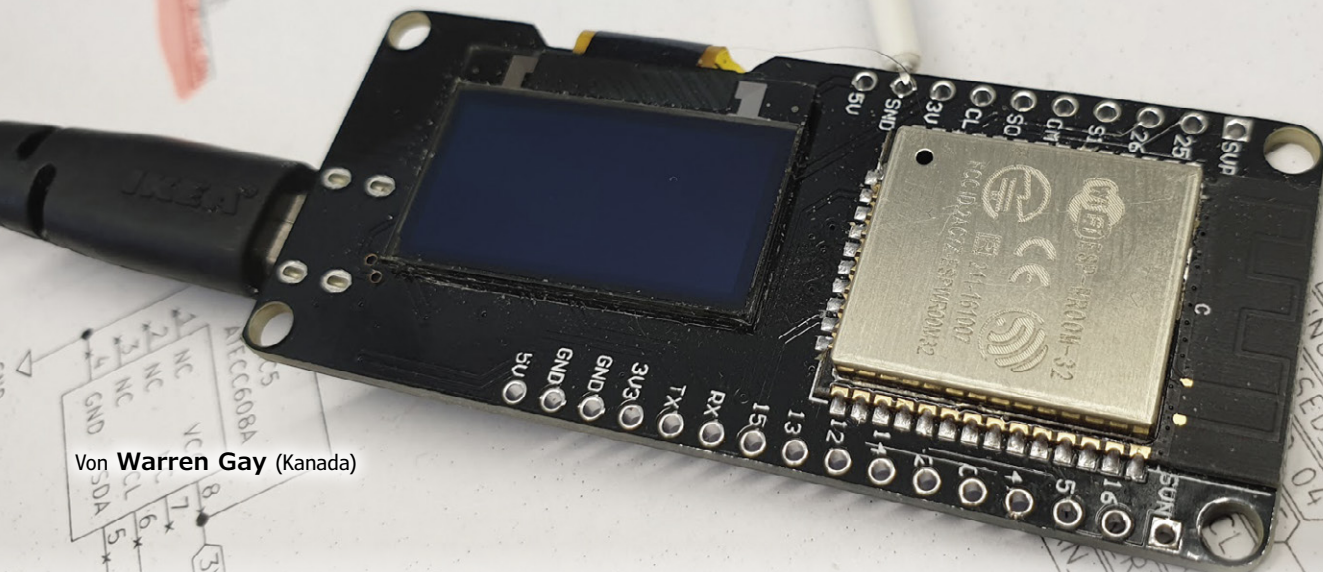


# Praktisches ESP32-Multitasking

## Task-Programmierung mit FreeRTOS und der Arduino-IDE



Von **Warren Gay** (Kanada)

Wenn ein Mikrocontroller als Drehscheibe eines Projekts verwendet wird, haben Entwickler oft das Problem, dass mehrere Aufgaben gleichzeitig ausgeführt werden müssen: Abtasten von Sensorwerten, Steuern von Stellgliedern, Visualisieren von Gerätezuständen und/oder Warten auf die Eingabe eines menschlichen Nutzers. Glücklicherweise kann dieses Problem mit der Task-Programmierung auf Basis leichtgewichtiger Embedded-Betriebssysteme auf sehr elegante Weise gelöst werden. FreeRTOS ist ein solches einfaches und weit verbreitetes Open-Source-Betriebssystem, das für viele Mikrocontroller-Plattformen verfügbar ist. Das sehr beliebte Tandem ESP32 und Arduino-IDE macht es besonders einfach, FreeRTOS für die Task-Programmierung zu nutzen, da es bereits in die Core-Bibliotheken integriert ist. Vermutlich haben Sie FreeRTOS schon immer benutzt, ohne es zu wissen!

Die ESP32-Plattform der Firma Espressif ist ein spannender Mikrocontroller für Maker-Projekte. Seine Hardwarefähigkeiten, die umfangreiche Software-Unterstützung und der moderate Preis machen ihn für viele zu einem „must have“. Es ist sowohl eine Arduino-kompatible als auch eine native Entwicklungsumgebung (ESP-IDF, Espressif IoT Development Framework) verfügbar. In diesem Artikel wird die vertraute Arduino-Umgebung verwendet. Der größte Teil des API (Application Programming Interface) ist für beide Entwicklungsumgebungen geeignet [1]. Einige unserer Leser wissen vielleicht, dass Espressif das beliebte Open-Source-Embedded-Betriebssystem FreeRTOS verwendet, um die ESP32-Bibliotheksfunktionen für WLAN, Bluetooth und viele weitere Funktionen des Mikrocontrollers zu realisieren. In diesem Artikel werden wir sehen, dass FreeRTOS [2] und die Task-Programmierung auch für die einfachen Arduino-Funktionen `setup()` und `loop()` verwendet wird. Zuerst werden wir ein einfaches ESP32-Demo-Projekt beschreiben [3], dann schauen wir hinter die Kulissen und werden später unsere eigenen Tasks für die Demo festlegen.

### Applikation

Für diesen Artikel haben wir eine kleine Testanwendung entwickelt, die die Stellung eines Potis mit dem Analog-Digital-Wand-

ler (ADC) einliest und in einem OLED-Display als Balkendiagramm anzeigt. Zusätzlich steuern wir eine LED mit Pulsweitenmodulation (PWM) an und variieren sie ja nach Potistellung von schwach bis hell leuchtend.

Dieser Artikel verwendet das ESP32-Board von Lolin, das ein integriertes OLED-Display besitzt (**Bild 1**) [4]. Wenn Sie ein anderes ESP32-Board verwenden, das über kein SSD1306-kompatibles OLED-Display verfügt, können Sie das OLED-Display im Programm deaktivieren und sich stattdessen auf die Helligkeitssteuerung der LED konzentrieren.

Die LED und das Potentiometer werden wie in **Bild 2** an den ESP32 angeschlossen.

Wenn die Anwendung läuft, kann durch Drehen des Potentiometers die Eingangsspannung an GPIO36 verändert werden. Diese Spannung wird mit einer Auflösung von 12 Bit abgetastet, der Wert liegt also zwischen 0...4095. Wenn Sie das Poti im Uhrzeigersinn bis zum Anschlag drehen, wird ein Wert von 4095 angezeigt und die LED leuchtet hell. Sollte das nicht so sein, vertauschen Sie einfach die äußeren Anschlüsse des Potis. Der mittlere Anschluss ist der Schleifer des Potis, an dem man Spannungen im Bereich 0...+3,3 V messen kann.

Achten Sie darauf, dass Sie das Poti nicht an der +5-V-Versorgung anschließen, da dies die maximal erlaubte Spannung des GPIOs überschreitet.

### Programm-Konfiguration

Kommen wir nun zur Software. Unser Programm definiert C-Makros (siehe **Listing 1**), um eine Neukonfiguration zu vereinfachen. Setzen Sie Makro `CFG_OLED` auf Null, wenn Sie keine Anzeige verwenden.

Der Wert des Makros `CFG_ADC_GPIO` bestimmt, dass Kanal 0 (GPIO36) des ADC1 verwendet wird. `CFG_LED_GPIO` konfiguriert GPIO13 für den Anschluss der LED.

Der vollständige Quellcode ist bei github im Unterverzeichnis `freertos-tasks1` [3] verfügbar.

### Initialisierung

Abgesehen von den Tasks findet sich in **Listing 2** das `setup()` für unsere Anwendung, eine typische Initialisierung in einem Arduino-Sketch.

Die Anzeige wird nur initialisiert, wenn der entsprechende Code kompiliert wurde. Danach wird der ADC mit der Arduino-Routine `analogReadResolution()` so konfiguriert, dass er 12-Bit-Werte liest. Die Funktion `analogSetAttenuation()` wird aufgerufen, um einen Wert von 0...+3,3 V zu lesen. Der ADC besitzt nämlich einen internen Verstärker, so dass diese Konfiguration wichtig ist, damit der richtige Bereich verwendet wird.

Danach wird der GPIO-Pin für die LED (`gpio_led`) als Ausgang und durch Aufrufe von `ledcAttachPin()` und `ledcSetup()` für eine PWM konfiguriert.

### Balkenanzeige

Über den OLED-Displaycontroller SSD1306 zeichnet die Applikation ein Balkendiagramm, das den ADC-Wert anzeigt (siehe **Listing 3**).

#### Listing 1. Konfigurationsoptionen.

```
// 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
```

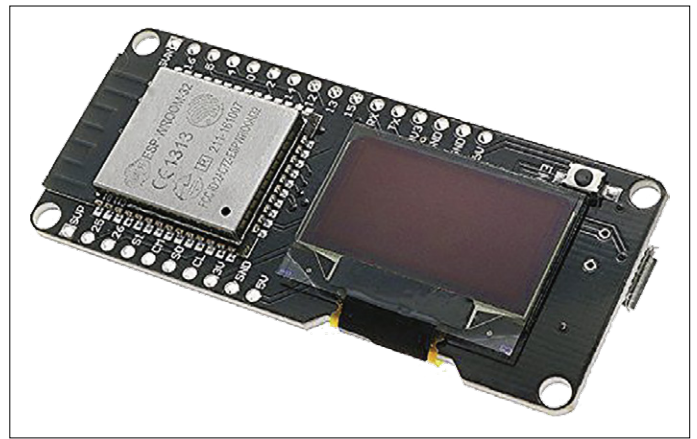


Bild 1. Lolin-ESP32 mit OLED-Display.

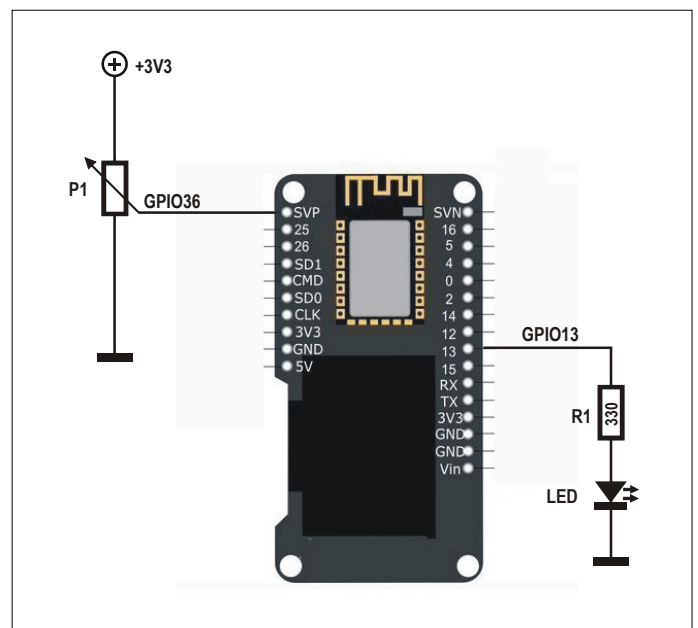


Bild 2. Lolin-ESP32-Modul mit Potentiometer und LED.

#### Listing 2. Die `setup()`-Funktion.

```
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. Die Balkenanzeige-Funktion.

```
#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.setTextAlign(TEXT_ALIGN_CENTER);
    display.setFont(ArialMT_Plain_24);
    display.drawString(64, 5, buf);
    display.display();
}
```

Die Funktion `barGraph()` akzeptiert einen Eingangswert `v` (den ADC-Wert) und formatiert diesen mit `snprintf()` zu einer kurzen Nachricht im Array `buf`. Die Rechtecke werden teils schwarz und teils weiß „gezeichnet“, um den ADC-Wert als Balkendiagramm darzustellen. In der Methode `display.drawString()` wird auch ein formatierter Text auf dem Display platziert. Am Ende wird die Methode `display.display()` aufgerufen, um das Anzeigebild aus dem Arbeitsspeicher in den OLED-Controller zu übertragen.

### Die loop()-Funktion

Würde es sich um ein normales Arduino-Programm handeln, so könnten wir eine Schleife wie in **Listing 4** schreiben. In diesem Code sind die Schritte einfach:

- Lesen des 12-Bit-ADC-Werts (von 0 bis 4095)
- Anzeige im Balkendiagramm (wenn konfiguriert)
- Ausgabe von „ADC“ und des Wertes im seriellen Monitor
- LED-Helligkeit durch Ändern des PWM-Parameters einstellen
- Verzögerung um 50 Ticks.

### Listing 4. Typischer Schleifencode bei Arduino.

```
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);
}
```

**Tabelle 1.**  
Tasks, die beim Arduino-Start ausgeführt werden.

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

Das Programm ist bewusst einfach gehalten. Aber wenn diese Anwendung komplizierter wäre, würde man sie in kleinere Aufgaben (Tasks) unterteilen.

### Was sind Tasks?

Mit einer ESP32-CPU mit zwei Kernen können zwei Programme *gleichzeitig* ausgeführt werden. Das ist eine spannende Angelegenheit! Jede CPU arbeitet mit einem eigenen Programmzähler und nutzt andere Register zur Ausführung von Anweisungen. Dies entspricht zwei Ausführungssträngen (*Threads*), im Gegensatz zu einer Single-Core-CPU mit nur einem Thread zu jedem Zeitpunkt.

Um noch mehr als zwei Programm-Aufgaben (*Tasks*) simultan auszuführen, wird ein Trick verwendet, nämlich eine Aufgabe auszusetzen und eine andere aufzunehmen. Dies ist die Hauptaufgabe von FreeRTOS. Dieses Wechselspiel vollzieht sich mit einer solchen Geschwindigkeit, dass es so erscheint, als würden die Tasks gleichzeitig ausgeführt, auch wenn in Wirklichkeit zu einem bestimmten Zeitpunkt (bei einer Dual-Core-CPU) nicht mehr als zwei Tasks gleichzeitig ausgeführt werden.

Um die Ausführung mehrerer scheinbar gleichzeitiger Tasks zu verwalten, speichert die FreeRTOS-Zeitsteuerung die Register des aktuellen Tasks und stellt die Register des nächsten auszuführenden Tasks (wieder) her. Auf diese Weise können mehrere Aufgaben unabhängig voneinander ausgeführt werden. Zusätzlich zum *program counter register* für jeden Task müssen auch die *stack pointer* gespeichert und wiederhergestellt werden. Dies ist notwendig, da C/C++-Programme Variablen und Return-Funktionsadressen auf dem Stack speichern. Jeder Task besitzt natürlich einen eigenen Stack.

Auf dem ESP32 laufen alle Tasks im gleichen Speicherbereich (im Gegensatz zum Beispiel zum Raspberry Pi). Aus diesem Grund muss ein Task sehr sorgfältig programmiert werden, damit er den von anderen Tasks verwendeten Speicherplatz nicht beschädigt. Wenn Sie einen Multi-CPU-Kern verwenden, gibt es noch andere Überlegungen, die aber nicht Thema dieses Artikels sein sollen (sondern eines künftigen).

### Arduino-Tasks

Wenn Ihr ESP32 ein Arduino-Programm startet, ist das dann ein Task? Auf jeden Fall! Zusätzlich zu Ihrem eigenen Task beim Starten werden noch FreeRTOS-Tasks ausgeführt. Einige dieser Tasks stellen Dienste wie Timer, TCP/IP oder Bluetooth bereit, zusätzliche Tasks können vom Anwender gestartet werden.

**Tabelle 1** zeigt ein Beispiel für FreeRTOS-Tasks, die ausgeführt werden, wenn die Arduino-Funktionen `setup()` und `loop()` aufgerufen werden. Der *loopTask* ist der Haupt-Arduino-Task. Die Spalte mit der Bezeichnung Stack zeigt die von FreeRTOS



nicht genutzten Stack-Bytes. Die Planung der Stackgrößen wird zu einem späteren Zeitpunkt behandelt.

Die Tabelle ist in umgekehrter chronologischer Reihenfolge nach Task-Nummern sortiert. Der Haupt-Task (*loopTask*) ist der zuletzt erstellte Task. Fehlende Task-Nummern deuten darauf hin, dass einige Tasks gestartet und beendet wurden, als ihre Aufgabe erledigt war. Die Spalte *Priority* zeigt die zugewiesenen Taskprioritäten (mit Null als niedrigster Priorität). Im Moment sollten Sie nur im Hinterkopf behalten, dass Prioritäten in FreeRTOS etwas anders funktionieren als beispielsweise unter Linux. Schließlich werden die Aufgaben zwischen CPU 0 und CPU 1 aufgeteilt. Espressif lässt seine Support-Tasks in CPU 0 ablaufen, während Applikations-Tasks CPU 1 verwenden. Dadurch laufen die Dienste für TCP/IP, Bluetooth und ähnliche reibungslos und ohne besondere Beeinflussung durch Ihre Anwendung. Trotz dieser Konvention können Sie aber beiden CPUs zusätzliche Tasks zuweisen.

### Arduino-Startup

Es ist gut zu wissen, wie der ESP32 Arduino-Programme vor dem Aufruf von `setup()` initialisiert. Betrachten Sie den (vereinfachten, die Watchdog-Timer-Elemente wurden weggelassen) Programmschnipsel in **Listing 5**. Aus diesem Beispiel können wir einige interessante Dinge lernen:

- Beachten Sie, dass es sich um ein C++-Startup handelt (daher die externe „C“-Deklaration von `app_main()`).
- Einige Arduino-Initialisierungen werden von `initArduino()` durchgeführt.
- Der `loopTask` wird durch den Aufruf von `xTaskCreatePinnedToCore()` mit einer Reihe von Argumenten erstellt und ausgeführt.

Das erste Argument ist die Adresse der Funktion, die für den Task (`loopTask`) ausgeführt werden soll. Wenn der Task erstellt wird, führt die Funktion `loopTask()` zunächst einen Aufruf von `setup()` und dann von `loop()` aus einer „ewigen“ For-Schleife aus.

Das zweite Argument ist ein String, der den Namen des Tasks als C-String („`loopTask`“) beschreibt. Das dritte Argument gibt die Stackgröße mit 8192 Bytes an. Beachten Sie, dass sich FreeRTOS hier von anderen Plattformen unterscheidet, die stattdessen 4-Byte-Wörter verwenden. Wenn der Task erstellt wird, wird diese Menge von Bytes aus dem Heap-Speicher dem Task zugeordnet, bevor die Funktion aufgerufen wird. Eine nicht unbeträchtliche Menge an SRAM wäre verschwendet, wenn der Main-Task nicht verwendet werden würde.

Alle FreeRTOS-Tasks akzeptieren einen Void-Pointer als Argument. In diesem Fall wird der Wert nicht verwendet und auf NULL gesetzt. Das fünfte Argument ist die FreeRTOS-Priorität, die dem Task zugewiesen wird, in diesem Beispiel „1“. Bevor die FreeRTOS-Prioritäten weiter erläutert werden, verwenden wir erst einmal die Priorität 1.

Nach dem Prioritätsargument ist ein Pointer auf eine Variable gesetzt, die das Handle zum Task enthält. Hier wird das Handle nicht verwendet und auf NULL gesetzt. Das letzte Argument gibt an, auf welchem CPU-Kern der Task laufen soll. Beim dualen ESP32 kann es „0“ oder „1“ sein. CPU 1 wird verwendet, um den Main-Task zu starten. Bei Single-Core-Prozessoren kann das Argument nur Null sein.

### Erstellen eines Tasks

Stellen wir uns einmal vor, unsere Anwendung wäre kompliziert und wir müssten den Code in zwei Tasks unterteilen. Wir

#### Listing 5. Vereinfachter ESP32-Arduino-Start.


```
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
}
```

haben bereits den `loopTask()`, der immer wieder `loop()` aufruft. Um den ihm zugewiesenen Stackplatz zu nutzen, würden wir dort normalerweise unseren stackintensivsten Programmteilstück programmieren.

In unserem Beispieltask führen wir in der Funktion `loop()` folgendes aus:

Anzeige



**CDS1**

**Configurable Display Switch**

- Frei konfigurierbares Eingabesystem
- Vollflächiger Touchscreen
- Rundes OLED Display
- Plug and Play

**SCHURTER**  
ELECTRONIC COMPONENTS

CDS1.schurter.ch

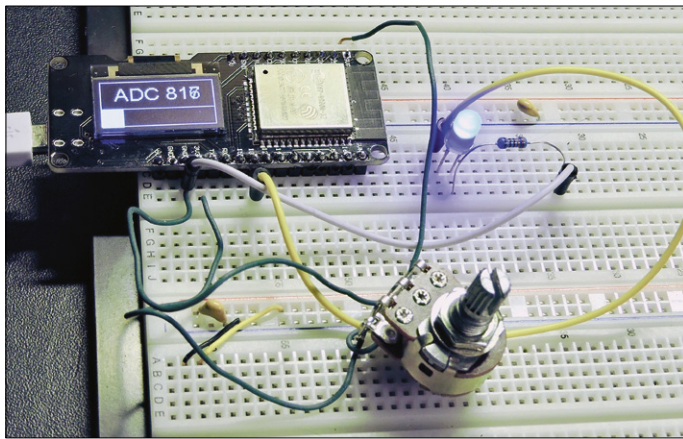


Bild 3. Ausführen der Demo.

- 12-Bit-Wert aus dem ADC lesen
- Wert auf den seriellen Monitor ausgeben
- ADC-Wert an den zweiten Task senden
- 50 Ticks Pause und zurück (von `loop()`)

Damit bleibt für den zweiten Task zu tun:

1. ADC-Wert von der `loop()`-Funktion (im `loopTask`) holen
2. ADC-Wert im Bargraph anzeigen (wenn konfiguriert)
3. PWM für die LED-Helligkeit entsprechend ändern

Der einzig sinnvolle Ort, um den zweiten Task zu erstellen, ist der `setup()`-Task, da diese Aufgabe nur einmal durchgeführt werden muss. Lassen Sie uns also zum `setup()`-Task hinzufügen:

```
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
}
```

Dieser Call startet einen neuen Task namens `dispTask` mit einem Stack von 2048 Bytes. Wir haben ihn der CPU 1 mit der Priorität 1 zugewiesen. Er wird die (noch nicht definierte) Funktion `dispTask()` ausführen. Da der Task auf derselben CPU und mit derselben Priorität wie unser `loopTask()` ausgeführt wird, wird die CPU zwischen `dispTask()`, `loopTask()` und allen anderen Tasks der Priorität 1 geteilt.

## Queues

Es fehlt noch ein Bestandteil: Wie senden wir den ADC-Wert von einem Task zu einem anderen? Sollen wir etwa etwas mit dem Speicher versuchen, was für Tasks auf derselben CPU funktionieren könnte? Aber die Angelegenheit ist hier komplizierter, weil Informationen von einer CPU an die andere weitergegeben werden. Die Lösung des Problems sind FreeRTOS-Queues! Die Queue ermöglicht es einem Task, ein Element auf „atomare“ Weise zu speichern. Es ermöglicht dem Empfänger ebenfalls, dieses Element auf atomare Weise abzuholen. Mit „atomar“ ist gemeint, dass das Element nicht nur teilweise gesendet oder empfangen werden kann. Bei einer Dual-Core-CPU mit zwei simultan ausgeführten Befehlen kann es Timing-Probleme und Race Conditions geben. Der Queue-Mechanismus ergreift besondere Maßnahmen, damit die Information atomar übermittelt wird. Eine FreeRTOS-Queue wird wie folgt erstellt:

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

Das erste Argument betrifft die Queuetiefe (die maximale Anzahl der aufnehmbaren Queueeinträge). Das zweite Argument gibt die Größe jedes in der Queue aufgenommenen Elements an. In diesem Fall ist es die Größe eines `uint`-Wertes in Bytes. Der Rückgabewert ist das Handle der Queues. Die Queue sollte unmittelbar vor der Erstellung des `dispTask` in `setup()` generiert werden, damit der Wert sofort verfügbar ist.

Bauen wir nun die Queue in die Schleife ein:

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

    printf("ADC %u\n", adc);
    xQueueSendToBack(qh, &adc, portMAX_DELAY);
    delay(50);
}
```

## Weblinks

- [1] ESP32 API-Referenz: <https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/>
- [2] FreeRTOS-Homepage: <http://www.freertos.org>
- [3] Quellcode des Projekts: [https://github.com/ve3wwg/esp32\\_freertos/blob/master/freertos-tasks1/freertos-tasks1.ino](https://github.com/ve3wwg/esp32_freertos/blob/master/freertos-tasks1/freertos-tasks1.ino)
- [4] ESP32-Lolin-Board mit OLED: [www.elektor.de/lolin-esp32-oled-module-with-wifi](http://www.elektor.de/lolin-esp32-oled-module-with-wifi)

Das Balkendiagramm und die LED-PWM werden in unserer neuen Funktion `dispTask()`, die immer noch nicht beschrieben ist, aktualisiert. Der `loopTask()` liest jedoch den ADC-Wert in `adc` ein und übergibt ihn dem seriellen Monitor. Danach wird er mit der Funktion `xQueueSendToBack()` auf die „Rückseite“ der Queue geschoben. Wir spezifizieren das Queue-Handle im ersten Argument. Der Wert, der in die Queue gestellt werden soll, wird mit einem Pointer im zweiten Argument bereitgestellt. Das letzte Argument gibt an, wie lange gewartet werden soll, wenn die Warteschlange voll ist. Durch die Angabe des Makros `portMAX_DELAY` geben wir an, dass die Ausführung blockiert werden soll, bis Platz im Queue ist.

### Der Display-Task

Nun untersuchen wir den Display-Task mit dem folgenden Code:

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

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

Diese Funktion erhält, wie alle Task-Funktionen, beim Start ein Pointer-Argument. Hier wird es nicht genutzt und ist deshalb NULL, da der Task so angelegt wurde.

Der Hauptteil des Tasks ist eine for-Schleife. Hier gibt `xQueueReceive()` den Datenwert der Queue zurück, wartet aber ansonsten „ewig“, wenn die Queue leer ist (der dritte Argumentwert ist `portMAX_DELAY`). Nach dem Empfang eines Wertes aus der Queue wird die Balkenanzeige aktualisiert und der LED-PWM-Wert geändert.

Beachten Sie, dass in diesem Display-Task kein `delay()` aufgerufen werden muss. Der Grund dafür ist, dass der Task automatisch in der Funktion `xQueueReceive()` wartet, wenn die Queue leer ist. Das Tempo der Ausführung wird also vom sendenden Task bestimmt.

### Ausführen der Demo

Nach dem Aufbau der Schaltung und der Programmierung sollten Sie die Balkenanzeige und das LED-Licht sehen (**Bild 3**).

Wenn die LED dunkel ist, sollte ein Dreh am Poti sie erhellen. Wenn Sie das Programm ohne Displaynutzung kompiliert haben, dann starten Sie den seriellen Monitor und schauen Sie die ausgegebenen Zeilen an, in der Form „ADC 3420“. Dreht man das Poti gegen den Uhrzeigersinn, wird die PWD-LED dunkler, mit dem Uhrzeiger heller. Ebenso werden die ADC-Werte niedriger beziehungsweise höher.

### Zusammenfassung

Wir haben in diesem Artikel die ESP32-Startprozedur kennen-gelernt. Auch die Erstellung des Haupttasks namens `loopTask` wurde abgedeckt. In der Demo haben wir einen eigenen zusätzlichen Task zur Steuerung des OLED-Displays und der LED-PWM erstellt und ihr über eine Queue Daten übermittelt. Dieser Code läuft unabhängig vom Haupttask.

Zusätzlich haben wir beim Haupttask gelernt, dass ihm (aus dem Heap) ziemlich viel Stackplatz zugewiesen wird. In einem zukünftigen Artikel werden wir besprechen, wie Sie den Stack für neue, noch zu erstellende Tasks bestimmen können.

Zugegeben, das Beispiel war nicht besonders komplex. Aber Sie haben erkannt, dass Tasks komplizierte Anwendungen vereinfachen können, indem Sie das Problem in kleine Portionen aufteilen. So können beispielsweise bei einem MIDI-Controller Sende-, Empfangs- und Steuerungsaufgaben separiert werden. Der empfangende Task kann sich auf die Aufgabe konzentrieren, eingehende serielle Daten zu einzulesen und in Events zu dekodieren, die dann vom Steuerungstask ausgeführt werden. Einige Steuerungsevents können wiederum dazu führen, dass serielle Daten vom Sendetask gesendet werden. Dies ist eine vorbildliche Arbeitsteilung und erhöht nicht nur die Korrektheit des Codes. Auch die Wartung des Programms wird sehr erleichtert. ◀

190182-02



#### IM ELEKTOR-STORE

→ **Lolin-ESP32-Modul mit OLED-Display**

[www.elektor.de/lolin-esp32-oled-module-with-Wi-Fi](http://www.elektor.de/lolin-esp32-oled-module-with-Wi-Fi)



**Leiterplatten online**  
konfigurieren & bestellen

[www.emsproto.com](http://www.emsproto.com)



BERECHNEN  
SIE IHREN  
PREIS  
**ONLINE**



LIEFERUNG  
**2 bis 12**  
TAGE



ANZAHL  
**1 bis 50**  
STÜCK

