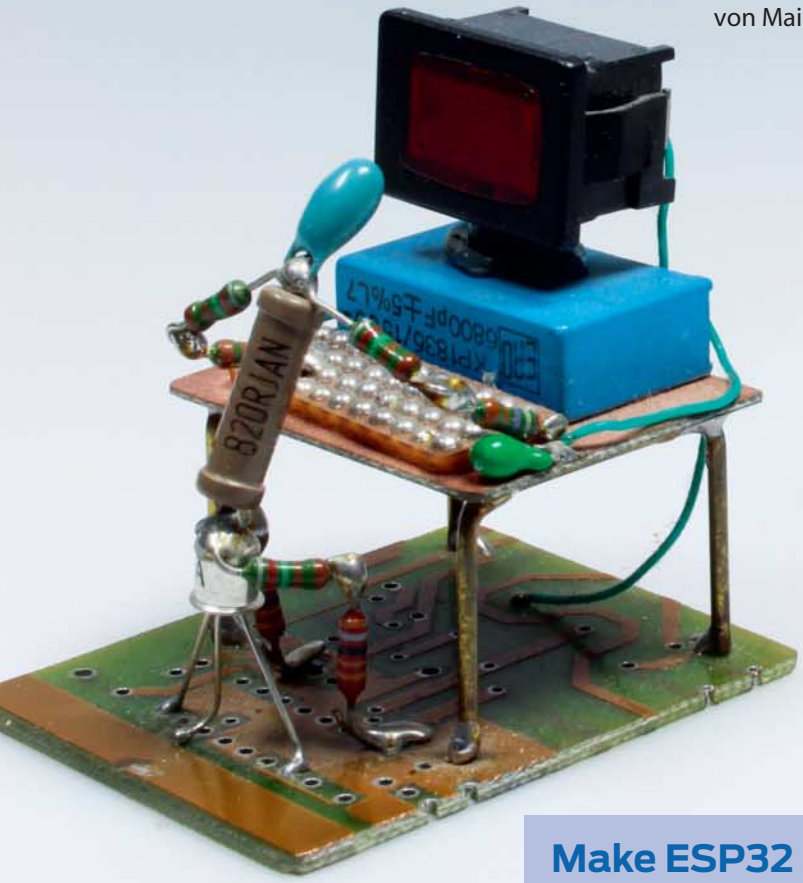
# ESP32 – Vollzugriff

**Die Arduino-IDE bietet einen einfachen Zugang zum ESP32 und ist perfekt für erste Experimente. Wer jedoch die volle Kontrolle übernehmen will, kommt am Entwickler-Framework ESP-IDF von Espressif nicht vorbei.**

von Maik Schmidt





## Make ESP32 Special

Das erste Make ESP32 Special erklärt die Hardware-Grundlagen des populären Wifi-Mikrocontrollers, hilft beim Programmiereinstieg und bringt IoT-Projektideen mit.

Von der Einrichtung in der Arduino-IDE bis zum WLAN-Server für eine Temperaturüberwachung erklären wir im neuen Make ESP32 Special den Umgang mit dem Wifi-Mikrocontroller. Das Heft ist ab sofort exklusiv und versandkostenfrei im heise shop erhältlich: Das Bundle mit einem ESP32 NodeMCU Entwicklerboard gibt es für 24,95 Euro zu bestellen.


Der erste Teil des 68 Seiten starken Hefts widmet sich den Grundlagen: Wir zeigen, wie Sie den ESP32 schnell über die beliebte Arduino-Programmierumgebung ansteuern. Was steckt in der Hardware der Wifi-Mikrocontroller? Das erklären wir und auch, wie Sie den ESP32 ins WLAN bringen oder per Bluetooth Daten austauschen. Anschließend geht es um weitere Funktionen des Boards vom Hallsensor bis zu Digital-Analog-Wandlern.

Das gedruckte Heft mit dem ESP32 gibt es exlusiv im heise shop für 24,95 Euro versandkostenfrei zu bestellen. Ohne Mikrocontroller ist das Special als PDF für 14,90 Euro erhältlich.

## Kurzinfo

- Espressif IoT Development Framework installieren
- Mit dem ESP-IDF eigene Software entwickeln
- Zugriff auf hardwarenahe Funktionen

### Checkliste

**Zeitaufwand:**
1 Stunde

**Programmieren:**
C/C++

Alles zum Artikel im Web unter
*make-magazin.de/xzuf*

### Hardware

- ESP32
- Breadboard

### Software (kostenlos)

- ESP-IDF

Für Hersteller von Mikrocontrollern ist nicht nur die Hardware ihrer Produkte wichtig. Sie müssen auch eine vernünftige Entwicklungsumgebung zur Verfügung stellen, um neue Nutzer anzulocken und von ihrem Produkt zu überzeugen. Ein Mikrocontroller, der sich nur umständlich programmieren lässt, hat am Markt keine große Chance.

Das gilt auch für den ESP32, und für die Firma Espressif stand eine einfache Programmierung ganz oben auf der Feature-Liste. Allerdings war die Arduino-IDE nicht die eigentliche Ziel-Plattform. Die originäre Plattform zur Entwicklung von Software auf dem ESP32 ist das ESP-IDF (IoT Development Framework). Das ist eine Sammlung von Werkzeugen, Bibliotheken und Dokumentationen, die alles beinhaltet, was man zur Programmierung des ESP32 braucht. Das IDF ist die Basis für das Projekt arduino-esp32, das die Entwicklung von ESP32-Software mit der Arduino-IDE ermöglicht.

Bevor man das IDF nutzen kann, muss man es installieren und konfigurieren. Das ist geringfügig aufwendiger als die Installation der Arduino-IDE.

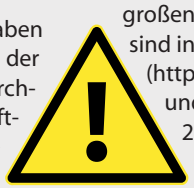
### Her damit!

Die Toolchain, die Espressif für die Arbeit mit dem ESP32 gewählt hat, stammt aus dem Linux- beziehungsweise Unix-Umfeld. Damit wäre es für Linux- und macOS-Nutzer kein großes Problem, die notwendigen Programme und Bibliotheken schrittweise zu installieren. Ein Großteil ist ohnehin auf vielen Systemen bereits installiert.

Um es den Nutzern aber bequemer zu machen und um sicherzustellen, dass alle dieselbe Umgebung verwenden, stellt Espressif Installationsdateien für die wichtigsten Betriebssysteme bereit. Diese enthalten alle benötigten Programme und müssen im Wesentlichen nur heruntergeladen und ausgepackt werden. Die wenigen zusätzlichen Schritte für Linux (https://docs.espressif.com/projects/esp-idf/en/stable/get-started/linux-setup.html) und macOS (https://docs.espressif.com/projects/esp-idf/en/stable/get-

## Bevor eine Sicherung durchbrennt

Bei älteren ESP32-Modulen (beim ESP32-Dev-kit NodeMCU beispielsweise) kann man durch Verbinden des GPIO-Pin 12 mit dem 3,3V-Pin in die ROM-Konsole statt in das gespeicherte Programm booten und das interne Basic starten.



Aktuelle Versionen des ESP-IDF haben die unangenehme Eigenheit, eine der Sicherungen (eFuse) im ESP32 durchbrennen zu lassen, wenn neue Software aufs Board gespielt wird. Sobald diese Sicherung durchgebrannt ist, kann man nicht mehr auf die ROM-Konsole und ihren Basic-Interpreter zugreifen.

Es gibt aber eine einfache Möglichkeit, die Sicherung vor dem Durchbrennen zu schützen. Wer also später noch Zugriff auf die ROM-Konsole haben möchte, sollte dies unbedingt tun, sofern sein Modul noch ins ROM booten kann! Bei aktuellen Modellen des ESP32 ist die Fuse leider bereits ab Werk durchgebrannt.

Zur Manipulation der eFuses bietet Espressif das Werkzeug `espefuse.py` an, das automatisch mit dem `esptool` (https://github.com/espressif/esptool) installiert wird. Das ist eine Sammlung von Programmen, die im täglichen Umgang mit dem ESP32 großen Nutzen stiftet. Alle Programme sind in der Programmiersprache Python (https://www.python.org/) geschrieben und funktionieren sowohl mit Python 2.7 als auch mit Python 3.4.

Wer die ESP-IDF-Toolchain installiert hat, hat mit großer Wahrscheinlichkeit auch einen Python-Interpreter. Das gilt insbesondere für die MinGW-Installation unter Windows. Mit dem Python-Paketmanager `pip` kann man `esptool` wie folgt installieren:

```
pip install esptool
```

Auf manchen Umgebungen, unter anderem unter MinGW, führt dieser Befehl zu einem Fehler. Dort kann man alternativ das folgende Kommando verwenden:

```
python -m pip install esptool
```

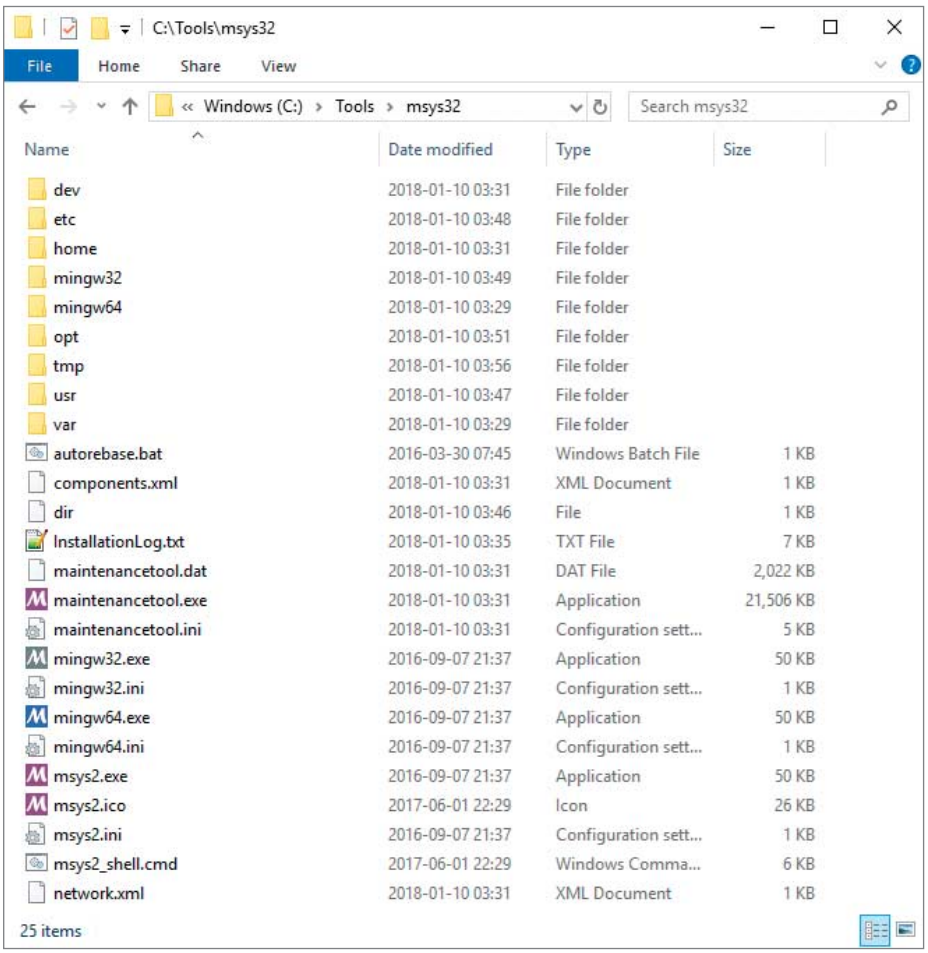
Anschließend muss man mit `espefuse.py` den Schreibschutz für die Sicherung mit dem Namen `CONSOLE_DEBUG_DISABLE` aktivieren:

```
espefuse.py -port <port-name>
write_protect_efuse
CONSOLE_DEBUG_DISABLE
```

Der Parameter port-name muss durch den Namen der seriellen Schnittstelle ersetzt werden, mit der das ESP32-Board verbunden ist. Hängt das Board beispielsweise unter Windows an Port COM6, dann lautet die Anweisung:

```
espefuse.py -port COM6 write_pro-
tect_efuse CONSOLE_DEBUG_DISABLE
```

Weil die Auswirkungen des Kommandos unumkehrbar sind, muss man die Aktion bestätigen, indem man das Wort BURN (alles in Großbuchstaben) eingibt. Anschließend kann die ROM-Konsole nie mehr deaktiviert werden.




**Unix-/Linux-Nutzern dürfte die Verzeichnisstruktur von MSYS32 bekannt vorkommen.**

started/macos-setup.html) dokumentiert der Hersteller online ausführlich.

Für Windows-Nutzer ist der Prozess ähnlich einfach, dürfte für die meisten Anwender aber etwas ungewohnt sein. Das liegt daran, dass die Entwicklungsumgebung des IDF dem Microsoft-Betriebssystem im Grunde ein Unix-System vorgaukelt.

Das ist notwendig, weil Windows an vielen Stellen gänzlich anders als Unix/Linux funktioniert. Daher würden die Programme, die zur Entwicklungsumgebung gehören, nicht ohne umfangreiche Modifikationen funktionieren.

## Cygwin

Dankenswerterweise haben sich ein paar findige Programmierer schon vor langer Zeit die Mühe gemacht, eine Unix-Kompatibilitätsschicht für Windows zu bauen. Die heißt Cygwin und macht es recht einfach, Unix-Programme auf die Windows-Plattform zu portieren.

Auch wenn das mit Cygwin etwas leichter ist, ist es angesichts der Vielzahl an Programmen noch immer eine ganze Menge Arbeit. Die hat das Team des Projekts MinGW (Minimalist GNU for Windows) erledigt und die wichtigsten Unix-Programme auf die Windows-Plattform gebracht. Dazu gehören unter anderem Bash (Bourne Again Shell) und GCC (GNU Compiler Collection).

Das reicht aber immer noch nicht ganz, denn zur Entwicklung von Software werden oft noch mehr Programme und insbesondere ein Paketmanager benötigt. Ferner wäre es schön, wenn man die ganze Software nicht selbst zusammenstellen müsste, sondern als Gesamtpaket bekommen könnte. Genau das hat sich das Projekt MSYS2 auf die Fahne geschrieben. Damit ist es verhältnismäßig leicht, ein individuelles Softwarepaket für die Entwicklung von Software auf Windows-Systemen zu schnüren.

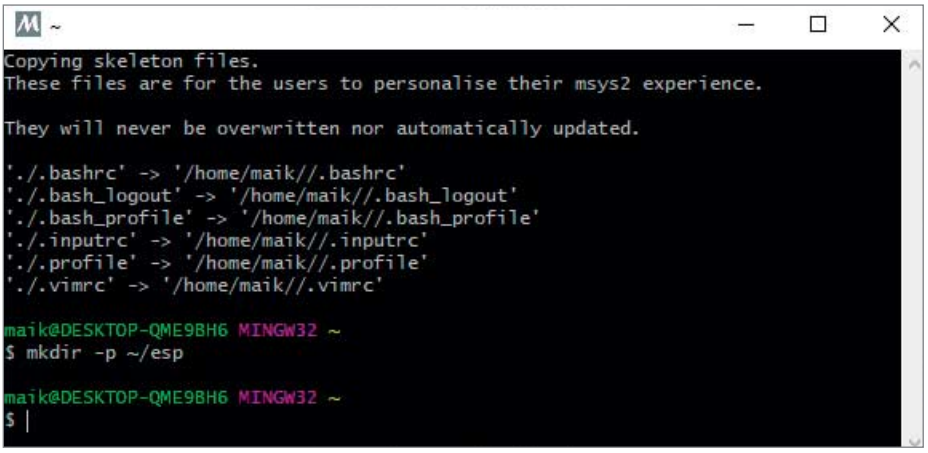
Die Firma Espressif nutzt MSYS2, um Windows-Nutzern dieselben Programmier-Werkzeuge zur Verfügung zu stellen, wie den Nutzern von Linux und macOS. Dazu muss man lediglich eine Zip-Datei herunterladen (https://dl.espressif.com/dl/esp32_win32_msys2_environment_and_toolchain-201801001.zip) und entpacken. Das Zip-Archiv ist knapp ein halbes Gigabyte groß und ausgepackt beanspruchen die Dateien circa 1,5 GB. Das ausgepackte Verzeichnis hat den Namen msys32 und man kann es an eine beliebige Stelle kopieren.

## Noch mehr Installationen

Mit der Installation der Entwicklungswerkzeuge ist schon viel erreicht, aber das eigentliche IDF-Framework fehlt noch und muss separat installiert werden. Dieser und alle weiteren Schritte sind dank der homogenen Toolchain für alle Betriebssysteme gleich.

Während man jedoch unter Linux und macOS automatisch eine Terminal-Anwendung hat, muss man unter Windows zunächst die Datei `mingw32.exe` aufrufen, die sich im Verzeichnis `msys32` befindet. Diese Datei öffnet ein Terminal-Fenster mit einer Bash-Shell.

In diesem Terminal ruft man `mkdir -p ~/esp` auf, um im Home-Verzeichnis des aktuellen Benutzers ein Unterverzeichnis mit dem Namen `esp` anzulegen. Die gesamte Entwicklung mit dem IDF findet in diesem Verzeichnis statt.




**Eine erste Begegnung mit dem MinGW-Terminal.**

In dieses Verzeichnis wird auch das eigentliche IDF installiert. Dazu wechselt man mit dem Kommando `cd ~/esp` in das Verzeichnis und kopiert aus dem Github-Verzeichnis übers Internet das Projekt auf die lokale Festplatte:

```
git clone -b v3.2 -recursive
https://github.com/espressif/
esp-idf.git
```

Das IDF wird mit dem Versionskontrollsystem Git verwaltet und das vorhergehende Kommando kopiert die Version 3.2 des IDF in das aktuelle Verzeichnis. Der Vorgang dauert einen Moment und erzeugt recht wüste Ausgaben. Am Ende liegt das IDF im Verzeichnis `esp-idf`. Selbstverständlich können auf diese Weise auch andere Versionen installiert werden und es lohnt sich zu prüfen, ob es schon eine neue stabile Version gibt.

Bevor die Entwicklung endlich losgehen kann, muss man noch die Umgebungsvariable `IDF_PATH` setzen, so dass sie auf das Verzeichnis `esp-idf` verweist. Unter Linux und macOS muss dazu die folgende Zeile an die Datei `~/.profile` angehängt werden:

```
export IDF_PATH=~/esp/esp-idf
```

Das setzt natürlich voraus, dass das IDF auch im Verzeichnis ~/esp/esp-idf liegt. Wer das IDF an eine andere Stelle kopiert hat, muss die Variable entsprechend anpassen.

Unter Windows ist das Prozedere ähnlich, stiftet aber schnell Verwirrung. Die Umgebungsvariable wird nämlich nicht dort eingetragen, wo Umgebungsvariablen normalerweise definiert werden. Sie muss im Profil der msys32-Installation eingetragen werden, denn das ist ja die Umgebung, in der die eigentliche Entwicklung stattfindet.

Wenn die Installation zum Beispiel im Verzeichnis `C:\Tools\msys32` liegt, dann muss im Verzeichnis `C:\Tools\msys32\etc\profile.d` eine neue Datei angelegt werden. Deren Name ist im Grunde egal, aber sie muss die Extension `.sh` haben. Eine gute Wahl ist zum Beispiel `export_idf_path.sh`. In diese Datei trägt man die folgende Zeile ein:

```
export
IDF_PATH=C:/Tools/msys32/home/
eigener-user-name/esp/esp-idf
```

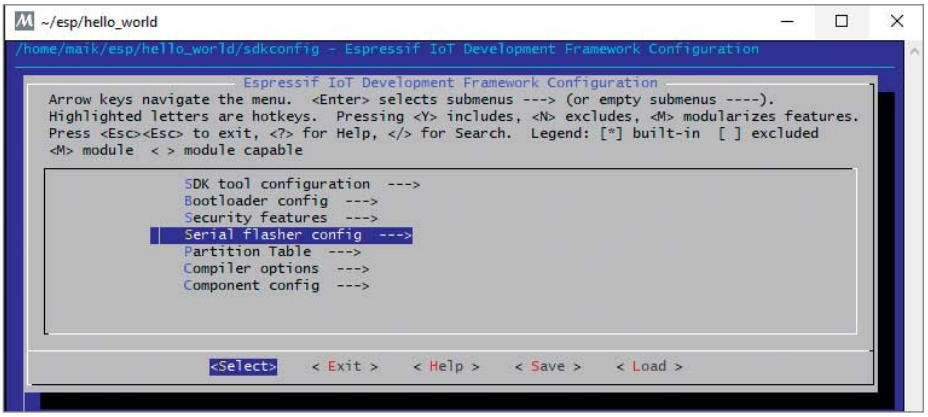
Dabei ist es ganz wichtig, den richtigen Installationspfad – in diesem Fall ist er `C:\Tools\msys32` – anzugeben. Darüber hinaus muss *eigener-user-name* durch den tatsächlichen Benutzernamen in der MinGW-Umgebung ersetzt werden. Schließlich müssen statt der Backslashes (\) hier Slashes (/) verwendet werden, weil MinGW den Unix-Dateisystemkonventionen gehorcht.

Nachdem die Variable ins Profil eingetragen wurde, ist ein Neustart des Terminals erforderlich. Damit ist die Installation des IDF und seiner Werkzeuge abgeschlossen. Der ganze Vorgang mag auf den ersten Blick kompliziert erscheinen, ist aber innerhalb weniger Minuten erledigt. Am Ende steht eine Umgebung, die sich unter allen Betriebssystemen gleich verhält.

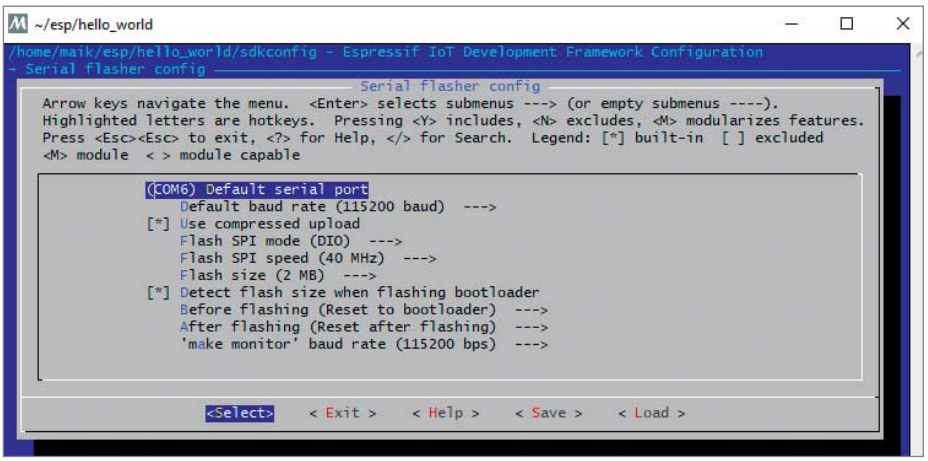
## Hallo?

Nach der Installation wird es Zeit für einen ersten Test und das IDF bringt jede Menge Beispiel-Projekte mit. Das klassische „Hello world"-Programm zählt ebenfalls dazu und soll als Erstes zum Laufen gebracht werden.

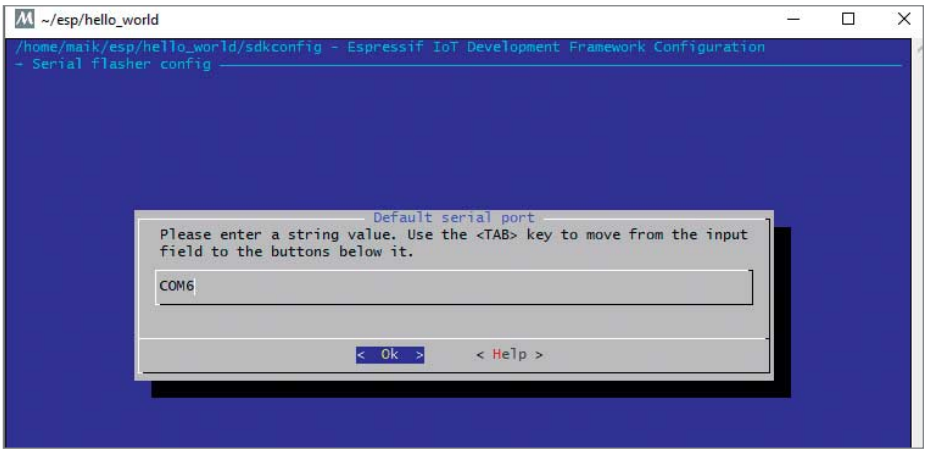
Das Programm gibt die Nachricht „Hello world!" und ein paar Informationen über den



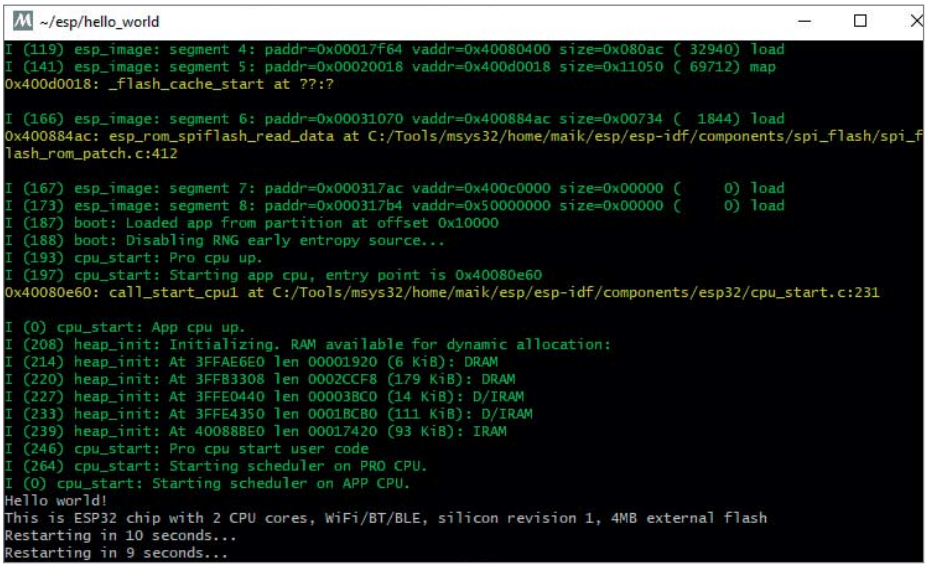

**Projekte werden grafisch konfiguriert.**




**Die serielle Schnittstelle kann auf vielfältige Weise konfiguriert werden.**




**Der Name der seriellen Schnittstelle muss fast immer angepasst werden.**




**Der eingebaute serielle Monitor reicht für viele Zwecke völlig aus.**

ESP32 auf der seriellen Schnittstelle aus. Anschließend startet es den ESP32 nach zehn Sekunden neu. Es liegt im examples-Verzeichnis des IDF und wird zunächst mit den folgenden Anweisungen in das eigene Arbeitsverzeichnis kopiert. Wichtig ist der Punkt am Ende des zweiten Kommandos:

```
cd ~/esp
cd hello_world
cp -r $IDF_PATH/examples/get-
started/hello_world .
```

Bevor man ein Programm nun übersetzen und auf den ESP32 transferieren kann, muss man noch ein paar Dinge konfigurieren. Zum Beispiel muss man angeben, an welcher seriellen Schnittstelle der ESP32 hängt. Das erfolgt über ein menügesteuertes Konfigurationsprogramm. Gestartet wird es mit dem Befehl `make menuconfig`.

Die Oberfläche weckt Erinnerungen an die neunziger Jahre, als man Linux-Kernet oft noch selbst kompilieren musste, aber das Programm erfüllt seinen Zweck. Im Hauptmenü gibt es den Menüpunkt „Serial flasher config“ und mit dem kann die zu verwendende serielle Schnittstelle festgelegt werden. Durch die Menüs bewegt man sich mit den Cursor-Tasten und die Return-Taste wählt einen Menüpunkt aus. Ferner bringt die Tab-Taste den Cursor zum nächsten Menüpunkt.

Im Menü „Serial flasher config“ kann man diverse Parameter für die serielle Kommunikation anpassen. Für einen ersten Test ist lediglich sicherzustellen, dass der Name der seriellen Schnittstelle der richtige ist. Unter Windows ist das ein Name, der mit COM beginnt und es ist derselbe, den man auch in der Arduino-IDE auswählen würde.

Ist der Name korrekt gesetzt, speichert man die Konfiguration mit dem Menüpunkt Save. Den vorgeschlagenen Dateinamen `sdkconfig` kann man getrost übernehmen. Anschließend beendet man das Konfigurationsprogramm mit dem Menüpunkt Exit.

Das Kommando `make flash` übersetzt das Programm und kopiert es auf den ESP32. Das Übersetzen dauert beim ersten Mal sehr viel länger als mit der Arduino-IDE. Das liegt daran, dass viele Bibliotheken gebaut werden, die der Arduino-IDE schon fertig beiliegen. Wenn man das Programm ein weiteres Mal übersetzt, geht es sehr viel schneller, weil dann nur noch die Teile übersetzt werden, die sich geändert haben.

Wenn alles fehlerfrei funktioniert, läuft das Test-Programm auf dem ESP32. Leider ist nicht zu erkennen, ob es auch tatsächlich tut, was es soll, das heißt, dass es einen Text auf der seriellen Schnittstelle ausgibt.

Das könnte man mit jedem seriellen Monitor, wie zum Beispiel PuTTY oder auch dem seriellen Monitor der Arduino-IDE, leicht prüfen, aber das IDF kommt mit einem eigenen kleinen Werkzeug daher. Ruft man nämlich `make monitor` auf, werden die Nachrichten, die der ESP32 an die serielle Schnittstelle sendet, im Terminal ausgegeben. Als Erstes gibt der Monitor ein paar Diagnose-Nachrichten aus, aber irgendwann erscheint der Text „Hello world!“. Nach einer Wartezeit von zehn Sekunden startet der ESP32 neu.

Damit wurde ein typischer Entwicklungszyklus einmal komplett durchlaufen. Das Programm wurde übersetzt, auf das Board übertragen und anschließend wurden dessen Ausgaben im seriellen Monitor inspiziert.

## Was steht denn drin?

Offen ist noch, wie der Quelltext des Programms aussieht und wo er steckt? Wenig überraschend liegt er im Verzeichnis `hello_world` und zwar in einem Unterverzeichnis namens `main`. Die Datei heißt

`hello_world_main.c`

**hello_world_main.c**

```
 1 #include <stdio.h>
 2 #include "freertos/FreeRTOS.h"
 3 #include "freertos/task.h"
 4 #include "esp_system.h"
 5 #include "esp_spi_flash.h"
 6
 7 void app_main() {
 8   printf("Hello world!\n");
 9
10   esp_chip_info_t chip_info;
11   esp_chip_info(&chip_info);
12   printf("This is ESP32 chip with %d CPU cores, WiFi%s%s, ",
13          chip_info.cores,
14          (chip_info.features & CHIP_FEATURE_BT) ? "/BT" : "",
15          (chip_info.features & CHIP_FEATURE_BLE) ? "/BLE" : "");
16
17   printf("silicon revision %d, ", chip_info.revision);
18
19   printf("%dMB %s flash\n", spi_flash_get_chip_size() /
     (1024 * 1024),
20          (chip_info.features & CHIP_FEATURE_EMB_FLASH) ?
            "embedded" : "external");
21
22   for (int i = 10; i >= 0; i--) {
23     printf("Restarting in %d seconds...\n", i);
24     vTaskDelay(1000 / portTICK_PERIOD_MS);
25   }
26   printf("Restarting now.\n");
27   fflush(stdout);
28   esp_restart();
29 }
```

Alle IDF-Projekte sehen halbwegs gleich aus und haben dieselben Verzeichnisstrukturen, in denen ein paar Standard-Dateien zu finden sind. Daher gibt es eine leere Schablone (https://github.com/espressif/esp-idf-template), die als Ausgangspunkt für eigene Projekte dienen kann. Von ihr sollte man nur abweichen, wenn man weiß, was man tut.

Das Programm selber ist ein reines C-Programm, was an der Extension des Dateinamens leicht zu erkennen ist. Wie die meisten C-Programme beginnt es mit einer Reihe von `#include`-Anweisungen, die diverse Header-Dateien einbinden. Die Datei `stdio.h` dürfte zu den populärsten im C-Umfeld gehören, denn sie definiert alle notwendigen Funktionen zur Ein- und Ausgabe von Daten.

Es folgen zwei `#include`-Anweisungen, die jeweils Dateien aus dem Verzeichnis `freertos` referenzieren. FreeRTOS ist ein Echtzeit-Betriebssystem (Real Time Operating System), das speziell auf die Bedürfnisse von Mikrocontrollern abgestimmt ist. Es läuft auf vielen unterschiedlichen Plattformen und ist das Betriebssystem, das den ESP32 antreibt.

Die bisherigen `#include`-Anweisungen haben nur indirekt mit dem ESP32 zu tun. Die letzten beiden hingegen binden Funktionen ein, die speziell für den ESP32 entwickelt wurden und zum IDF gehören.

Es folgt die Funktion `app_main` und C-Kenner merken sofort, dass die Funktion `main`, die eigentlich der Haupteinstiegspunkt eines jeden C-Programms ist, fehlt. Der Grund dafür ist, dass das Programm nicht als natives Programm, sondern als Task im FreeRTOS-System ausgeführt wird. In der Praxis ist dieser Unterschied selten relevant.

Der Funktionsrumpf beginnt mit einem Aufruf der `printf`-Funktion, die zur Ausgabe von Daten auf dem Ausgabekanal `stdout` dient. In C-Programmen auf dem PC ist das in der Regel der Bildschirm. Weil man den auf dem ESP32 nicht voraussetzen kann, wird die Ausgabe praktischerweise auf die serielle Schnittstelle umgeleitet.

## Struktur

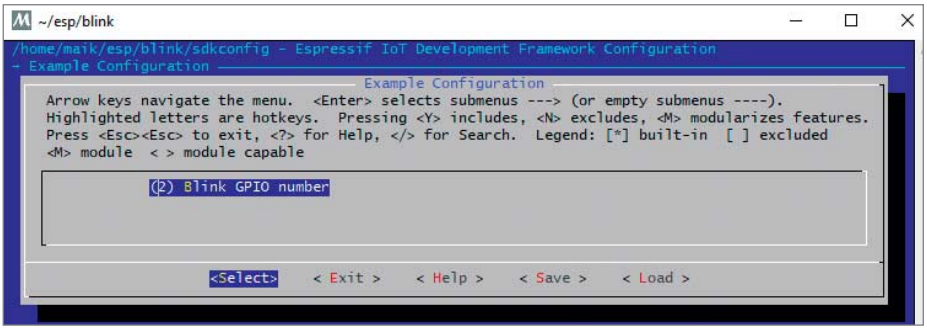
Weiter geht es mit der Ausgabe einiger Eigenschaften des ESP32. Dazu legt das Programm eine Struktur vom Typ `esp_chip_info_t` an. Diese Struktur bietet Platz für die wichtigsten Informationen über den ESP32, wie zum Beispiel die Anzahl der Prozessorkerne, die Bluetooth- und WLAN-Eigenschaften und die Versionsnummer des Chips. Gefüllt wird die Struktur mit der Funktion `esp_chip_info`. Anschließend werden die Informationen mittels `printf` und eines Formatstrings ausgegeben.

Eine kleine Besonderheit ist das Attribut `features` in der `esp_chip_info_t`-Struktur. Es ist eine Bit-Maske und um die einzelnen Informationen zu extrahieren, muss man den &-Operator (binäres Und) verwenden.

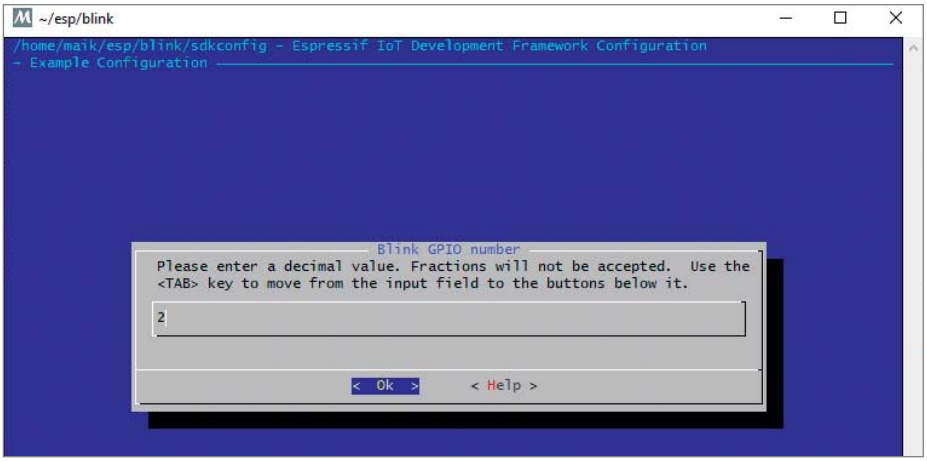
Analog erfolgt die Ausgabe der Informationen über den Flash-Speicher. Ob der Speicher in den ESP32 eingebaut ist oder extern angebunden wurde, erfährt man im

**blink.c**

```
 1 #include <stdio.h>
 2 #include "freertos/FreeRTOS.h"
 3 #include "freertos/task.h"
 4 #include "driver/gpio.h"
 5 #include "sdkconfig.h"
 6
 7 #define BLINK_GPIO CONFIG_BLINK_GPIO
 8
 9 void blink_task(void* pvParameter) {
10   gpio_pad_select_gpio(BLINK_GPIO);
11   gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
12
13   while (1) {
14     gpio_set_level(BLINK_GPIO, 0);
15     vTaskDelay(1000 / portTICK_PERIOD_MS);
16
17     gpio_set_level(BLINK_GPIO, 1);
18     vTaskDelay(1000 / portTICK_PERIOD_MS);
19   }
20 }
21
22 void app_main() {
23   xTaskCreate(&blink_task, "blink_task",
     configMINIMAL_STACK_SIZE, NULL, 5, NULL);
24 }
```




**Das Konfigurationsprogramm hilft auch bei der Anpassung von Programm-Parametern.**




**Hier wird der GPIO-Pin der LED von außen gesetzt.**

`features`-Attribut. Die Größe des Speichers liefert die Funktion `spi_flash_get_chip_size`.

Eine `for`-Schleife zählt von zehn runter bis zur 0 und informiert darüber, dass der ESP32 in Kürze neu gestartet wird. Die Funktion `vTaskDelay` wartet in jedem Schleifendurchlauf eine Sekunde, so dass bis zum Neustart insgesamt zehn Sekunden vergehen.

Vor dem Neustart mit der Funktion `esp_restart` sorgt ein Aufruf von `fflush` dafür, dass Daten, die sich noch im Ausgabepuffer befinden, an `stdout`, also an die serielle Schnittstelle, gesendet werden. So geht nichts verloren.

Zwar ist das Programm kurz, aber es berührt alle wichtigen Aspekte der ESP32-Programmierung. Es nutzt nämlich neben den herkömmlichen C-Bibliotheken auch die Funktionen des Betriebssystems FreeRTOS. Darüber hinaus macht es Gebrauch von den nützlichen APIs, die Espressif dem IDF mitgegeben hat. Wer den ESP32 mit dem IDF zähmen möchte, sollte sich in allen drei Bereichen gut auskennen.

Ohne gute Dokumentation sind die vielen schicken APIs aber nicht viel wert und auch in dieser Beziehung verhält sich der Hersteller vorbildlich. Auf dessen Webseite findet man neben guten Tutorials jede Menge Referenzmaterial zur Hardware und zu den APIs (https://docs.espressif.com/projects/esp-idf/en/latest/index.html). Die Informationen stehen als HTML-Seiten und als PDF-Dokumente bereit. Leider nur auf Englisch und Chinesisch.

Zusätzlich gibt es im examples-Verzeichnis zu allen Aspekten des ESP32 viele Beispielprogramme. Es lohnt sich also, dieses Verzeichnis genauer zu inspizieren, um ein Gefühl dafür zu bekommen, was alles möglich ist. Bei einer solchen Inspektion merkt man aber auch schnell, wie viel Komplexität die Arduino-Umgebung vor ihren Nutzern verbirgt, denn die Beispiel-Programme operieren oft auf sehr niedriger Ebene.

## Jetzt alle gleichzeitig

Einer der großen Unterschiede zwischen dem ESP32 und dem Arduino ist, dass auf dem ESP32 mit FreeRTOS ein Betriebssystem läuft. Damit ist unter anderem die Programmierung von Anwendungen, die mehrere Aufgaben parallel erledigen, vergleichsweise einfach.

Das wichtigste Hilfsmittel sind die RTOS-Tasks. Sie sind vergleichbar mit Prozessen auf anderen Betriebssystemen und werden von einer zentralen Instanz, dem Task-Scheduler, verwaltet.

Die grundlegende Verwendung der Tasks demonstriert das Listing blink.c, das die Status-LED des Entwicklungsboards zum Blinken bringt. Dazu nutzt es dieselben Mechanismen wie das klassische Arduino-Beispiel. Allerdings lagert es die Programm-Logik in einen Task aus.

Das Programm gehört zu den Standard-Beispielen des IDF und wird mit den folgenden Anweisungen ins Verzeichnis `esp` kopiert:

```
cd ~/esp
cblink
cp -r $IDF_PATH/examples/get-
started/blink.
```

Wie gewohnt werden zu Anfang alle notwendigen Header-Dateien eingebunden. Diesmal gehört auch der GPIO-Treiber dazu, denn schließlich soll ja eine LED ein- und ausgeschaltet werden. Das Präprozessor-Makro `BLINK_GPIO` definiert den GPIO-Pin, mit dem die Status-LED verbunden ist. Statt ihn jedoch auf einen konkreten Wert zu setzen, wird er durch die Zeichenkette `CONFIG_BLINK_GPIO` repräsentiert. Die wurde zuvor aber gar nicht definiert.

Das erscheint zunächst verwirrend, liegt aber in einer praktischen Eigenschaft des IDF begründet. Dort ist es nämlich möglich, solche Parametrisierungen mit Konfigurationsdateien beziehungsweise mit dem grafischen Konfigurationsprogramm vorzunehmen. Letzteres startet man – wie bei der Konfiguration der seriellen Schnittstelle – mit dem Kommando `make menuconfig .`

Wenn man schon mal dort ist, sollte man zunächst den Namen der seriellen Schnittstelle setzen, denn das muss man für jedes Projekt gesondert erledigen. Anschließend wählt man den Menüpunkt „Example Configuration“. Dann kann man den Punkt „Blink GPIO Number“ wählen und schließlich die Pin-Nummer, in diesem Fall die 2, eingeben. Dann wird das Ganze noch gespeichert und nach dem Verlassen des Konfigurationsprogramms wird beim Übersetzen die korrekte Pin-Nummer verwendet. Selbstverständlich kann man aber auch die Konstante im Quelltext direkt durch den Wert 2 ersetzen.

Im Programm selbst geht es mit der Funktion `blink_task` weiter. Der Name suggeriert schon, dass dies eine Funktion ist, die als Task ausgeführt werden soll. Deshalb bekommt sie auch einen Parameter vom Typ `void*`, mit dem der Task optional parametrisiert werden kann. Dass der Name der Funktion auf `_task` endet, ist übrigens reine Konvention.

Der Aufruf von `gpio_pad_select_gpio` sorgt dafür, dass der LED-Pin als GPIO-Pin

**multi.c**

```
 1 #include <stdio.h>
 2 #include "freertos/FreeRTOS.h"
 3 #include "freertos/task.h"
 4 #include "driver/gpio.h"
 5 #include "sdkconfig.h"
 6
 7 #define BLINK_GPIO CONFIG_BLINK_GPIO
 8
 9 void blink_task(void *pvParameter) {
10   gpio_pad_select_gpio(BLINK_GPIO);
11   gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
12
13   while(1) {
14     gpio_set_level(BLINK_GPIO, 0);
15     vTaskDelay(1000 / portTICK_PERIOD_MS);
16
17     gpio_set_level(BLINK_GPIO, 1);
18     vTaskDelay(1000 / portTICK_PERIOD_MS);
19   }
20 }
21
22 void hello_task(void *pvParameter) {
23   while (1) {
24     printf("Hallo, Make-Magazin!\n");
25     vTaskDelay(100 / portTICK_PERIOD_MS);
26   }
27 }
28
29 void app_main() {
30   xTaskCreate(&blink_task, "blink_task",
     configMINIMAL_STACK_SIZE, NULL, 5, NULL);
31   xTaskCreate(&hello_task, "hello_task",
     configMINIMAL_STACK_SIZE, NULL, 5, NULL);
32 }
```

fungiert. Das ist notwendig, weil der ESP32 regen Gebrauch vom IO-Multiplexing macht. Das bedeutet, dass die meisten Pins prinzipiell mehr als eine Aufgabe übernehmen können und deshalb muss man explizit festlegen, welche gerade gewünscht ist.

Wie in der Arduino-Umgebung auch, muss man noch festlegen, ob der GPIO-Pin als Eingang oder als Ausgang dienen soll. Die entsprechende Funktion heißt im IDF aber nicht `pinMode` sondern `gpio_set_direction`.

In einer Endlosschleife wird die Status-LED im Sekundentakt ein- und ausgeschaltet. Zur Änderung des Pin-Zustands dient die Funktion `gpio_set_level`. Sie ist das Äquivalent zur Arduino-Funktion `digitalWrite`.

## Ticks

Für die Pausen ist die Funktion `vTaskDelay` zuständig. Sie wartet eine vorgegebene Anzahl an Ticks. Die sind in einem Echtzeit-Betriebssystem eine wichtige Maßeinheit und sie werden in einem festen Zeitintervall hochgezählt. Der Task-Scheduler entscheidet anhand des aktuellen Tick-Zählerstands, welcher Task zur Ausführung kommt.

Weil die Dauer eines Ticks von der Taktfrequenz der CPU abhängt, muss man die Konstante `portTICK_PERIOD_MS` verwenden, um Millisekunden in Ticks umzurechnen. Übrigens schläft nur der Task für die angegebene Dauer. Der Rest des Systems arbeitet weiter, das heißt, die Ticks, die der Blink-Task verschläft, werden anderen Tasks zugeteilt.

Zum Schluss startet die Hauptfunktion `app_main` den Blink-Task mittels der Funktion `xTaskCreate`. Die erhält einige Parameter und der erste ist ein Zeiger auf die auszuführende Funktion. Der zweite ist ein frei wählbarer Name für den Task. Der hat keine große Bedeutung und dient maximal als Hilfe beim Debugging.

Wichtiger ist schon der dritte Parameter, denn der definiert die Größe des Stacks, der dem Task zur Verfügung steht. Dieser Wert wird nicht in Bytes, sondern in Datenworten angegeben. Auf einem 16-Bit-System beansprucht ein Datenwort zum Beispiel zwei Bytes. Setzt man den dritten Parameter auf einem solchen System auf den Wert 100, so werden für den Stack 200 Bytes reserviert. Auf einem 32-Bit-System wären es 400, weil dort ein Datenwort doppelt so groß ist. Für den Blink-Task reicht die voreingestellte Stack-Größe, die mit der Konstanten `configMINIMAL_STACK_SIZE` definiert wird, aus.

Mit dem vierten Parameter kann man dem Task einen Parameter in Form eines `void`-Zeigers mitgeben. Der nächste Parameter enthält die Priorität des Tasks. Je größer diese Zahl ist, umso wichtiger ist der Task. Konsequenterweise hat der Idle-Task, der immer dann läuft, wenn kein anderer Task läuft, die Priorität 0.

Schließlich kann man als letzten Parameter noch einen Zeiger auf eine `TaskHandle_t`-Struktur übergeben. Dieser Parameter wird mit einem Zeiger auf den erzeugten Task initialisiert, so dass der Task im weiteren Verlauf des Programms noch referenziert werden kann, um ihn beispielsweise zu löschen.

Lädt man das Programm mittels `make flash` auf den ESP32, wird die Status-LED blinken. Das ist im Grunde nichts besonderes, aber das Blinken wird in diesem Fall in einem Task erzeugt. Allerdings kann man das von außen nicht sehen. Zur Kontrolle muss daher noch ein zweiter Task her.

## So viele Aufgaben

Um die Multitasking-Fähigkeiten von FreeRTOS besser zu veranschaulichen, soll nicht nur die Status-LED des ESP32 blinken, sondern das Board soll parallel Nachrichten an die serielle Schnittstelle senden. Listing multi.c erledigt das und ist eine Erweiterung des Blink-Programms.

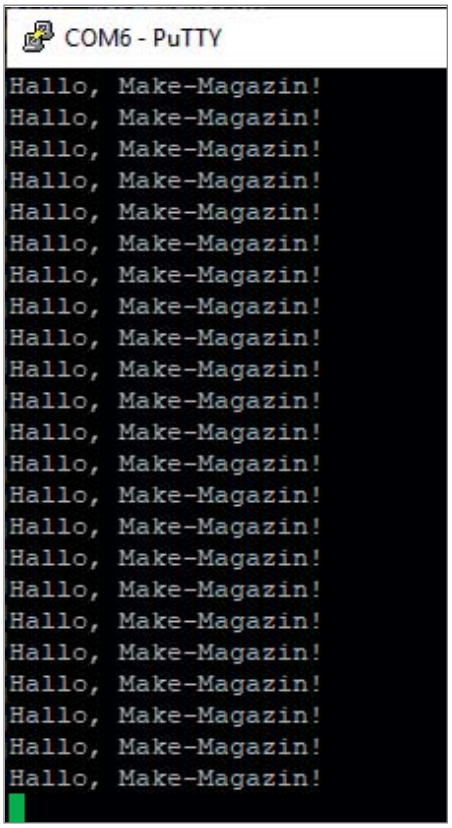
Neu ist im Wesentlichen die Funktion `hello_task`, deren Aufgabe es ist, alle 100 Millisekunden die Nachricht „Hallo, Make-Magazin!“ an die serielle Schnittstelle zu senden. Wie schon die `blink_task`-Funktion, erledigt auch diese Funktion ihre Arbeit in einer Endlosschleife.

Um den neuen Task zu starten, wurde die Funktion `app_main` um einen weiteren Aufruf von `xTaskCreate` ergänzt.

Nachdem man das Programm aufs Board geladen hat, sieht man zunächst keinen Unterschied, denn die Status-LED blinkt weiter tapfer vor sich hin. Startet man aber einen seriellen Monitor, zum Beispiel mittels `make monitor`, kann man sehen, dass kontinuierlich der gewünschte Text ausgegeben wird.

Das Betriebssystem übernimmt die vollständige Koordination der Tasks. Weil FreeRTOS ein Echtzeit-Betriebssystem ist, blinkt die LED pünktlich im Sekundentakt und die Nachricht wird alle 100 Millisekunden an die serielle Schnittstelle gesendet.

Dieses Programmiermodell passt hervorragend zu vielen typischen Mikrocontroller-Anwendungen, denn es erspart die oft aufwendige Friemelei mit Timern und Interrupts. Obendrein garantiert diese Vorgehensweise, dass vorgegebene Zeiten auch tatsächlich eingehalten werden. Ganz ohne Tücken ist dieses Modell aber nicht, denn spätestens, wenn verschiedene Tasks auf gemeinsame Ressourcen zugreifen, muss man sich Gedanken über deren Synchronisation machen.




**Die Ausgaben des Programms kann man sich mit jedem seriellen Monitor ansehen.**

## Fazit

Anfänger, die zuvor mit der Arduino-IDE gearbeitet haben, werden das ESP-IDF vermutlich zunächst als spartanisch und unbequem empfinden, weil es nicht einmal eine IDE bietet. Fortgeschrittene und Profis genießen vom ersten Moment an die Freiheit und die Flexibilität der Umgebung, denn sie können ihr gewohntes Werkzeug einsetzen und so gut wie alle Vorgänge vollständig automatisieren.

Wer mit dem IDF arbeitet, kommt darüber hinaus vor allen anderen in den Genuss der allerneuesten Erweiterungen und Funktionen. Schließlich bietet das IDF eine ganze Menge Features, die in der Arduino-Umgebung noch nicht zur Verfügung stehen.

Es ist aber auch wichtig zu verstehen, dass sich beide Umgebungen nicht ausschließen. Das heißt, man kann zum Beispiel RTOS-Tasks in der Arduino-IDE programmieren und umgekehrt Bibliotheken des arduino-esp32-Projekts im IDF verwenden. Am Ende ist die Entscheidung für viele eher eine Geschmacksfrage. *—dab*