Praktisches ESP32-Multitasking

Teil 2: Task-Prioritäten

Von **Warren Gay** (Kanada)

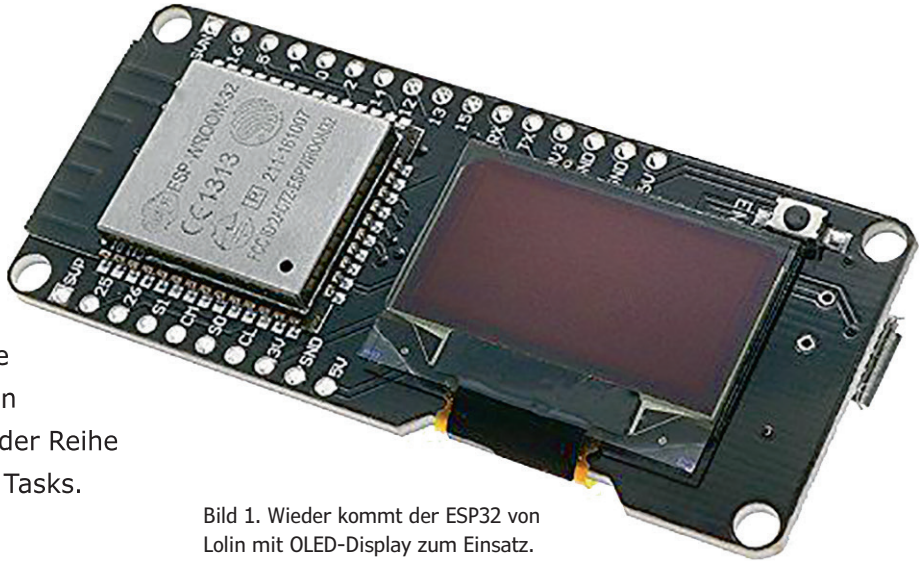

Bild 1. Wieder kommt der ESP32 von Lolin mit OLED-Display zum Einsatz.

In Mikrocontroller-Projekten stehen Entwickler oft vor dem Problem, dass viele Prozessoraufgaben gleichzeitig ausgeführt werden müssen. Der ESP32 und die Arduino-IDE machen die Task-Programmierung einfach, da das beliebte FreeRTOS bereits in die Core-Bibliotheken integriert ist [1]. In diesem zweiten Teil der Reihe befassen wir uns mit den Prioritäten der Tasks.

Bei der ESP32-Implementierung des FreeRTOS-Schedulers werden die Tasks entsprechend ihrer Priorität ausgeführt, die bei der Erstellung des Tasks zugewiesen wurde (was aber später geändert werden kann). Tasks mit höherer Priorität werden von der ausgewählten CPU zuerst berücksichtigt, während Tasks mit Null-Priorität als letzte beachtet werden. Die Ausführung von Prioritäten mag ein bekanntes Konzept sein, aber der FreeRTOS-Echtzeit-Scheduler funktioniert anders als Sie es von Linux oder Windows gewohnt sind. In diesem Artikel wird der Unterschied anhand beispielhafter Versuche demonstriert. Die ESP32-Implementierung umfasst maximal 25 Prioritätsstufen, die von Null bis 24 reichen. Standardmäßig laufen die Arduino-Funktionen `setup()` und `loop()` (erinnern Sie sich, diese Funktionen stammen aus demselben Main-Task [1]) auf dem Prioritätslevel 1.

Es lebe der Unterschied!

Wie unterschiedlich kann die Taskplanung sein? Auf einem Linux-System zum Beispiel beeinflusst die Priorität die relative Dringlichkeit des Prozesses oder Threads. Aber sogar ein Prozess mit niedrigster Priorität bekommt etwas CPU-Zeit ab: Normalerweise dauert es zwar länger, ihn auszuführen, aber der Prozess wird letztendlich doch abgehandelt. Und genau hier liegt der Unterschied.

In einem Echtzeitsystem wie FreeRTOS garantiert der Scheduler nicht, dass Aufgaben mit niedrigerer Priorität überhaupt jemals ausgeführt werden. Wenn Sie zum Beispiel Tasks mit der Priorität 9 haben, die immer *ready to execute* sind, dann wird kein Task mit Priorität 8 oder noch niedriger auf derselben CPU durchgeführt. Mit anderen Worten, die Tasks mit Priorität 9 lassen alle Tasks mit niedrigerer Priorität verhungern.

Ready to execute

Es ist wichtig zu verstehen, was *ready* für FreeRTOS bedeutet. Ein Task ist *ready*, wenn er nicht dadurch blockiert ist, dass er auf irgendetwas wartet, sei es ein Ereignis, ein (neuer) Eintrag in eine Queue oder ein *Mutex*, der entriegelt werden muss (was das ist, wird in einer späteren Artikelfolge behandelt). Ein ausführungsbereiter Task wird entsprechend seiner Priorität in die *Ready*-Liste des Schedulers aufgenommen und ausgeführt, wenn er an der Reihe ist. Da es sich um eine nach Priorität sortierte Liste handelt, werden die Aufgaben mit der höchsten Priorität zuerst abgehandelt.

Tasks mit gleicher Priorität werden nach einem *Round-Robin*-Verfahren [2] ausgeführt. Drei Ready-Tasks mit gleicher Priorität 9 (a, b und c) wechseln sich ab:

- task9a
- task9b
- task9c
- task9a
- task9b
- und so weiter

Solange diese Tasks nicht von Tasks höherer Priorität blockiert werden, geht das ewig so weiter. So kann sich ein ESP32-Task mit hoher Priorität namens *idc1* (für CPU 1) vor die Tasks mit Priorität 9 „drängeln“, um einige Aufgaben zu erledigen. Sobald der Task *idc1* fertig und wieder *not ready ist,* werden die Tasks mit Priorität 9 dort fortgesetzt, wo sie unterbrochen wurden.

Hier sind einige Beispiele dafür, warum ein FreeRTOS-Task **nicht** *ready* sein kann:

- *sleep* oder *delay* für eine Weile (auf einen Timer warten)
- warten auf Mutex oder Semaphore

- warten auf den Empfang einer Nachricht aus einer leeren Queue
- warten auf das Einfügen einer Nachricht in eine volle Queue
- warten auf ein FreeRTOS-Event oder eine Gruppe von Events
- warten auf den Abschluss einer I/O-Aktion
- *suspended* = ausgesetzt (entweder durch den Task selbst oder durch einen anderen Task).

Eine der Möglichkeiten, *blocked* zu werden, ist das Warten auf den Empfang einer Nachricht aus einer leeren Queue, denn ist sie leer, gibt es für diesen Task nichts zu tun. Der Scheduler entfernt diesen Task aus der Ready-Liste und sucht nach anderen, die ausgeführt werden sollen. Dabei werden nur Tasks auf der Ready-Liste berücksichtigt. Ist dort kein Task zu finden, wird stattdessen die Leerlauf-Task des Systems ausgeführt.
Beachten Sie, dass der Funktionsaufruf `taskYIELD()` nicht bei den Gründen in der Liste aufgeführt ist. Wenn ein Task die Kontrolle aufgibt, sei es, weil seine ihm zugestandene Zeit abgelaufen ist oder auch freiwillig durch den Aufruf von `taskYIELD()`, geht die Kontrolle wieder an den FreeRTOS-Scheduler über, so dass dieser einen anderen Task für den nächsten Zeitabschnitt auswählen kann. *Yielding* ist etwas anderes als *Blocking*, da diese Tasks weiterhin ausführungsbereit sind und bei der nächsten Gelegenheit wieder CPU-Zeit erhalten.

SMP-Änderungen von FreeRTOS in der ESP-IDF

FreeRTOS wurde für Single-Core-Mikrocontroller entwickelt. Da alle ESP32-Varianten (außer dem ESP32-S2) eine Dual-CPU besitzen, hat Espressif die Scheduler-Komponente angepasst. Kurz überblickt gibt es die folgenden ESP32-CPUs:
- CPU 0, bekannt als PRO_CPU (Protokoll-CPU) und
- CPU 1, bekannt als die APP_CPU (Applikations-CPU).

Espressif gibt an, dass „die beiden Kerne in der Praxis identisch sind und sich den gleichen Speicher teilen."
Zur Unterstützung von symmetrischem Multiprocessing (SMP) gibt Espressif an, dass „der Scheduler Tasks überspringt, wenn er eine Round-Robin-Planung zwischen mehreren Tasks gleicher Priorität im Ready-Zustand durchführt." Dies ergibt sich aus der Beschränkung, wenn man eine Ready-Liste, die für eine einzelne CPU entwickelt wurde, nun auf einer Plattform mit zwei CPUs [3] verwendet.
Das Problem, mit dem sich Expressif konfrontiert sah, war folgendes: Wenn eine CPU einen Task-Kontextwechsel benötigt, um den nächsten Ready-Task auszuführen, kann sie nur auf eine einzige Task-Ready-Liste zum Durchsuchen zurückgreifen. Wenn also der Index der aktuellen Liste auf bereite Tasks für die andere CPU zeigt, dann müssen diese Einträge übersprungen werden, bis ein Eintrag für die betreffende CPU gefunden werden kann. Unter diesen Bedingungen kann ein Round-Robin-Verfahren nicht immer optimal sein.
Das Fazit für den Entwickler ist also, dass das Round-Robin-Verfahren bei den zwei CPUs des ESP32 nicht ganz fair ist. Bei vielen Projekten dürfte dies nicht weiter auffallen, aber wenn es problematisch wird, muss man Möglichkeiten nutzen, dies zu umgehen. Seien Sie sich dessen bei der Task-Planung bewusst.

Ein Versuch

Wir werden jetzt ein Demo-Programm für das ESP32-OLED-Displaymodul (**Bild 1**) nutzen. Durch Änderung einiger Makros im Programm können Sie die Prioritäten von vier verschiedenen Tasks im Programm einstellen. Das Programm ist so konzipiert, dass es drei Spannerraupen (engl. Inchworm) zeigt, die im OLED horizontal hin und her kriechen. Jede der Raupen kriecht aber nur dann hin und her, wenn „ihr" Task CPU-Zeit erhält. Wird einem Task nur wenig CPU-Zeit gegönnt, so sitzt die Raupe still und kriecht nur ganz langsam.
Jede Raupe wird von einem Task bewegt, der seine CPU-Zeit aufbraucht und dann eine Nachricht an den vierten Task sendet. Dieser vierte Task ist dann dafür verantwortlich, dass die Raupe sich bewegt und angezeigt wird.
Der Code zum Zeichnen und Verwalten des Zustands der Raupe wird in der `InchWorm`-Klasse definiert (hier nicht gezeigt). In diesem Artikel konzentrieren wir uns einfach auf die Wirkung der Methode `InchWorm::draw()` für jede Raupe. Jede Instanz der `InchWorm`-Klasse verwaltet ihren eigenen Status und Fortschritt. Die Anzeige- und Raupeninstanzen werden im Programm wie folgt deklariert:

```
static Display oled;
static InchWorm worm1(oled,1);
static InchWorm worm2(oled,2);
static InchWorm worm3(oled,3);
```

Jedes Raupen-Objekt verwendet eine C++-Referenz (wie ein C-Pointer) auf die Display-Klasse im ersten und die Nummer der Raupe als zweites Argument. Die Referenz auf das Display ermöglicht eine zukünftige Erweiterung, zum Beispiel die Unterstützung mehrerer Displays. Die Nummer der Raupe bestimmt, wo sie (vertikal) auf dem OLED angezeigt wird (1, 2 und 3 bezeichnen die obere, die mittlere und die untere Zeile).
Der Task hinter jeder Raupe ist einfach eine Schleife, die die CPU-Zeit vergehen lässt, und ein Aufruf zum Senden der Nachricht:

```
void worm_task(void *arg) {
  InchWorm *worm = (InchWorm*)arg;

  for (;;) {
    for ( int x=0; x<800000; ++x )
      __asm__ __volatile__(„nop“);
    xQueueSendToBack(qh,&worm,0);
    // vTaskDelay(10);
  }
}
```

Es ist wichtig, die Funktion `vTaskDelay()` zunächst auskommentiert zu lassen. Sie wird in einem späteren Experiment verwendet werden.
Für alle drei Inchworm-Tasks wird dieselbe Task-Funktion verwendet, wobei das Argument `arg` angibt, welche Instanz der Raupe wir kriechen lassen wollen. Die Adresse der Raupe wird aus einem Void-Pointer umgewandelt und in der lokalen Variablen `worm` gespeichert. Sie wird nur innerhalb dieses Tasks verwendet, um als Nachricht an den Haupt-Display-Task gesendet zu werden, um anzugeben, welche Raupe sich hin- und herbewegt.
Beachten Sie, dass in diesem Versuch beim Aufruf von `xQueueSendToBack()` der Wartezeit-Parameter mit Null angegeben wurde (drittes Argument). Dies weist FreeRTOS an, sich

wenn möglich in die Queue einzureihen, schlägt aber sofort fehl, wenn die Queue voll ist. Dies ist auch beabsichtigt, denn wenn die Queue voll wird, wollen wir nicht, dass unser Raupen-Task deren Ausführung blockiert. Der Task darf die CPU in diesem Versuch nicht freigeben, damit er die CPU wirklich für sich beanspruchen kann.
Die äußere `for`-Schleife hat die Aufgabe, ihre Operationen für immer auszuführen. Die innere `for`-Schleife für die CPU-Wartezeit führt 800.000 Mal eine No-Operation-Operation (`nop`) aus. Das Schlüsselwort `__volatile__` verhindert, dass der Compiler dieses Loop-Statement wegoptimiert. Ungeachtet dessen, was der Compiler denken mag, ja, wir wollen die CPU-Zeit tatsächlich verschwenden!

Listing 1. Die Setup()-Funktion.

```
void setup() {
  TaskHandle_t h = xTaskGetCurrentTaskHandle();

  app_cpu = xPortGetCoreID(); // Which CPU?
  oled.init();
  vTaskPrioritySet(h,MAIN_TASK_PRIORITY);
  qh = xQueueCreate(4,sizeof(InchWorm*));

  // Draw at least one worm each:
  worm1.draw();
  worm2.draw();
  worm3.draw();

  xTaskCreatePinnedToCore(
    worm_task,  // Function
    “worm1”,    // Task name
    3000,       // Stack size
    &worm1,     // Argument
    WORM1_TASK_PRIORITY,
    nullptr,    // No handle returned
    app_cpu);

  xTaskCreatePinnedToCore(
    worm_task,  // Function
    “worm2”,    // Task name
    3000,       // Stack size
    &worm2,     // Argument
    WORM2_TASK_PRIORITY,
    nullptr,    // No handle returned
    app_cpu);

  xTaskCreatePinnedToCore(
    worm_task,  // Function
    “worm3”,    // Task name
    3000,       // Stack size
    &worm3,     // Argument
    WORM3_TASK_PRIORITY,
    nullptr,    // No handle returned
    app_cpu);
}
```

Nach Abschluss der Wartezeit-Schleife senden wir die Adresse der zu kriechenden Raupe an die Queue, identifiziert mit dem Handle `qh`. Sobald die Nachricht vom Display-Task empfangen wird, veranlasst er, dass unsere Raupe in der Anzeige vorrückt. Die Arduino-Haupttask `loop()` wird als der Display-Task verwendet, um das Kriechen der Raupe durchzuführen:

```
void loop() {
  InchWorm *worm = nullptr;

  if ( xQueueReceive(qh,&worm,portMAX_DELAY) )
    worm->draw();
}
```

Diese Schleife blockiert die Ausführung, bis einer der Tasks die Adresse der zu zeichnenden Raupe sendet. Sobald dieser Class-Pointer empfangen wird, wird die Methode `InchWorm::draw()` aufgerufen, um die Raupe zu zeichnen und voran kriechen zu lassen.
Die `setup()`-Funktion in **Listing 1** zeigt, wie die drei Raupen-Tasks und die Queue erstellt werden.

Ändern der Priorität

FreeRTOS erlaubt es einem Task, seine eigene oder die Priorität eines anderen Tasks mit Hilfe der Funktion `vTaskPrioritySet()` zu ändern. Standardmäßig läuft der Task, der `setup()` und `loop()` aufruft, auf der Prioritätsstufe 1 (diese Funktionen werden vom selben Haupttask aufgerufen). Für diese Demo muss diese Priorität höher sein als die der anderen drei Raupen-Tasks. Die `setup()`-Funktion ändert die Priorität ihres eigenen Tasks wie folgt:

```
static int app_cpu = 0; // Updated by setup()
...
void setup() {
  TaskHandle_t h = xTaskGetCurrentTaskHandle();

  app_cpu = xPortGetCoreID(); // Which CPU?
  ...
  vTaskPrioritySet(h,MAIN_TASK_PRIORITY);
```

Wie zu sehen, erhält die `setup()`-Funktion ihr eigenes Task-Handle durch den Aufruf von `xTaskGetCurrentTaskHandle()` und dessen Speicherung in `h`. Durch die Änderung der Haupttask-Priorität im Aufruf von `vTaskPrioritySet()` wird auch die von `loop()` verwendete Task-Priorität beeinflusst. Dies ist ein Beispiel dafür, wie Task-Prioritäten angepasst werden können. Im ersten Experiment werden den Raupen-Tasks die Prioritäten 9, 8 und 7 zugewiesen. Dies setzt voraus, dass unser Display- oder Haupt-Task die Priorität 9 oder höher besitzt (wir setzen sie auf 10). Wenn dies nicht getan wird, verliert der Haupttask `loop()` die CPU und kann die Raupen mehr nicht animieren.

Welche CPU?

Im `setup()`-Schnipsel wurde eine weitere ESP32-API-Funktion namens `xPortGetCoreID()` eingesetzt, um herauszufinden, auf welcher CPU die Applikation läuft. Dies wird im Programm in der statischen Variablen `app_cpu` gespeichert, so dass der Code

weiß, für welche CPU neue Tasks bestimmt sind. Für den Dual-Core-ESP32 ist der Wert von `app_cpu` 1, das heißt, die Applikation wird in einer Dual-Core-Konfiguration auf CPU 1 ausgeführt. Bei Single-CPU-Plattformen wird `app_cpu` auf null gesetzt. Durch diese Kodierung kann die Applikation portabel auf Plattformen mit Single- oder Dual-Core-CPUs ausgeführt werden.
Diese Demo dürfte jedoch aufgrund der Monopolisierung der CPU auf einer Einzel-CPU-Plattform nicht gut funktionieren. Der Watchdog-Timer würde auslösen und Resets verursachen. Sie haben nun aber gelernt, wie bei anderen Anwendungen grundsätzlich eine Portabilität erreicht werden kann.

Beispielhafte Konfiguration

Der Quellcode zu diesem Versuch ist unter [4] verfügbar. Am Anfang des Demonstrationsprogramms befinden sich Makrodefinitionen, die jedes Experiment konfigurieren:

```
// Worm task priorities
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 8
#define WORM3_TASK_PRIORITY 7

// loop() must have highest priority
#define MAIN_TASK_PRIORITY  10
```

Belassen Sie diese Prioritäten zunächst wie beim ersten Experiment.

Andere OLED-Anzeige

Falls Sie nicht das empfohlene ESP32 Lolin Board mit seinem integrierten OLED verwenden, können Sie die Displayeinstellungen selbst definieren:

```
Display(
    int width=128,
    int height=64,
    int addr=0x3C,
    int sda=5,
    int scl=4);
```

Wenn Ihre Einstellungen korrekt konfiguriert sind, sollte das OLED beim Programmstart sofort weiß werden. Wenn nicht, überprüfen Sie die Verbindungen und Einstellungen erneut.

Versuch 1

Sie können den heruntergeladenen Code einfach kompilieren, flashen und die Anwendung ausführen. Ihr OLED sollte sofort weiß werden und drei schwarze Raupen zeichnen (siehe **Bild 2**). Die Konfiguration für dieses Experiment ist (wieder):

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 8
#define WORM3_TASK_PRIORITY 7
#define MAIN_TASK_PRIORITY  10
```

Diese Konfiguration bewirkt, dass sich die obere Raupe entlangschlängelt, während die beiden unteren still stehen. Die Frage ist: Warum bewegen sich der mittlere und der untere Wurm nicht?

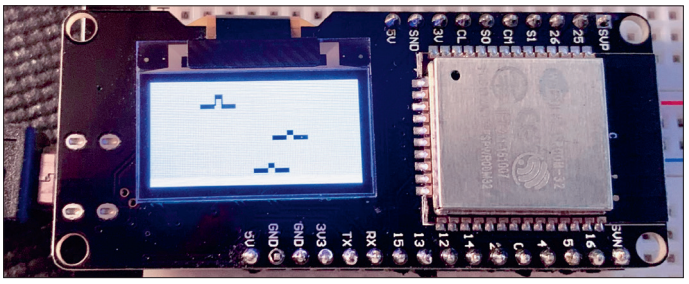

Bild 2. Wenn Tasks ausgeführt werden, bewegen sich die „Versuchstiere“.

```
  _¯_
_¯_
_¯_
```

Erinnern Sie sich, dass wir dem Haupt-Task die Priorität 10 belassen haben? Er genießt also die höchste Priorität aller Tasks in unserer Applikation. Die erste Raupe in der obersten Zeile des OLED konnte sich weiterbewegen, da er der einzige Task mit der Priorität 9 war, der durch die CPU ausgeführt werden konnte, weil der Display-Task mit Priorität 10 die Ausgabe zum OLED ausführt und dann darauf wartet, dass Nachrichten in der Nachrichten-Queue ankommen (und somit blockiert wird). Wenn aber der Display-Task blockiert ist, können andere Tasks mit niedrigerer Priorität eingeplant werden.
Die Tasks mit Priorität 8 und 7 für die mittlere und die untere Raupe aber werden nie ausgeführt, weil der Task mit Priorität 9 die CPU vollständig monopolisiert. Dies liegt in der Natur des Echtzeit-Scheduling in FreeRTOS. Im Gegensatz zu Linux oder Windows erhalten Tasks mit niedrigerer Priorität keinerlei Chance auf eine Ausführung.

Versuch 2

Für das zweite Experiment ändern Sie die Konfiguration so, dass alle drei Raupen die gleiche Priorität erhalten und der Haupt-Display-Task bei Priorität 10 belassen wird. Stellen Sie alle drei Raupen-Tasks auf die gleiche Priorität 9, 8 oder 7 ein. Ich werde hier Priorität 9 verwenden:

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 9
#define WORM3_TASK_PRIORITY 9
#define MAIN_TASK_PRIORITY  10
```

Was sehen Sie, wenn Sie den Code nun neu kompilieren und wieder in den ESP32 flashen? Dies:

```
  _¯_
  _¯_
  _¯_
```

Die Raupen marschieren jetzt im gleichen Tempo (oder fast im gleichen Tempo) über den Bildschirm. Nur wenn Sie den Versuch lange genug laufen lassen, könnte ein Wurm den anderen ein wenig voraus sein.

Versuch 3

Ändern Sie in diesem Experiment die Konfiguration, um wie im letzten Versuch den drei Raupen die gleiche Priorität zu geben,

aber setzen Sie auch den Haupt-Task auf die gleiche Priorität. Ich werde all diesen Tasks die Priorität 9 zuteilen.

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 9
#define WORM3_TASK_PRIORITY 9
#define MAIN_TASK_PRIORITY  9
```

Was beobachten Sie nun nach der Neukompilierung und dem erneuten Flashen, wenn Sie den Code ausführen? Sehen Sie den Unterschied? Warum kommen die Raupen mit unterschiedlicher Geschwindigkeit voran?

Wenn ich den Code ausführe, scheint die untere Raupe die meiste CPU-Zeit abzubekommen (sie kriecht am schnellsten). Die obere Raupe bewegt sich am langsamsten. Auch hier ist die von Espressif festgestellte Einschränkung der Round-Robin-Fairness daran schuld. Im Idealfall sollte der Displaytask nur ein wenig CPU-Zeit benötigen, um die Raupe zu zeichnen. In der restlichen Zeit sollten sich die drei übrigen Raupen-Tasks die verbleibende CPU-Zeit brüderlich zu gleichen Teilen teilen. Dennoch sehen wir, dass die Zeitplanung unausgewogen ist. Beide CPUs reagieren auf Timer und andere Interrupts. Der fehlerhafte Scheduler-Code ist dafür verantwortlich, dass die Fairness des Round-Robin-Verfahrens gestört wird.

Versuch 4

Bei jedem der bisherigen Versuche hat jede Raupe so viel CPU-Zeit verbraucht, wie sie bekommen konnte. Wie ändert sich das Verhalten, wenn wir eine kleine Verzögerung (zum Blockieren) innerhalb der Schleife einführen? Stellen Sie die Konfiguration so auf den Anfangszustand zurück, dass die Haupt-Display-Task die Priorität 10 und die Raupen-Tasks wieder die Prioritäten 9, 8 und 7 haben.

```
#define WORM1_TASK_PRIORITY 9
#define WORM2_TASK_PRIORITY 8
#define WORM3_TASK_PRIORITY 7
#define MAIN_TASK_PRIORITY  10
```

Dann heben Sie die Auskommentierung der Zeile auf, in der `vTaskDelay()` aufgerufen wird, so dass die Task-Schleife wie folgt aussieht:

```
void worm_task(void *arg) {
  InchWorm *worm = (InchWorm*)arg;
  for (;;) {
    ...
    for ( int x=0; x<800000; ++x )
      __asm__ __volatile__(„nop“);
    xQueueSendToBack(qh,&worm,0);
    vTaskDelay(10); // Uncommented
  }
}
```

Jetzt verbraucht jeden Raupe-Task CPU-Zeit, versucht, eine Raupe in die Warteschlange zu stellen und blockiert dann 10 Millisekunden lang. Kompilieren Sie dieses Beispiel, flashen Sie es und führen Sie es aus. Was beobachten Sie?
Die obere Raupe bewegt sich am schnellsten und die untere am langsamsten. Die obere Raupe bekommt aufgrund ihrer hohen Priorität 9 (während der Display-Task blockiert ist) den ersten Zuschlag in der CPU. Wenn der Raupen-Task aber durch den `vTaskDelay(10)`-Call blockiert ist, verbraucht der Task mit der nächstniedrigeren Priorität (die mittlere Raupe) etwas CPU-Zeit und ruft ihrerseits `vTaskDelay(10)` auf. Dies wiederum ermöglicht es dem noch niedrigeren Task mit der Priorität 7, einige CPU-Zyklen zu erhalten. Dies hat einen Pferdeäpfel-Effekt [5], der die CPU von der höchsten zur niedrigsten Ebene aufteilt. Beachten Sie jedoch, dass die Tasks mit Priorität 8 und Priorität 7 immer dann gestoppt werden, wenn der Task mit der höheren Priorität 9 wieder bereit ist. Deshalb bewegt sich der oberste Wurm am schnellsten. Die mittlere Raupe wiederum kann manchmal den Task mit Priorität 7 stoppen, so dass sie tendenziell schneller ist als die untere Raupe.

Weitere Experimente

Was passiert, wenn Sie die `vTaskDelay()`-Zeit viel länger als 10 Millisekunden machen? Versuchen Sie, sich die Antwort erst zu überlegen und führen Sie erst dann den Versuch aus. Warum haben Sie dieses Ergebnis erhalten? Was passiert eigentlich, wenn Sie die Verzögerungszeit auf 1 Millisekunde reduzieren? Diese Erkundungen bleiben Ihnen überlassen!

Konfiguration der Priorität

Obwohl wir Interrupts im ESP32 noch nicht behandelt haben, sollten Sie sich über die Header-Datei *FreeRTOSConfig.h* informieren, die die Prioritäten für die Plattform konfiguriert und

Weblinks

[1] Praktisches ESP32-Multitasking (1), Elektor 1/2020: www.elektormagazine.de/190182-02

[2] Round-Robin-Verfahren: https://de.wikipedia.org/wiki/Round_Robin_(Informatik)

[3] Symmetric Multiprocessing: https://thc420.xyz/esp-idf/file/docs/en/api-guides/freertos-smp.rst.html

[4] Projekt Source-Code: https://github.com/ve3wwg/esp32_freertos/blob/master/priority-worms1/priority-worms1.ino

[5] Pferdeäpfel-Theorie: https://de.wikipedia.org/wiki/Trickle-down-Theorie

hier zu finden ist:

```
$IDF_PATH/components/freertos/include/freertos/
FreeRTOSConfig.h
```

Der Header definiert die folgenden Prioritäten-Makro-Werte. Die kompilierten Werte sind.

```
configMAX_PRIORITIES = 25
configKERNEL_INTERRUPT_PRIORITY = 1
configMAX_SYSCALL_INTERRUPT_PRIORITY = 3
```

Das erste Makro definiert die maximale Anzahl der verfügbaren Prioritäten. Das bedeutet, dass die gültigen Prioritätsnummern von 0 bis 24 reichen.
Das zweite Makro definiert die vom Kernel selbst verwendete Priorität für Interrupts. Damit verbunden ist das dritte Makro, das die höchste Priorität festlegt, die vom Kernel für Interrupts verwendet wird. Jeder FreeRTOS-API-Aufruf, der **innerhalb** einer Interrupt-Service-Routine (ISR) erfolgt, darf nur FreeRTOS-API-Funktionen mit Namen aufrufen, die auf `FromISR()` enden. Außerdem können diese Funktionen mit den gezeigten Werten nur von den Interrupt-Task-Prioritäten 1 bis einschließlich 3 aufgerufen werden. Wenn keine `FromISR()`-Calls getätigt werden, kann die ISR frei mit den Prioritäten 4 bis einschließlich 24 arbeiten.

Zusammenfassung

Was können wir aus diesen Experimenten schließen? Sah das Konzept der Prioritäten zuerst scheinbar einfach aus, entpuppte es sich als voller Fallstricke. Die Konsequenz ist, dass es zu Überraschungen kommen kann, wenn Ihre Task-Prioritäten nicht gut geplant sind. Wir haben noch nicht über Watchdog-Timer gesprochen, aber das wirkt sich auch auf sie aus. Wenn beispielsweise der Watchdog-Timer in CPU 0 auslöst, wird Ihr ESP32 zurückgesetzt und neu gestartet.
Für den Dual-Core-ESP32 gibt es ein zusätzliches Problem, dass das Round-Robin-Verfahren auf derselben Prioritätsebene zu ungleichen Ausführungszeiten führen kann. Dies kann für einige Anwendungen problematisch sein, bei anderen spielt es keine Rolle. Das Problem hängt von der Art Ihrer Anwendung ab.

Für viele Anwendungen können Sie einfach Tasks erstellen, die mit Priorität 1 ausgeführt werden sollen. Dies ist die Priorität, die die Arduino-Tasks `setup()` und `loop()` besitzen. Tasks mit höherer Priorität können sicher verwendet werden, wenn sie durch eine Queue, einen Semaphor oder ein anderes Ereignis blockiert werden. Wenn ein Task blockiert oder angehalten wird, wird die CPU von anderen Tasks gleicher oder niedrigerer Priorität verwendet. Eine Anwendung mit ordnungsgemäß konfigurierten Taskprioritäten arbeitet wie eine gut geölte Maschine. ⏮

191195-03