64-211 Übung Eingebettete Systeme

optionale Zusatzaufgabe Z 3

Gruppe	
Name(n)	Matrikelnummer(n)

Z 3 Echtzeit-Betriebssystem FreeRTOS

Alle bisher in der Übung behandelten Anwendungsbeispiele basieren auf dem klassischen von-Neumann Paradigma mit einem einzelnen aktiven Prozess. Das gesamte Programm wird als lineare Abfolge von Funktionen organisiert, die nacheinander vom Hauptprogramm aus aufgerufen werden — also ausgehend von `main()` bzw. in der Arduino Programmierumgebung ausgehend von `setup()` und `loop()`. Um trotzdem auf externe Ereignisse reagieren zu können, haben wir zusätzlich Interrupts kennengelernt und verwendet, die temporär die Ausführung des Hauptprogramms unterbrechen.

Für komplexere Anwendungen ist es natürlich wünschenswert, das Programm in mehrere Prozesse aufzuteilen, die jeweils eine Teilaufgabe übernehmen und dann (quasi-) parallel auf einem oder mehreren CPU-Kernen ausgeführt werden und miteinander kommunizieren. Gerade die Verwaltung und Synchronisation von mehreren parallel auszuführenden Aktionen mit jeweils eigenen Deadlines wird stark vereinfacht. Und für aktuelle Mikrocontroller wie dem ESP-32 oder Raspberry RP-2040, die bereits zwei CPU-Kerne enthalten, sind mehrere parallele Prozesse und Prozesskommunikation schon deswegen notwendig, um die verfügbare Hardware überhaupt effizient nutzen zu können.

In dieser Übungsaufgabe betrachten wir **FreeRTOS** als konkretes Beispiel für die in der Vorlesung behandelten modularen Echtzeit-Betriebssysteme. Der folgende Abschnitt gibt zunächst eine Übersicht über das System. Anschließend schauen wir uns die die wichtigsten Konzepte und Funktionen für das Task-Konzept, die Speicherverwaltung, und die Verwendung von Semaphoren/Mutexen anhand von vorbereitenen Beispielprogrammen für den Arduino Due an.

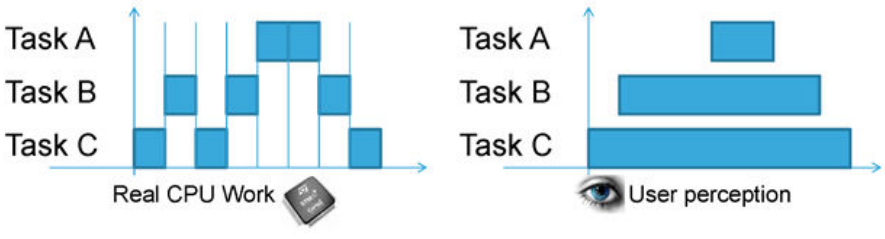


Abbildung 1: Grundidee des Multi-Tasking: mehrere Tasks werden nacheinander auf die CPU geladen und jeweils für kurze Zeitintervalle ausgeführt. Der Anwender sieht dies als parallele Ausführung der Tasks. (Bild von codeinsideout.com/blog/freertos/overview/).

Z 3.1 FreeRTOS Grundlagen

Das seit 2003 von Richard Barry und weiteren Autoren in C entwickelte FreeRTOS System umfasst einen Echtzeit-Kernel und diverse Hilfs-Bibliotheken für Synchronisation und Task-Kommunikation [1, 2]. Durch die Kombination aus großzügiger Open-Source Lizenz und geringer Hardwareanforderungen hat sich FreeRTOS als das beliebteste Echtzeit-Betriebssystem für eingebettete Systeme etabliert und ist für fast alle Mikrocontroller verfügbar [3]. Aktuell wird das Projekt bei Amazon AWS gehostet, wo auch Support und eine Reihe von weiteren Bibliotheken für Netzwerk- und Cloud-Zugriff erhältlich sind [4].

Wie in der Vorlesung motiviert, ist FreeRTOS komplett modular ausgelegt. Durch die Implementierung als Header-only Bibliothek werden nur die vom Anwender benötigten und in der zentralen Konfigurationsdatei aktivierten Funktionen compiliert und dem fertigen Binary hinzugefügt. Eine Minimalversion für 8-bit Mikrocontroller benötigt nur wenige kByte an Speicher. Der präemtive Scheduling-Algorithmus arbeitet mit vom Anwender konfigurierten festen Zeitscheiben („ticks“) und wählt jeweils den höchstpriorisierten rechenwilligen Task zur Ausführung aus. Viele Funktionen sind in Varianten verfügbar, die mit statisch alloziertem Speicher arbeiten und daher zur Laufzeit ohne dynamische Speicheranforderung auskommen, so dass im Prinzip auch das Einhalten von harten Deadlines garantiert werden kann.

FreeRTOS wird aktiv weiterentwickelt; so unterstützt die seit Dezember 2023 verfügbare Version 11.0.0 erstmals Symmetric Multiprocessing für Mikrocontroller mit mehreren CPU-Kernen. Zum Beispiel kann der FreeRTOS-Scheduler auf dem Raspberry Pico RP2040 jetzt Tasks auf beiden CPU-Kernen erzeugen, verwalten, und synchronisieren.

In dieser Übung lernen wir die Konzepte und wichtige Funktionen des Systems kennen:

- das FreeRTOS Multitasking-Konzept,
- Erzeugung und Konfiguration von Tasks,
- Pausieren und Fortsetzen von Tasks,
- Semaphore und (rekursive) Mutexe,
- Deadline-basiertes Scheduling.

Kern des Systems ist der Scheduler. Einmal gestartet, läuft dieser Prozess bis zum Ausschalten des Systems durch und verwaltet die verschiedenen Tasks. Dabei werden vier mögliche Task-Zustände unterschieden:

- **Running**
  - Task is actually executing
- **Ready**
  - Task is ready to execute but a task of equal or higher priority is Running.
- **Blocked**
  - Task is waiting for some event.
    - **Time**: if a task calls vTaskDelay() it will block until the delay period has expired.
    - **Resource**: Tasks can also block waiting for queue and semaphore events.
- **Suspended**
  - Much like blocked, but not waiting for anything.
  - Tasks will only enter or exit the suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively.

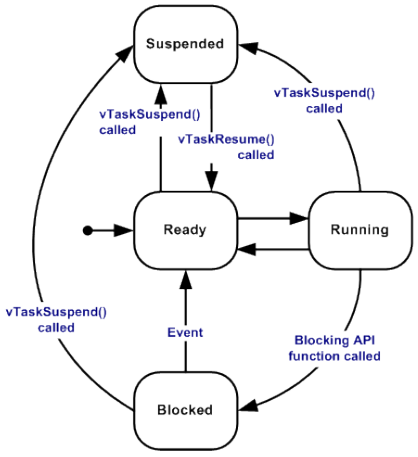


Abbildung 2: Die vier Task-Zustände im FreeRTOS-Scheduler.

Der Default-Scheduler von FreeRTOS ist prioritäts-basiert, es wird also unter allen Tasks im Zustand `ready` oder `running` immer der Task mit der höchsten Priorität ausgesucht. Wenn mehrere Tasks mit höchster Priorität im Zustand `ready` warten, werden diese abwechselnd (round-robin Verfahren) für jeweils eine Zeitscheibe ausgeführt. Dieses Verfahren ist für viele Anwendungsfälle ausreichend; bei Bedarf kann der Scheduling-Algorithmus aber auch modifiziert oder ausgetauscht werden.

FreeRTOS benötigt einen Hardware-Timer für sich, um präemptives Multitasking zu ermöglichen. Der Timer löst periodisch einen Interrupt aus, womit der laufende Task unterbrochen und statt dessen der Scheduler ausgeführt wird. Das Timer-Intervall und damit auch die Dauer einer Zeitscheibe beim Scheduling kann frei konfiguriert werden. Auf aktuellen Mikrocontrollern wird oft 1 msec als Zeitintervall gewählt, als Kompromiss aus feiner Zeitauflösung einerseits und dem Overhead durch zu häufigen Taskwechsel anderseits.

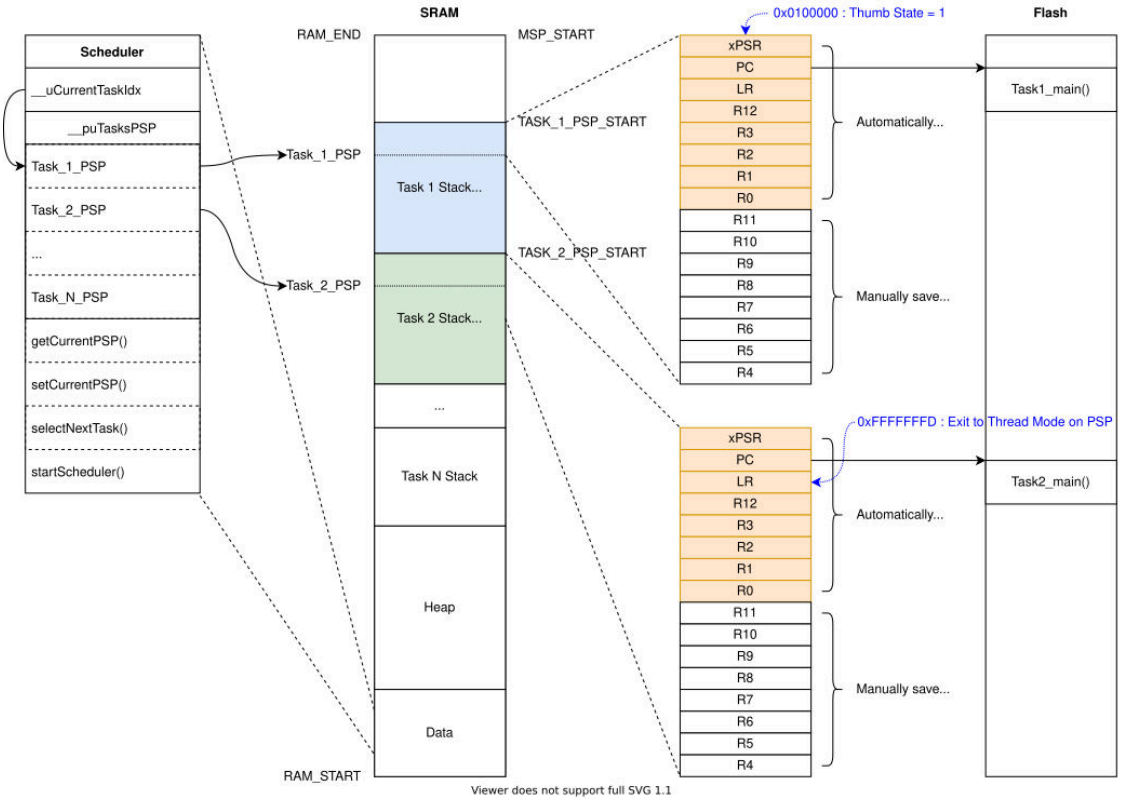


Abbildung 3: Arduino-Due Memory-Layout mit FreeRTOS. Das Hauptprogramm (Scheduler) verwaltet die Adressen für statische globale Variablen, dynamisch allozierte Objekte (Heap) und separate Speicherblöcke für die Stacks der einzelnen Tasks. Der aktive Task benutzt jeweils die Prozessorregister, beim ARM Cortex-M3 Prozessor sind dies die 16 Universalregister inklusive PC (program counter) sowie die PSR (status register) und (je nach Prozessormodell) die Floating-Point Register. Beim Kontextwechsel nach jedem tick-Interrupt werden die Register des bisher laufenden Tasks auf den privaten Stack des Tasks abgespeichert, und dann die Prozessorregister vom privaten Stack des neuen Tasks geladen. Der nächste Befehl des Tasks wird dann von der passenden Stelle aus dem Programmspeicher (Flash-Memory) geladen und dann ausgeführt. (Bild von www.codeinsideout.com/blog/freertos/overview/).

Z 3.2 FreeRTOS Installation als Arduino-Bibliothek

Für das Arduino Due Board gibt es aktuell zwei vorkonfigurierte Versionen von FreeRTOS, die sich direkt als Bibliothek in die Arduino-IDE einbinden lassen, nämlich FreeRTOS_ARM (basierend auf FreeRTOS 8.2.3) und DueFreeRTOS (basierend auf 10.1.1). Je nach verwendeter Version der Arduino-IDE können diese Bibliotheken eventuell direkt über den Bibliotheksmanager ausgewählt und installiert werden. Ansonsten reicht es aus, die DueFreeRTOS-Library von der Website der Veranstaltung herunterzuladen und im Arduino `libraries` Verzeichnis zu entpacken oder die github-Dateien direkt in das Arduino `libraries` Verzeichnis zu clonen:

```
cd ~/Arduino/libraries                                  # on Arduino+Linux, libs go here
git clone https://github.com/TAMS-Group/DueFreeRTOS     # install DueFreeRTOS
gedit DueFreeRTOS/src/FreeRTOSConfig.h                  # check FreeRTOS configuration
```

Da FreeRTOS als „header-only“ Quellcode realisiert ist, können die verschiedenen Werte in der Konfigurationsdatei `FreeRTOSConfig.h` jederzeit nach Wunsch abgeändert werden, z.B. die Anzahl der verfügbaren Task-Prioritäten. Bei einigen Optionen sind allerdings weitere Arbeiten erforderlich, etwa die passende Konfiguration eines weiteren Hardware-Timers, um die Profiling-Funktionen zu aktivieren.

Hinweis Der hier verwendete Fork von `github.com/bdmihai/DueFreeRTOS` stellt die zentrale Header-Datei unter dem Namen `DueFreeRTOS.h` bereit, um Konflikte mit weiteren eventuell parallel installierten Versionen von FreeRTOS für andere CPUs/Boards zu vermeiden.

Aufgabe Z 3.1

Installieren Sie DueFreeRTOS in ihre Arduino-IDE. Ermitteln Sie dann die vorkonfigurierten Werte für folgende Variablen und notieren Sie diese hier:

- `configCPU_CLOCK_HZ`
- `configTICK_RATE_HZ`
- `configMAX_PRIORITIES`
- `configMINIMAL_STACK_SIZE`
- `configUSE_MUTEXES`

Zur Lösung können Sie entweder den Quellcode studieren oder ein kleines Testprogramm schreiben, dass diese Werte zur Laufzeit mittels `Serial.print()` in die Console schreibt.

Aufgabe Z 3.2

Schätzen Sie ab, wie lange ein Kontextwechsel auf dem Arduino-Due Board alleine durch das Retten und Restaurieren der Prozessorregister auf die beiden Stacks (bisheriger und neuer Task) dauert. Wir nehmen dabei an, dass alle Speicheroperationen im SRAM in jeweils einem Prozessortakt ausgeführt werden.

Z 3.3 Ein erstes FreeRTOS Beispiel

Für einen ersten Test von FreeRTOS verwenden wir als externe Komonente lediglich eine LED (mit Vorwiderstand) an Pin D13. Öffnen Sie zunächst das Beispielprogramm es_10_1_due_rtos_read_analog.ino und compilieren und laden Sie es auf das Arduino-Due Board.

Unser Programm verwendet drei parallele Tasks:

- Task 1: Langsames Pulsieren der LED an Pin 13.
- Task 2: Serielle Kommunikation mit dem Host.
- Task 3: Auslesen und Filtern eines Analogwerts von Pin A4.

Wie in der Arduino-Umgebung üblich, wird gleich nach Programmstart zunächst die `setup()`-Methode aufgerufen. Wie üblich werden hier die benötigten globale Variablen und Datenstrukturen initialisiert. Außerdem werden hier jetzt die benötigten Tasks mittels `xTaskCreate()` erzeugt und initialisiert, aber noch nicht gestartet. Dies passiert erst durch Aufruf der `vTaskStartScheduler()` Funktion, die den Scheduler-Task und damit die Ausführung von FreeRTOS startet:

```
#include <DueFreeRTOS.h>                    // main FreeRTOS header file
#include <task.h>                           // "task" module definitions
...
void task1_heartbeat(   void* arg ) { ... }  // task 1 animates a LED
void task2_serial_comm( void* arg ) { ... }  // task 2 for Serial.print()
void task3_read_analog( void* arg ) { ... }  // task 3 reads and filters A0
...

void setup() {
  xTaskCreate( task1_heartbeat, ... );    // create and prepare task 1
  xTaskCreate( task2_serial_comm, ... );  // create and prepare task 2
  xTaskCreate( task3_read_analog, ... );  // create and prepare task 3
  ...                                     // more tasks as needed
  vTaskStartScheduler();                  // start everything
  error_blink();                          // signal that scheduler has crashed
}
```

Der mit `vTaskStartScheduler()` gestartete Scheduler läuft in einer Endlosschleife; die Kontrolle wird also nur dann an `setup()` zurückgegeben, wenn der Scheduler wegen eines Laufzeitfehlers (Nullpointer-Zugriffe, Stack-Overflow oder Out-of-Memory) abgebrochen wurde. Wie hier angedeutet, werden solche Crashes häufig durch ein Blinken signalisiert. Komplexere Funktionsaufrufe sind in dieser Situation kaum möglich, da eben oft kein Speicher mehr zur Verfügung steht.

Tatsächlich enthält `DueFreeRTOS`-Version in der Datei `hooks.c` eine Funktion `blink(int n)`, die sowohl für Software-Fehler als auch bei bestimmten Hardwareproblemen aufgerufen wird. Die Funktion lässt eingebaute LED (Pin D13) bis zum Ausschalten langsam n-mal blinken, wobei die Anzahl der Blinkimpulse den Fehler kodiert. Zum Beispiel zeigt einmaliges Blinken einen fehlgeschlagenen `assert()`-Test an, zweimaliges einen Speichermangel im Heap nach `malloc()`, dreimaliges einen Stackoverflow. Vierfaches Blinken signalisiert einen `HardFault`, ausgelöst etwa durch nicht oder falsch initialisierte Peripherie des Prozessors, aber auch schlicht durch eine Nullpointer-Deferenzierung.

Der erste Task in unserem Beispiel steuert die eingebaute LED (pin D13) des Arduino Due an. Wir schalten die LED aber nicht einfach nur an und aus, sondern nutzen die `analogWrite()` Funktion, um die Helligkeit der LED langsam ansteigen und wieder abfallen zu lassen:

```
void task1_heartbeat( void* arg ) {
  int phase = 0;                            // initialize task variables
  int led   = LED_BUILTIN;                  // pin D13 on most Arduino boards

  while (1) {                               // endless loop
    phase++;                                // increment phase counter
    int v = phase & 0x3f;                   // mask out lower 6 bits, range [0..63]
    if (v < 0x20) analogWrite( led, v );    // increase LED level [ 0..31] -> [0..31]
    else          analogWrite( led, 0x3f-v ); // decrease LED level [32..63] -> [31..0]
    vTaskDelay( 33 );                       // pause task execution for 33 ticks
  }
}
```

Die hier verwendete Struktur ist typisch für FreeRTOS Task „worker"-Funktionen:

- die Funktion bekommt genau ein `void*` Argument, also einen generischen Pointer. Damit kann bei Bedarf eine beliebige C-Datenstruktur via `xTaskCreate` als Argument an einen Task übergeben werden. Häufig wird dieses Argument aber gar nicht benutzt (so wie hier), und beim Aufruf der Funktion wird einfach `NULL` als Argument übergeben;
- alle benötigten lokalen Variablen werden zu Beginn der Funktion deklariert und initialisiert;
- anschließend erfolgt die eigentliche vom Task auszuführende Berechnung. Wenn der Task nicht nur einmal sondern endlos laufen soll, ist eine eigene Endlosschleife mit `while(1)` oder `for(;;)` nötig.
- Alle Zeitangaben im Code beziehen sich auf die Dauer einer Zeitscheibe („tick") des Schedulers. Der gängige Wert hierfür ist 1 msec und die innerhalb von FreeRTOS modellierbaren Zeitangaben sind Vielfache dieser Zeitscheibe.
- durch Aufruf von `vTaskDelay()` oder `vTaskDelayUntil()` kann sich ein Task für die angegebene Zeitdauer schlafen legen. Im Beispiel wartet der `heartbeat`-Task also 33 msec nach jeder Iteration der `while`-Schleife.

Unser zweiter Task übernimmt die Kommunikation über die serielle Schnittstelle am „programming port" des Arduino Due Boards:

```
void task2_serial_comm( void* arg ) {
  Serial.begin( 115200 );             // initialize Serial interface, 115.2kbaud
  while (!Serial) {                   // wait until host/console window is ready
    vTaskDelay( 50 );                 // pause a bit, then check again
  }                                   // Serial port is now ready
  Serial.println( "Welcome to DueFreeRTOS demo 10_1..." ); // greeting

  while (1) {                         // endless loop for this task
    Serial.println( v_average );      // print latest filtered value
    vTaskDelay( 500 );                // 500 msecs pause
  }
}
```

Der dritte Task dient als Prototyp für das periodische Auslesen und Verarbeiten von Sensordaten. Im Beispiel nutzen wir `analogRead()`, um die aktuell an Pin A4 anliegende Spannung zu messen. Um das Rauschen und „Outlier" zu reduzieren, lesen wir den Pin mehrfach mit einer kleinen Pause dazwischen aus und berechnen dann den Mittelwert. Das Resultat wird in die zugehörige globale Variable geschrieben:

```
...
float v_average = 0;                   // global var for filtered analog value

/** reads analog pin A4 eight times and averages the value */
void task3_read_analog( void* arg ) {
  while(1) {                           // endless loop
    float v = 0;                       // initialize local variable
    for( int i=0; i < 8; i++ ) {
      v += analogRead( A4 );           // read analog input value
      delayMicroseconds( 5 );          // short pause (busy waiting)
    }
    v_average = (v/8);                 // update global var with average
    vTaskDelay( 10 );                  // 10 ticks = 10 msecs
  }
}
```

Im obigigen Codebeispiel für `setup()` hatten wir die Parameter der Funktion `xTaskCreate(...)` zunächst nur angedeutet. Tatsächlich erwartet diese Funktion sechs Parameter, nämlich den Pointer auf die auszuführende „worker"-Funktion, den Namen des Tasks als C-String, die Größe des Runtime-Stacks für den Task, den Parameter für die

worker-Funktion (häufig NULL), die Task-Priorität als unsigned Integer, und einen Pointer (oder NULL). Dieser Pointer zeigt auf eine intern von FreeRTOS für den Task angelegte Datenstruktur, den `tskTaskControlBlock`, die alle für das Task-Scheduling nötigen Informationen enthält. Der Rückgabewert ist entweder `pdPASS` (wenn erfolgreich) oder `errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY`.

```
void setup() {
  ...
  BaseType_t status; TaskHandle_t* task1_handle;
  status = xTaskCreate( task1_heartbeat,     // pointer to "worker" function
                  "Task_1",                  // short human-readable task name
                  configMINIMAL_STACK_SIZE, // increase as needed
                  NULL,                      // argument to task_heartbeat() worker
                  3,                         // task priority 1 .. configMAX_PRIORITIES-1
                  task1_handle );            // pointer to TaskControlBlock
  ...
```

Hinweis Die von Arduino gewohnte Funktion `loop()` wird im obigen Beispiel eigentlich gar nicht benötigt, da alle Berechungen innerhalb der vom Benutzer gestarteten Tasks ablaufen. Die `DueFreeRTOS`-Implementierung für den Arduino Due erfordert allerdings, dass trotzdem eine Funktion `loop()` vorhanden ist. Diese wird dann „hinter den Kulissen“ in einem zusätzlichen FreeRTOS-Task ausgeführt:

```
void loop() {
  ; // empty. As usual for Arduino, executed in an outer endless loop.
}
```

Aufgabe Z 3.3

Laden Sie die Template/Beispielprogramme von der Webseite herunter. Öffnen Sie dann das Beispielprogramm es_10_1_due_rtos_read_analog.ino in der Arduino-IDE und laden Sie es auf das Board hoch. Schließen Sie zusätzlich ein Potentiometer an Pin A4 an öffnen Sie die Arduino-Console (115200 baud). Die on-board LED sollte jetzt pulsieren und die gefilterten Analogwerte werden in die Console ausgegeben.

Aufgabe Z 3.4

Mit FreeRTOS lassen sich die drei Tasks jeder für sich elegant beschreiben, und die parallele Ausführung wird von FreeRTOS übernommen. Die alternative herkömmliche Realisierung als single-Thread Arduino-Programm wäre in diesem Fall natürlich auch denkbar, siehe es_10_1_due_read_analog.ino:

```
void loop() {
  read_analog();      // read analog value
  serial_comm();      // write value to console
  heartbeat();        // heartbeat blinking
}
```

Allerdings wäre es jetzt sehr viel schwieriger, die Zeitbedingungen zu modellieren und einzuhalten. Überlegen Sie sich, wie eine single-Thread Realisierung nur mit Aufrufen von `delay()` aussehen müsste. Alternativ könnte man auch einen oder mehrere Timer konfigurieren. Welche Nachteile hätte diese Lösung?

Z 3.4 Hardwareaufbau für die Demos

Um möglichst wenig Zeit mit dem Hardwareaufbau zu verschwenden, haben wir für die folgenden Aufgaben eine kleine Platine mit fünf LEDs und Tastern vorbereitet, um die Ausführung der parallelen Tasks zu visualisieren und einzelne Tasks bzw. Demos zu starten (Abbildung 4). Wir verwenden dazu den „Erweiterungsstecker“ an der kurzen Seite des Arduino Due Boards (Pins D22..D53). Alternativ können Sie auch je fünf LEDs und Taster auf einer Steckplatine vorbereiten und diese an Pins 52,46,40,34,28 (LEDs) bzw. 53,47,41,35,29 (Taster) sowie GND anschließen. Zusätzlich schließen wir eine LED an Pins D13 und GND an (also parallel zur on-board LED), die zum Abspielen eines pulsierenden Herzschlags genutzt wird.

Abbildung 4: Arduino-Due mit Zusatzplatine (fünf LEDs und Buttons).

Aufgabe Z 3.5

Für einen ersten Test der Hardware laden Sie bitte das Beispielprogramm es_10_2_due_5leds.ino von der Webseite der Übung herunter und compilieren Sie es. Dies ist ein herkömmliches single-task Arduino-Programm, das in `setup()` die IO-Pins für die LEDs und Taster konfiguriert. Innerhalb von `loop()` werden dann die LEDs dann mit einem Binärcode animiert und Tastendrücke werden mittels `Serial.print()` an die Arduino-Console geschickt.

Aufgabe Z 3.6

Für den Funktionstest der LEDs laden Sie bitte den Sketch es_10_2_due_rtos_debounce.ino auf das Board hoch. Das Programm verwendet wiederum drei Tasks, wobei Task 1 wiederum für die langsame Heartbeat-Animation der LED an Pin 13 sorgt. Der zweite Task steuert die `user_LEDs` an, und zwar im Binärcode mit je einer LED für die unteren fünf Bits der Zählvariable `iteration`:

```
/** display 5-bit binary counting pattern on user_LEDS 0..4. */
void led_count( void* arg ) {
  int iteration = 0;

  while (1) {
    iteration ++;
    digitalWrite( user_LEDs[0], (iteration&0x0001) != 0 );
    digitalWrite( user_LEDs[1], (iteration&0x0002) != 0 );
    digitalWrite( user_LEDs[2], (iteration&0x0004) != 0 );
    digitalWrite( user_LEDs[3], (iteration&0x0008) != 0 );
    digitalWrite( user_LEDs[4], (iteration&0x0010) != 0 );

    vTaskDelay( 100 * MSECS ); // delay( 100 );
  }
}
```

Der dritte Task übernimmt das Auslesen und Entprellen der Taster. Hier könnten wir wie in Aufgabe 3.3 einen Hardware-Timer konfigurieren und die Tastendrücke in einer Interrupt-Routine behandeln. Da ein Auslesen der

Taster mit ca. 50 Hz jedoch schnell genug ist, verwenden wir statt des separaten Timers einfach den FreeRTOS-Scheduler:

```
#define N_BUTTONS (5)
int    button_pins[ N_BUTTONS ] = { 53, 47, 41, 35, 29 };
int button_pressed[ N_BUTTONS ];  // global stable key-press state for each button

void read_debounce_buttons( void* arg ) {
  int button_sample_1[ N_BUTTONS ];      // local variables for first and second read
  int button_sample_2[ N_BUTTONS ];
  while (1) {
    for( int i=0; i < N_BUTTONS; i++ ) {
      button_sample_1[i] = digitalRead( button_pins[i] ); // read all buttons once
    }
    vTaskDelay( pdMS_TO_TICKS( 3 ));                     // sleep 3msec for deboucing
    for( int i=0; i < N_BUTTONS; i++ ) {
      button_sample_2[i] = digitalRead( button_pins[i] ); // read all buttons again
    }

    for( int i=0; i < N_BUTTONS; i++ ) {                 // check if stable, do actions
      if (button_sample_1[i] == button_sample_2[i]) {    // button is stable now
        if (button_sample_2[i] != button_pressed[i]) {   // but button state changed
          button_pressed[i] = button_sample_2[i];        // save current state as global
          if (button_pressed[i] == 0) { /* do button-pressed action */ }
          else                        { /* do button-released action */ }
        }
      }
    }
    vTaskDelay( pdMS_TO_TICKS( 17 ));        // sleep 17ms (total 3+17 ms or 50Hz)
  }
}
```

In jeder Iteration der äußeren Schleife werden zunächst die aktuellen Inputwerte aller Taster (0=gedrückt, 1=offen) eingelesen und im lokalen Array `button_sample_1` abgespeichert. Anschließend legt sich der Task für die gewünschte Entprellzeit (hier 3 msec) schlafen und liest dann alle Taster erneut ein. Das hier verwendete Makro pdMS_TO_TICKS() erwartet eine Zeitangabe in Millisekunden und konvertiert den Wert in die benötigte Anzahl von Ticks.

In der dritten Schleife wird dann für jeden Taster überprüft, ob die beiden Messwerte übereinstimmen. Wenn ja, war der Wert für die Entprellzeit stabil und wird mit dem bisherigen globalen Wert `button_pressed` verglichen. Sollte sich dieser Wert geändert haben, werden dann die notwendigen Aktionen („Knopf gedrückt / losgelassen“) ausgeführt, und schließlich wird der neue Inputwert auch in die globale Variable `button_pressed` übernommen.

Z 3.5 Experimente mit Task-Prioritäten

Das Programm es_10_3_due_rtos_priorities.ino dient für Experimente mit Task-Prioritäten. Dazu stellen wir die Programmstruktur etwas um. Passend für die Hardware mit fünf LEDs und fünf Tastern erzeugen wir fünf Worker-Tasks {A, B, C, D, E}, die jeweils durch Tastendruck (re-) aktiviert werden und die während der aktiven Ausführung ihre zugehörige LED blinken lassen. Damit ist auf einen Blick ersichtlich, welche Tasks momentan aktiv sind.

Um die fünf Tasks referenzieren zu können, definieren wir zunächst fünf `TaskHandle_t` Variablen, die später beim Aufruf von `xCreateTask()` initialisiert werden. Zusätzlich definieren wir globale Arrays, die für jeden dieser Worker-Tasks die gewünschte Laufzeit in Sekunden, die CPU-Ausnutzung in Prozent (0..100), den Task-Fortschritt in Zehntelsekunden, und ein idle/active-Flag speichern. (Hinweis: Die meisten dieser Werte könnten auch von FreeRTOS selbst geliefert werden, aber die DueFreeRTOS-Version ist dafür nicht konfiguriert.)

```
#define N_WORKERS  (5)
unsigned int   task_busy[ N_WORKERS ] = { 0, 0, 0, 0, 0 }; // status: 0=idle, 1=active
unsigned int   work_done[ N_WORKERS ] = { 0, 0, 0, 0, 0 }; // work done since start
unsigned int   last_done[ N_WORKERS ] = { 0, 0, 0, 0, 0 }; // since last statistics print
TaskHandle_t   handle_A, handle_B, handle_C, handle_D, handle_E;
```

In unserem Beispiel wollen wir mindestens drei verschiedene Task-Prioritäten ausprobieren, wobei die Tasks `A` bis `D` die CPU komplett auslasten sollen (100%), während Task `E` während seiner Laufzeit lediglich 50% des Prozessors belegen soll. Dazu werden wir eine Funktion `generic_worker()` implementieren, die mit vier Parametern (Task-Name, LED-Index, Laufzeit in Sekunden, CPU-Prozent) aufgerufen wird:

```
//   Task-Funktion                    Name, LED, Laufzeit, CPU-Prozent
void worker_A( void *pvParams ) { generic_worker( "A", 0, 10, 100 }
void worker_B( void *pvParams ) { generic_worker( "B", 1, 10, 100 }
void worker_C( void *pvParams ) { generic_worker( "C", 2, 10, 100 }
void worker_D( void *pvParams ) { generic_worker( "D", 3, 10, 100 }
void worker_E( void *pvParams ) { generic_worker( "E", 4, 10,  50 }
```

Die Priorität der Tasks wird beim Erzeugen der Tasks festgelegt:

```
void setup() {
  ...
  //              Task-Funktion, Name, Stack-Größe, pvParams, Prio., Task-Handle
  BaseType_t status;
  status = xTaskCreate( worker_A, "A", MIN_STACK_SIZE+100, NULL, 2, &handle_A );
  status = xTaskCreate( worker_B, "B", MIN_STACK_SIZE+100, NULL, 2, &handle_B );
  status = xTaskCreate( worker_C, "C", MIN_STACK_SIZE+100, NULL, 3, &handle_C );
  status = xTaskCreate( worker_D, "D", MIN_STACK_SIZE+100, NULL, 4, &handle_D );
  status = xTaskCreate( worker_E, "E", MIN_STACK_SIZE+100, NULL, 4, &handle_E );
  vTaskStartScheduler();
}
```

Einmal durch Tastendruck gestartet, sollen die einzelnen Tasks sich nach Ablauf der geforderten Laufzeit selbst schlafen legen. Dazu könnte man natürlich nach jedem Tastendruck dynamisch einen neuen Task erzeugen und diesen nach Ablauf wieder bereinigen. Es ist aber übersichtlicher und effizienter (und daher üblich), alle Tasks statisch zu definieren, sofern dies für eine Anwendung überhaupt möglich ist. Das ist hier der Fall. Tatsächlich können wir für unsere `generic_worker`-Funktion wieder das bekannte Konzept mit einer Endlosschleife verwenden. Neu ist jetzt nur, dass die Funktion am Ende einer Iteration die Funktion `vTaskSuspend()` aufruft, mit der der jeweils laufende Task sich selbst schlafen legt:

```
void generic_worker( char* wname, int index, int duration, int load )
{
  load = clamp( load, 1, 100 );            // limit value to range [1..100]

  while( 1 ) {                                      // outer endless loop
    task_busy[index] = 1;                  // task has been (re-) started
    work_done[index] = 0;                 // counts cpu-time (cycles) used

    // one blink takes 100+100 msecs, so we multipy by 5 to get seconds,
    // task LED will blink while task executes:
    for( int i=0; i < 5*duration; i++ ) {
      digitalWrite( user_LEDs[index], HIGH );             // switch LED on
      delayMicroseconds( load*1000 );           // cpu busy for load msecs
      vTaskDelay( 100-load );             // cpu free for remaining period
      work_done[index] += 1;               // we've done 100 msecs of work

      digitalWrite( user_LEDs[index], LOW );              // switch LED off
      delayMicroseconds( load*1000 );           // cpu busy for load msecs
      vTaskDelay( 100-load );             // cpu free for remaining period
      work_done[index] += 1;               // we've done 100 msecs of work
    }

    task_busy[index] = 0;              // finished "duration" secs of work
    vTaskSuspend( NULL );                           // task suspends itself
  }
}
```

Wie in der Arduino-IDE gewohnt, ist die Funktion `delayMicroseconds()` auch in DueFreeRTOS als busy-waiting Schleife implementiert und belegt die CPU für die angeforderte Zeit. Normalerweise wird diese Funktion nur verwendet, um bei bestimmten I/O-Zugriffen Wartezeiten von einigen Mikrosekunden einzuhalten. In unserem Fall dient die Funktion als Platzhalter für „echte“ nützliche Berechnungen und legt die CPU für die angegebene Wartezeit von (bis zu) 100 Millisekunden lahm. Natürlich unterliegt auch das aktiven Warten in `delayMicroseconds()` noch dem präemptiven Multi-Tasking von FreeRTOS und kann jederzeit beim nächsten Scheduler-Tick unterbrochen werden.

Dagegen gibt `vTaskDelay()` die Kontrolle direkt an den FreeRTOS-Scheduler zurück, der dann den aufrufenden Thread frühestens nach Ablauf der gewünschten Wartezeit wieder aufruft. Durch die Kombination der Zeiten aus einem Anteil von `delayMicroseconds()` und einem Anteil `vTaskDelay()` erreichen wir den Effekt, dass ein Task die CPU nur zu einem gewissen Prozentanteil auslastet.

Neben diesen Worker-Tasks brauchen wir wiederum einen separaten Task zum gemeinsamen Auslesen und Entprellen aller Tasten. Da die auszuführende Aktion für alle Tasten gleich ist, kommen wir mit einer Hilfsfunktion `check_restart_worker_task()` aus, die den Index der gedrückten Taste als Parameter bekommt und dann `vTaskResume()` mit dem passenden (globalen) Task-Handle aufruft. Dadurch wird dieser Task dann von FreeRTOS wieder als aktiv markiert und wird vom Task-Scheduler dann wieder berücksichtigt:

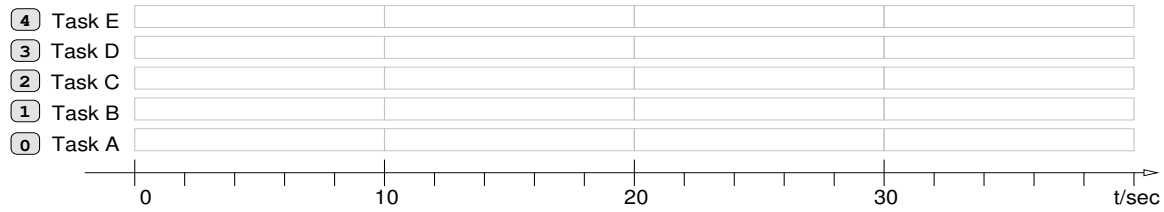
```
void check_restart_worker_task( int worker_id ) {
  if (worker_id == 0) vTaskResume( handle_A );
  if (worker_id == 1) vTaskResume( handle_B );
  if (worker_id == 2) vTaskResume( handle_C );
  if (worker_id == 3) vTaskResume( handle_D );
  if (worker_id == 4) vTaskResume( handle_E );
}
```

```
void task_Y_debounce_buttons( void* pvParameters )
{
  int tmp_1[N_BUTTONS], tmp_2[N_BUTTONS];

  while (1) {
    ... // sample all buttons once into tmp_1 array
    ... // sleep a bit for debouncing
    ... // sample buttons again into tmp_2 array

    for( int i=0; i < N_BUTTONS; i++ ) { // i      check all buttons
      if (tmp_1[i] == tmp_2[i]) {           i       // button stable
        if (tmp_2[i] != button_states[i]) {       // button changed
          if (tmp_2[i] == 0) {                  // button is pressed
            check_restart_worker_task( i );       // restart worker
          }
          button_states[i] = tmp_2[i]; // remember new button state
        }
      }
    } // for
    ...                                 // sleep until next iteration
  } // while
}
```

Zusätzlich starten wieder separate Tasks für das Pulsieren der LED an Pin D13 und für Debug- bzw. Logging-Ausgaben in die serielle Konsole.

Aufgabe Z 3.7

Spielen Sie ein bisschen mit dem Programm herum und probieren Sie diverse Kombinationen aus Laufzeit und Priorität der gestarteten Tasks aus. Skizzieren Sie zum Beispiel, was passiert wenn Sie die Tasten `(0)` und `(1)` gleichzeitig drücken, und ca 5 Sekunden später die Tasten `(2)` und `(4)`?



Aufgabe Z 3.8

In allen bisherigen Codebeispielen und auch in der Skizze auf Seite 3 hatte jeder Tasks seine eigene Worker-Funktion. Hier dagegen wird dieselbe Funktion `generic_worker()` offenbar von mehreren Tasks gleichzeitig ausgeführt. Überlegen Sie sich gründlich, wie es überhaupt möglich ist, dass ein und dieselbe Funktion mit verschiedenen Argumentwerten gleichzeitig von verschiedenen Tasks (hier fünf) benutzt werden kann. Begründen Sie kurz, wie dies möglich ist:

Z 3.6 Semaphore und Synchronisation

Wie in der Vorlesung erläutert, sind Mechanismen zur Verwaltung gemeinsam genutzer Resourcen und zur Koordination asynchroner Abläufe für jedes Multi-Tasking System essentiell. Im Prinzip muss jeder nicht-atomare Schreib- oder Lesezugriff auf den Speicher geschützt werden, also Schreibzugriffe auf Datentypen wie Arrays oder Structs, und natürlich viele Algorithmen, die komplexe Datenstrukturen durchlaufen oder modifizieren (z.B. Iteratoren). Auf dem Arduino Due mit 32-bit ARM Prozessor müssen aber selbst Zugriffe auf 64-bit Datentypen wie `uint64_t` oder `double` geschützt werden, da jeder Lese- oder Schreibzugriff zwei Zyklen erfordert, und ein Zugriff damit zwischendurch vom Scheduler unterbrochen werden könnte.

FreeRTOS stellt die drei grundlegenden Mechanismen zur Verfügung; die sogenannten binären Semaphore (ohne weitere Extras), Mutexe (mutual exclusion) mit Prioritätsvererbung, und rekursive Mutexe. Die Datenstrukturen und zugehörigen Funktionen `xSemaphoreTake()` und `xSemaphoreGive()` sind in einer eigenen Headerdatei definiert, die mit `#include <semphr.h>` eingebunden werden muss.

Ein Semaphor kann von verschiedenen Tasks reserviert und wieder freigegeben werden; bei einem Mutex müssen beide Operationen vom selben Task aus erfolgen. Rekursive Mutexe können von einem Task mehrfach reserviert werden und sind erst nach der gleichen Anzahl von Freigaben wieder verfügbar.

Zur Veranschaulichung bauen wir jetzt das Beispiel zur Prioritätsvererbung aus der Vorlesung nach. Dazu benötigen wir drei Tasks. Dies ist zunächst der Task A mit niedriger Priorität, der von Anfang an ($t = 0$ sec) läuft und zwei Sekunden später ($t = 2$ sec) eine Resource R belegt, sprich ein Semaphor bzw. Mutex reserviert. Dieser Task möchte anschließend sechs Sekunden lang rechnen, bevor er die Resource R wieder freigibt, dann noch zwei weitere Sekunden lang rechnet, und sich anschließend beendet.

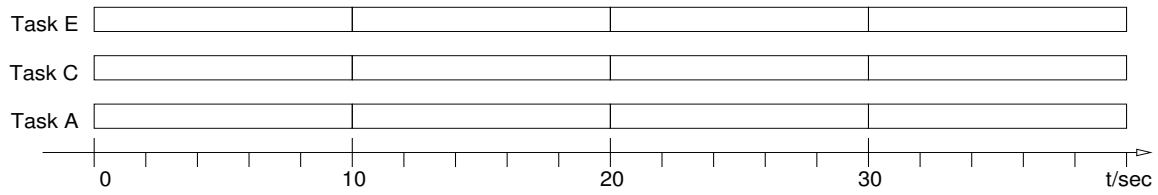
Der Task C mit mittlerer Priorität greift nicht auf die Resource R zu, sondern möchte einfach nur 10 Sekunden lang rechnen. Dieser Task wird bei $t = 4$ gestartet und verdrängt dabei gleich den Task A.

Schließlich wird bei $t = 6$ der Task E mit hoher Priorität gestartet, der dann auch sofort ausgeführt wird und wiederum Task C verdrängt. Bei $t = 8$ versucht dann Task E den Zugriff auf die Resource R, will danach noch vier Sekunden lang rechnen, dann R wieder freigeben, und sich nach zwei weiteren Sekunden beenden.

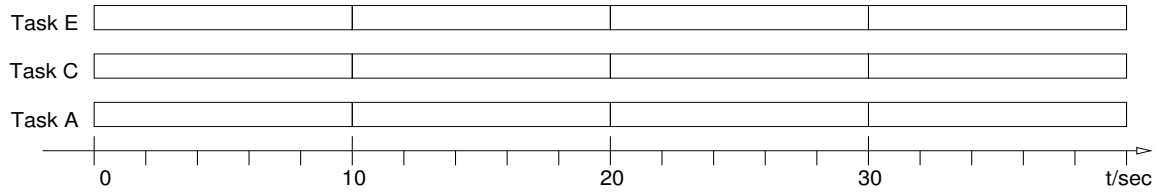
Aber der Zugriff auf R von Task-E bei $t = 8$ schlägt zunächt fehl, denn die Resource ist zu diesem Zeitpunkt noch von Task A reserviert. Das weitere Verhalten hängt jetzt davon ab, ob unser FreeRTOS-Programm ein einfaches Semaphor (ohne Prioritätsvererbung) oder einen Mutex (mit Prioritätsvererbung) verwendet.

Aufgabe Z 3.9

Überlegen Sie sich das Zeitverhalten des Programms in beiden Fällen und zeichnen Sie das erwartete Verhalten in die folgenden beiden Diagramm ein, zunächst mit Semaphor ohne Prioritätsvererbung:



Und dann mit Mutex und Prioritätsvererbung:



Laden Sie das entsprechende Arduino-Programm es_10_4_due_rtos_inversion.ino in den Editor und studieren Sie den Quellcode. Wir benötigen offenbar drei Worker-Tasks A, C, E und ein Semaphor bzw. Mutex:

```
/* es_10_4_due_rtos_inversion.ino */
#include <DueFreeRTOS.h>                  // we want FreeRTOS
#include <task.h>                         // we need the tasks module
#include <semphr.h>                       // and also the semaphore module

#define WORKER_STACK_SIZE (configMINIMAL_STACK_SIZE + 200)

TaskHandle_t handle_A;                    // three worker threads
TaskHandle_t handle_C;
TaskHandle_t handle_E;

SemaphoreHandle_t lock;                   // we need one semaphore or mutex.
...

void setup() {
  ...                                     // setup I/O pins, setup Serial, ...

  lock = xSemaphoreCreateMutex();         // mutex: with priority inheritance.
  // lock = xSemaphoreCreateBinary();     // simple binary lock, without priority inheritance.

  // create three worker tasks, one per LED / button
  xTaskCreate( worker_A, "A", WORKER_STACK_SIZE, NULL, 2, &handle_A );
  xTaskCreate( worker_C, "C", WORKER_STACK_SIZE, NULL, 3, &handle_C );
  xTaskCreate( worker_E, "E", WORKER_STACK_SIZE, NULL, 4, &handle_E );

  // system tasks: LED heartbeat, console logging, debounce buttons, command parser
  //
  xTaskCreate( task_V_heartbeat, "V", WORKER_STACK_SIZE, NULL, 5, &handle_V );
  xTaskCreate( task_X_statistics, "X", WORKER_STACK_SIZE, NULL, 5, &handle_X );
  xTaskCreate( task_Y_debounce_buttons, "Y", WORKER_STACK_SIZE, NULL, 5, &handle_Y );
  xTaskCreate( task_Z_read_serial, "Z", WORKER_STACK_SIZE, NULL, 5, &handle_Z );
  ...                                     // initialize global task data
  vTaskStartScheduler();                  // start scheduler
}
```

Der bzw. die jeweils aktiven Worker-Tasks sollen wieder durch blinkende LEDs visualisiert werden. Dabei sollen unterschiedliche Blinkmuster die verschiedenen Zustände der Worker-Tasks andeuten:

- LED leuchtet schwach, erreicht durch schnelles Blinken 1/24 msec (an/aus): der Task wurde gestartet, ist aber noch nicht aktiv;
- Blinken 100/100 msec (an/aus): Task „running“, kein Semaphor/Mutex gehalten;
- Blinken 175/25 msec (an/aus): Task „running“, Semaphor/Mutex reserviert;
- LED dauerhaft an oder aus: Task blockiert, entweder Priorität zu niedrig oder Warten auf Resource;
- alle LEDs dauerhaft aus: die Demo ist fertig durchgelaufen.

Dazu definieren wir uns eine Hilfsfunktion, die das Blinken der ausgewählten LED mit den gewünschten An/Aus-Zeiten sowie der geforderten Gesamtdauer implementiert:

```
/* blink the chosen user LED (0..4) for the given total_duration,
 * using pulses with the given on/off period. All times in msecs.
 * Waiting uses vTaskDelay and does not block the CPU.  */
void idle_blink( int index, int t_high, int t_low, int total_duration )
{
  int runtime = 0;
  while (runtime < total_duration) {
    digitalWrite( user_LEDs[index], HIGH );
    vTaskDelay( t_high );
    runtime += t_high;

    digitalWrite( user_LEDs[index], LOW );
    vTaskDelay( t_low );
    runtime += t_low;
  }
}
```

Die zweite Funktion `busy_blink()` ist sehr ähnlich, benutzt aber busy-waiting via `delayMicroseconds()` anstelle von `vTaskDelay()`. Die verbrauchte Rechenzeit in Millisekunden wird hier außerdem in der zum Task-Index i gehörigen Variablen `work_done[i]` aufsummiert.

Mit diesen Hilfsfunktionen können die Worker-Tasks jetzt kompakt aufgeschrieben werden. Zum Beispiel ergibt sich für Task-A, der nach zwei Sekunden Rechenzeit die Resource R reserviert, dann XXX Sekunden rechnet, die Resource wieder freigibt, nochmal zwei Sekunden rechnet, und sich dann selbst schlafen legt:

```
/* low-priority task for priority-inheritance demo */
void worker_A( void *pvParameters ) {
  int index = 0;          // LED 0
  while( 1 ) {
    task_busy[index] = 1; // started or re-restarted

    busy_blink( index, 100, 100, 2000 );  // 100+100 blink (on/off) for 2 seconds

    xSemaphoreTake( lock, 1 );    // reserve semaphore/mutex
    have_lock[index] = 1;
    busy_blink( index, 175, 25, 10*1000 ); // 175+25 blink for 10 seconds

    xSemaphoreGive( lock );       // unlock semaphore/mutex
    have_lock[index] = 0;
    busy_blink( index, 100, 100, 4000 );   // 100+100 blink for 4 seconds

    task_busy[index] = 0;
    vTaskSuspend( NULL ); // suspend calling thread (until next restart)
  } // while(1)
} // worker_A
```

Nachdem die Demo komplett durchgelaufen ist, kann sie durch Drücken der {linken/mittleren/rechten} Taste neu gestartet werden, und zwar {mit einfachem-Semaphor/im-bisherigen-Modus/mit-Mutex}.

Zusätzlich dient der Task Z als Parser für Kommandos über die serielle Schnittstelle. Durch Eingabe der Befehle 'a', 'c', 'e' können die Worker-Tasks A, C, E einzeln neu gestartet werden (mit unterschiedlichen Startzeitpunkten), während der Befehl 'g' („go“) die gesamte Demo neu startet. Die beiden Befehle 'r' und 's' erlauben es, den Typ des Semaphors umzuschalten, nämlich 's' für ein einfaches binäres Semaphor ohne Prioritätsvererbung und 'r' für einen Mutex mit Prioritätsvererbung.

Aufgabe Z 3.10

Laden Sie das Programm und beobachten Sie, welche Tasks in welcher Reihenfolge gestartet wurden. Öffnen Sie dann die serielle Konsole und prüfen Sie anhand der Log-Meldungen, welche Sequenz tatsächlich ausgeführt wurde. Schalten Sie dann durch Eingabe von 's' und 'r' zwischen Semaphor und Mutex um, und starten Sie dann die Demo erneut (Eingabe von 'g' oder Taste drücken). Vergleichen Sie die Sequenz mit ihren Erwartungen aus Aufgabe **??**.

```
Task V: 000002 started...
Task E: 000004 created: prio 4, [0-wait-8 8-blink-10 take-lock 10-blink-14 ...
Task E: 000012 started and idle waiting...
Task B: 000016 created: prio 3, [0-wait-6 6-blink-14 finish]
Task B: 000021 started...
Task A: 000024 created: prio 2, [0-blink-4 take-lock 4-blink-long-10 release-lock ...
Task A: 000032 started...
loop  : 004053
Task A: 002052 locked the semaphore...
Task B: 006024 executing now...
Task E: 008016 active now...
Task E: 010027 trying to get the semaphore...
Task B: 016082 finished and suspend...
Task E: 020027 got and locked the semaphore...
Task E: 026060 unlocking the semaphore...
Task E: 026060 continuing for 2 more seconds...
Task E: 028071 finished and suspend...
Task A: 030221 unlocked the semaphore...
Task A: 034243 finished and suspend...
loop  : 036245
```

Schauen Sie sich jetzt den Quellcode des Programms noch einmal gründlich an. Um eine sauber formatierte Ausgabe zu erzeugen, zum Beispiel links- oder rechtsbündig ausgerichtete Zahlenwerte, kommt in C-Programmen oft die Funktion `printf()` bzw. deren Varianten `sprintf()` und `snprintf()` zum Einsatz. Für einige Mikroprozessoren bietet die Arduino-IDE direkt die passende Funktion `Serial.printf()` an, nicht aber für das Arduino Due Board.

Wir verwenden statt dessen eine weitere kleine Hilfsfunktion `pprint()`, die intern `snprintf()` zweimal mit denselben Argumenten aufruft. Der erste Aufruf dient dazu, die Länge des formatierten Strings zu messen, bevor dann ein ausreichend grosses Array auf dem Stack angelegt wird. Beim zweiten Aufruf füllt `snprintf()` dann dieses Array, und schließlich wird dieses mit der normalen `Serial.print()` Funktion an die Konsole ausgegeben:

```
// from https://www.e-tinkers.com/2020/01/do-you-know-arduino-sprintf-and-floating-point/
//
template <typename... T>
void pprint(const char *str, T... args) {
  int len = snprintf(NULL, 0, str, args...);
  if (len) {
    char buff[len+1];
    snprintf(buff, len+1, str, args...);
    Serial.print(buff);
  }
}
```

Der dabei auf dem Stack für das Array `buff` benötigte Speicher ist dann auch der Grund für die vergrößerte Stack-Size im `xCreateTask()` Aufruf für unsere Worker (und System-) Tasks.

Vermutlich werden Sie sich an dieser Stelle fragen, ob denn die serielle Ausgabe nicht auch als gemeinsam genutze Resource zu betrachten ist? Und hätten wir dann nicht auch die verschiedenen Aufrufe von `Serial.print()` aus den verschiedenen Tasks heraus mit Semaphoren schützen müssen? Die Antwort darauf ist natürlich zweimal „ja“, aber der daraus resultierende Overhead würde das Programm um einiges unübersichtlicher machen. Tatsächlich funktioniert das Programm aber auch ohne, offenbar ist also `Serial.print()` intern so umgesetzt, dass ein einzelner Aufruf nicht zwischendurch von weiteren Aufrufen unterbrochen oder abgebrochen wird.

Das ist aber keineswegs garantiert. Wenn Sie auch die ältere Version `FreeRTOS_ARM` installieren, finden Sie dort eine Reihe von einfachen Beispielprogrammen. Diese verwenden die Holzhammermethode, um die serielle Ausgabe zu sichern:

```
// FreeRTOS_ARM/src/basic_io_arm.cpp

void vPrintString( const char *pcString ) {
  /* Print the string, suspending the scheduler as method of mutual exclusion. */
  vTaskSuspendAll();
  {
      Serial.print(pcString);
      Serial.flush();
  }
  xTaskResumeAll();
}
```

Den Scheduler schlichtweg zu beenden, ist natürlich nicht im Sinne eines echten Multi-Tasking, und statt dessen sollten Semaphore/Mutexe genutzt werden. Trotzdem kann dieser Notbehelf in manchen Situationen nützlich oder sogar notwending sein.

Hinweis Einige Versionen von FreeRTOS (u.a. ESP32) installieren standardmässig zusätzlich zum Hardware-Timer für den Scheduler noch einen Watchdog-Timer, der bei Ausbleiben des tick-Interrupts einen Alarm bzw. Neustart des Prozessors auslöst. Das Abschalten des Schedulers mit `vTaskSuspendAll()` funktioniert hier also gar nicht bzw. nur dann, wenn der Scheduler anschließend noch innerhalb des Watchdog-Intervalls auch wieder gestartet wurde.

Z 3.7 Rate-Monotonic Scheduling

In den bisherigen Beispielen hatten wir die Prioritäten der Tasks mehr oder weniger willkürlich gewählt. Im praxisrelevanten Szenario einer festen Anzahl von periodischen Tasks liefert Rate-Monotonic Scheduling (RMS) eine einfache Regel: die Tasks werden nach ihrer Periode sortiert; dabei erhalten die Tasks mit der kürzesten Periode die höchste Priorität. In jedem Zeitschritt wird der höchstpriorisierte rechenwillige Task ausgeführt. Damit lässt sich RMS direkt mit dem FreeRTOS Scheduler realisieren, es muss lediglich die ausreichende Anzahl von Prioritäten konfiguriert werden.

Zusätzlich kommen bei RMS auch erstmals Deadlines ins Spiel, denn jeder einzelne Task muss innerhalb seiner Periode mit seiner Berechnung fertig sein. Für unsere Demo konfigurieren wir fünf periodische Tasks. Die aktive Ausführung jedes Tasks wird durch Aktivieren der zugehörigen LED angezeigt, ein Überschreiten der Deadline durch schnelles Blinken für zwei Sekunden. Die Ausführung der Tasks wird zusätzlich in der Konsole protokolliert.

Die Rechenzeit jedes Tasks kann durch (kurze) Tastendrücke schrittweise erhöht oder durch langen Tastendruck auf den voreingestellten Wert zurückgesetzt werden. Damit startet die Demo zunächst mit geringer Rechenlast, bei der zunächst noch keine Deadlines verletzt werden. Durch schrittweise Erhöhung der Rechenlast der einzelnen Tasks wird das Scheduling aber zunehmend vor Probleme gestellt. Alternativ können die Perioden und Rechenlast der Tasks auch textbasiert über die serielle Konsole konfiguriert werden.

Für unsere fünf periodischen „Worker“-Tasks { A, B, C, D, E } wählen wir die Prioritäts-Level { 2, 3, 4, 5, 6 } aus, um Störungen durch die Arduino `loop()`-Funktion oder den Idle-Task (mit Priorität 1 und 0) zu vermeiden.

Da die einzelnen Tasks nicht beeinflussen können, wann und ob sie vom FreeRTOS-Scheduler gestartet werden, benötigen wir für diese Überwachung einen weiteren „Supervisor“-Task. Dieser wird mit Priorität 7 und hoher Rate (100 Hz bzw. 10 msec) ausgeführt. Er überprüft, ob die Tasks ihre Deadlines eingehalten haben, protokolliert Abläufe in der Textkonsole und startet ggf. das Blinken der LEDs als Warnsignal bei Überschreiten einer Deadline. Nebenbei übernimmt dieser Task auch das Pulsieren der Status-LED (D13).

Zusätzlich starten wir zwei weitere „System“-Tasks, nämlich wie gewöhnt einen Task für das Auslesen und Entprellen der Tasten sowie einen Task für das Parsen und Ausführen von Befehlen über die Arduino-Konsole. Auch diese Tasks laufen mit Priorität 7, so dass FreeRTOS mindestens mit `configMMAX_PRIORITIES` 8 konfiguriert werden muss.

```
void setup() {
  ...
  // five (periodic) worker tasks, one per LED/button
  //
  xTaskCreate( worker_A, "....A", WORKER_STACK_SIZE, NULL, 2, &handle_A );
  xTaskCreate( worker_B, "...B.", WORKER_STACK_SIZE, NULL, 3, &handle_B );
  xTaskCreate( worker_C, "..C..", WORKER_STACK_SIZE, NULL, 4, &handle_C );
  xTaskCreate( worker_D, ".D...", WORKER_STACK_SIZE, NULL, 5, &handle_D );
  xTaskCreate( worker_E, "E....", WORKER_STACK_SIZE, NULL, 5, &handle_E );

  xTaskCreate( task_V_supervisor, "V", WORKER_STACK_SIZE, NULL, 7, &handle_V );
  xTaskCreate( task_Y_debounce_buttons, "Y", WORKER_STACK_SIZE, NULL, 7, &handle_Y );
  xTaskCreate( task_Z_read_serial, "Z", WORKER_STACK_SIZE, NULL, 7, &handle_Z );

  Serial.print( "Rate-Monotonic Scheduling demo: staring scheduler now...\n\n" );
  vTaskStartScheduler();
}
```

Für die Konfiguration der Worker-Tasks verwenden wir wieder globale Variaben, nämlich Arrays mit Werten für die einzelnen Worker (A=0, ..., E=4). Dabei definieren `task_period` und `task_deadline` die Periode und die nächste Deadline eines Worker-Tasks, während `work_todo` die vom Task während einer Periode zu leistende Arbeit bestimmt. Alle Werte werden in Ticks bzw. Millisekunden gemessen. Die Anfangswerte werden zusätzlich in den `default_...` Arrays gespeichert, damit diese Werte später restauriert werden können:

```
#define N_TASKS (5)
char task_name[ N_TASKS ][20] = { "....A", "...B.", "..C..", ".D...", "E...." };

unsigned int task_prio[ N_TASKS ]       = { 2, 3, 4, 5, 6 };
unsigned int deadline_missed[ N_TASKS ] = { 0, 0, 0, 0, 0 };
unsigned int task_period[ N_TASKS ];       // in milliseconds/ticks
unsigned int task_deadline[ N_TASKS ];    // milliseconds/ticks of next period = deadline
unsigned int task_led_state[ N_TASKS ];   // blinking state from task, or on from supervisor

unsigned int work_todo[ N_TASKS ]; // count that should be reached by each task [0..goal]
unsigned int work_done[ N_TASKS ]; // count reached by the task so far

unsigned int default_period[ N_TASKS ]    = { 5000, 4100, 2700, 1750, 1000 }; // A..E
unsigned int default_work_todo[ N_TASKS ] = { 1000,  500,  400,  200,  200 }; // A..E
...
```

Die Worker-Funktionen der fünf periodischen Tasks können jetzt wieder mit einer gemeinsamen Funktion `periodic_task()` implementiert werden, die mit den passenden Werten für Task-Namen, Index (LED, Button), Priorität, Periode, und zu leistende Arbeit aufgerufen wird. Schauen Sie sich bitte den Quellcode diesr Funktion gründlich an. Im Detail ist der Code durch Abfragen und Debug-Ausgaben etwas unübersichtlich, das Grundkonzept ist aber ganz einfach:

```
void periodic_task( char* name, int index, int priority, uint32_t period, uint32_t work )
{
  ...                                   // print "task creation message" to console
  task_period[index] = period;          // copy arguments into global arrays
  work_todo[index]   = work;
  ...
  task_deadline[index] = period;        // first deadline is at end of first period
  deadline_missed[index] = 0;           // so far, we didn't miss our deadline

  while( 1 ) {                          // endless loop, one iteration per period
    work_done[index] = 0;               // measure work done (cpu time): none yet
    ...                                 // print "task restarted" message to conole

    if (deadline_missed[index] > 0) {   // supervisor says we missed our deadline
      vTaskDelay( 100 );                // idle wait until supervisor resets the flag
    }
    else {                              // start or resume working
      while( work_done[index] < work_todo[index] ) { // finished?
         task_led_state[index] = 1;     // request led on (done by supervisor)
         delayMicroseconds( 50*1000 );  // spend 50 msecs of cpu time on work
         work_done[index] += 50;        // remember work done
      }
      // if we arrive here, we have completed our work, sleep until next period
      //
      task_led_state[index] = 0;        // request led off
      task_deadline[index] = t_next + period; // prepare next period deadline
      t_now = xTaskGetTickCount();      // current clock
      ...                               // print status message to console
      if (t_next > t_now) {             // check if finished in time
        vTaskDelay( t_next - t_now );   // yes, wait until end of task period
      }
    } // else deadline missed
  } // while (1)
} // periodic_worker
```

Der „Supervisor“-Task ist für die Funktion von RMS nicht notwendig, sondern dient nur zur Animation der LEDs und zur Überwachung der Deadlines. Zusätzlich übernimmt der Supervisor-Task auch das langsame Pulsieren der LED an Pin D13. Dazu läuft er mit höhrerer Priorität als die von RMS verwalteten Worker-Tasks.

Die Aktivität der Worker-Tasks kann einfach anhand der Zeitstempel im `t_latest_led_update[i]` Array ermittelt werden, die von den aktiven Worker-Tasks regelmässig (alle 50 msec) aktualisiert werden. Ein aktiver Tasks wird durch Einschalten der zugehörigen LED angezeigt.

Bei Überschreiten der Task-Deadline wird statt dessen ein schnelles Blinken der Task-LED ausgelöst und der Task (mittels `deadline_missed[i]` Flag) pausiert. Nach jeweils 3 Sekunden Zeitstrafe wird der pausierte Task dann wieder reaktiviert. Da ein derart pausierter Tasks keine Rechenzeit verbraucht, können während der Zeitstrafe auch niedriger priorisierte Tasks ausgeführt werden, die im Normalbetrieb gar nicht oder nur selten gestartet würden.

Aufgabe Z 3.11

Öffnen Sie den Sketch es_10_5_due_rtos_rms.ino und lesen Sie den Quellcode. Überlegen Sie sich zunächst, ob die voreingestellten Werte für Periode und Rechenlast („work-todo“) der fünf Tasks mit Rate-Monotonic Scheduling überhaupt zu realisieren sind? Welche Gesamtauslastung der CPU ergibt sich aus diesen Werten? Passen Sie die Perioden und/oder Rechenlasten ggf. an, um ein stabiles RMS Scheduling ohne Deadline-Verletzungen zu garantieren.

Aufgabe Z 3.12

Starten Sie die Demo und beobachten Sie die LEDs, um ein Gespür für das Zeitverhalten von RMS zu bekommen. Öffnen Sie auch die serielle Konsole, wo Start und Ende der Task-Aktivierung mit den zugehörigen Zeitpunkten protokolliert werden. Durch Eingabe von 'v' können Sie das Protokoll zwischen der detaillierten Ansicht und einer kurzen Ausgabe nur des jeweils aktiven Tasks ('A'..'E') umschalten.

Aufgabe Z 3.13

Bei jedem Tastendruck verlängert sich die Rechenlast der zugehörigen Tasks um 0.1 Sekunden, so dass das Scheduling knapper wird. Probieren Sie dies aus, und provozieren Sie nacheinander Deadline-Verletzungen (schnelles Blinken) zunächst bei Task A, dann aber auch bei den anderen Tasks. Versuchen Sie auch, gleichzeitig Deadline-Verletzungen bei zwei Tasks zu provozieren. Was muss man dafür tun?

Z 3.8 Earliest-Deadline-First Scheduling

FreeRTOS selbst stellt nur den bisher behandelten Scheduling-Algorithmus mit festen Task-Prioritäten zur Verfügung. Es gibt aber eine Reihe von Projekten, die FreeRTOS um weitere Scheduling-Algorithmen erweitern. Für die Demonstration von EDF-Scheduling benutzten wir **ESFree** oder „Efficient Scheduling Libary for FreeRTOS", das 2016 von Robin Kase im Rahmen seines Masterprojekts an der KTH Stockholm entwickelt wurde. Die Software ist quelloffen, `https://github.com/RobinK2/ESFree.git`, und in der Masterarbeit dokumentiert, `http://www.diva-portal.org/smash/get/diva2:1085303/FULLTEXT01.pdf`. Für den Einsatz an den Arduino-Compiler sind kleinere Anpassungen nötig, die in unseren Beispielprogrammen bereits enthalten sind.

Der ESFree-Scheduler unterscheidet drei Klassen von Tasks:

- Periodische Tasks mit fester Periode und Deadline, sowie dem als „Phase" bezeichneten Offset zum allerersten Starten des Tasks.
- Aperiodische Tasks, die also nicht periodisch gestartet werden sondern nur einmal durchlaufen. Diese Tasks können zu jeder beliebigen Zeit erzeugt werden, haben aber keine zugewiesene Deadline. Deshalb können diese Tasks mit geringer Priorität behandelt werden und bekommen nur Rechenzeit, wenn gerade nichts anderes zu tun ist.
- Sporadische Tasks, die ebenfalls nur einmal durchlaufen, für die aber eine Deadline gesetzt wurde. Diese Tasks können zu jeder beliebigen Zeit erzeugt werden und werden zusammen mit den periodischen Tasks anhand des EDF-Algorithmus verwaltet.

Der im letzten Abschnitt vorgestellte RMS-Algorithmus ist auf periodische Tasks beschränkt und ein erfolgreiches Scheduling kann nur garantiert werden, wenn die Gesamtauslastung der CPU unterhalb von 69% liegt. Demgegenüber unterstützt EDF alle drei Klassen von Jobs und kann das Scheduling-Problem bis zur maximalen CPU-Auslastung von 100% lösen. Bei dieser Angabe ist natürlich zu berücksichtigen, dass die genaue Ausführungszeit der einzelnen Tasks normalerweise nicht exakt bekannt ist und statt dessen die Worst-Case Execution Time (WCET) konservativ approximiert wird. Die tatsächlich mögliche CPU-Auslastung liegt deshalb auch beim Einsatz von EDF normalerweise weit niedriger. Außerdem ist der Aufwand für das Scheduling signifikant höher als bei den bisher betrachteten Verfahren, denn in jedem Zeitschritt muss der Task mit der dringensten Deadline ermittelt und dann auch gestartet werden.

Die ESFree-Software stellt zwei alternative Implementierungen des EDF-Algorithmus zur Verfügung, die in der Datei `scheduler.h` durch Definition von entweder `#define schedEDF_NAIVE` oder `#define schedEDF_EFFICIENT` ausgewählt werden.

Dabei nutzt der „naive" Algorithmus den normalen FreeRTOS-Scheduler und startet einfach einen zusätzlichen Task mit höchster Priorität, der dann zu Beginn jedes FreeRTOS Zyklus gestartet wird. Dieser Task wiederum überprüft die Deadlines aller nicht-blockierten anderen Tasks und aktiviert seinerseits den dringesten Task. Dies ist einfach zu implementieren und weitgehend mit allen anderen FreeRTOS-Mechanismen kompatibel. So funktionieren zum Beispiel Mutexe mit Prioritätsvererbung wie gewohnt. Der Nachteil des „naiven" Verfahrens ist der zusätzliche Overhead durch den ständigen Kontextwechsel vom EDF-Scheduler zum Worker-Tasks in jedem Zyklus.

Der „effiziente“ Algorithmus vermeidet diesen Nachteil und ersetzt den FreeRTOS Scheduler komplett. Für EDF werden letztlich nur zwei Prioritäten benötigt: eine hohe für den jeweils dringensten Task, der dann auch ausgeführt wird, und eine niedrige für alle übrigen Tasks.

Vom Scheduler abgesehen, bleiben die meisten grundlegenden Konzepte, Datentypen und Funktionen von FreeRTOS erhalten. Es gibt nur sechs zusätzliche Funktionen (siehe [8] Kapitel 4 für Details):

- `vSchedulerInit()`: Diese Funktion muss zu Beginn einmal vor allen anderen ESFree-Funktionen aufgerufen werden. Sie initialisiert die internen Datenstrukturen (verkettete Listen oder Arrays) für das ESFree-Scheduling.
- `vSchedulerPeriodicTaskCreate()`: Erzeugt die erweiterten Datenstrukturen (`SchedTCB` aka. Scheduler Task Control Block) für einen periodischen Task.
- `vSchedulerPeriodicTaskDelete()`: Löscht einen periodischen Task.
- `vSchedulerStart()`: Startet den ESFree-Scheduler mit dem jeweils ausgewählten Algorithmus (insbesondere RMS oder EDF) und allen vorher erzeugten periodischen Tasks.
- `vSchedulerAperiodicJobCreate()`: Erzeugt die Datenstrukturen (AJCB) für einen aperiodischen Job (ohne Deadline) und meldet diesen beim Scheduler an.
- `xSchedulerSporadicJobCreate()`: Erzeugt die Datenstrukturen (SJCB) für einen periodischen Job (mit Deadline) und meldet diesen beim Scheduler an.

Aus Effizienzgründen sind die beiden letzten Funktionen in ESFree optional. Sie stehen nur zur Verfügung, wenn Unterstützung für aperiodische und sporadische Jobs in `scheduler.h` mittels `schedUSE_APERIODIC_JOBS` und `scheUSE_SPORADIC_JOBS` aktiviert wurde.

Die Funktion zum Anlegen eines periodischen Tasks übernimmt die ersten sechs Parameter 1:1 von FreeRTOS `xCreateTask()`. Neu sind die letzten vier Parameter, die das Zeitverhalten des Tasks definieren, und die wie üblich alle in „Ticks“ (oder mittels `psMS_TO_TICKS()` in Millisekunden) gemessen werden:

```
vSchedulerPeriodicTaskCreate( worker_function, name, stack_size, &pcParam,
  priority, &cxHandle, phase, period, WCET, deadline );
```

Dabei definiert `period` die Periode des Tasks und `phase` den Zeitoffset vom Funktionsaufruf bis zur ersten Starten des Tasks an. Die (angenommene) Worst-Case Ausführungszeit und die Deadline des Tasks werden über die beiden Parameter `WCET` und `deadline` an den Scheduler übergeben.

Interessanterweise verwendet das mitgelieferte Beispielprojekt von ESFree (siehe Datei es_10_7_due_esfree_demo.ino) für seinen Task „t1“ eine Deadline von 500 msec bei einer Periode von 200 msec. Dies ergibt wenig Sinn, aber da die WCET des Tasks auf 100 msec begrenzt ist, kommt es im Beispiel nicht zu Problemen.

Wie betrachten zunächst ein vereinfachtes Beispiel, bei dem die verschiedenen Tasks (ähnlich wie oben in Aufgabe 10.5) mit Tastendruck gestartet werden können. Due Ausführung der Tasks wird dabei wieder mit den LEDs visualisiert: LED aus für einen beendeten Task, LED schwach leuchtend für einen gestarteten aber aktuell nicht aktiven Task, LED mit voller Helligkeit für den aktuell ausgeführten Task (bzw. für alle während der letzten 200 msec aktiven Tasks), und LED schnell blinkend für einen Task, der seine Deadline überschritten hat.

Für unsere Demo verwenden wir wieder fünf Tasks mit folgenden Attributen:

- Task-A: sporadisch, WCET (Workload) 500 ms, Deadline 5000 ms,
- Task-B: sporadisch, WCET (Workload) 2500 ms, Deadline 5000 ms,
- Task-C: sporadisch, WCET (Workload) 4000 ms, Deadline 5000 ms,
- Task-D: sporadisch, WCET (Workload) 1000 ms, Deadline 3000 ms,
- Task-E: periodisch, Periode 3000 ms, Workload 1000 ms.

Task-A erzeugt also nur geringe Rechenlast und kann leicht mit anderen Tasks kombiniert werden, während Task-C die CPU bereits alle die CPU zu 80% auslastet. Das Konzept der Worker-Tasks wird aus der vorherigen Aufgabe übernommen, ein aktiv rechnender Task aktualisiert also regelmässig seinen Zeitstempel im `t_latest_led_udpate`-Array. Ebenso übernehmen wir den Supervisor-Task, der für die Animation und das Blinken der LEDs zuständig ist. Dieser wird als periodischer Task erzeugt und seine Ausführung durch Setzen entsprechend kurzer Deadlines sichergestellt.

Aufgabe Z 3.14

Bitte öffnen Sie den Sketch es_10_6_due_rtos_edf.ino in der Arduino-IDE. Der Editor enthält drei Dateien, nämlich das eigentliche Arduino-Hauptprogramm mit `setup()` und `loop()` sowie die Headerdatei `scheduler.h` und `scheduler.ino` mit der Implementierung der ESFree Scheduling-Algorithmen. Laden Sie das Programm und Starten Sie verschiedene Kombinationen der Tasks A bis E durch Tastendruck.

Z 3.9 Multi-Tasking Stolpersteine

Bei der Umstellung eigener Programme auf Multi-Tasking ist zu beachten, dass es durchaus zu Problemen mit anderen Bibliotheken und Treibern für externe Sensoren kommen kann. Dies liegt einfach daran, dass ein Großteil der verfügbaren Arduino-Bibliotheken unter der single-thread Annahme entwickelt wurde und deshalb nicht immer mit präemptiven Multitasking kompatibel ist.

Zum Beispiel muss ein I2C-Bus Lesezugriff auf einen IMU-Chip die Timing-Bedingungen laut Datenblatt einhalten und darf nicht zwischendurch unterbrochen werden — schon gar nicht durch einen separaten I2C-Bus Zugriff auf einen anderen Chip. Als Abhilfe wird es in vielen Fällen ausreichen, die FreeRTOS-Tasks entsprechend auszuwählen, also zum Beispiel alle I2C-Bus Geräte nur von einem Tasks aus anzusprechen. Bei hartnäckigen Problemen könnte es nötig sein, die betroffenen Bibliotheken zu debuggen und multitask-fähig zu machen.

Literatur

[1] FreeRTOS homepage, 2024. Online: https://www.freertos.org/

[2] Richard Berry and the FreeRTOS team, Mastering the FreeRTOS™ Real Time Kernel — A Hands-On Tutorial Guide Tutorial, 2023. Online: https://www.freertos.org/Documentation/Mastering-the-FreeRTOS-Real-Time-Kernel.v1.0.pdf

[3] FreeRTOS source code (github), 2024. Online: https://github.com/FreeRTOS

[4] FreeRTOS team, What is FreeRTOS, Online: https://docs.aws.amazon.com/freertos/latest/userguide/what-is-freertos.html

[5] FreeRTOS Reference Manual, API Functions and Configuration Options, V10.0.0, 2024. Online: https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf

[6] Bill Greiman, FreeRTOS-Arduino, a port of FreeRTOS as Arduino libraries, version 8.2.3. Online: https://github.com/greiman/FreeRTOS-Arduino

[7] Mihai, DueFreeRTOS, FreeRTOS port for the Arduino Due (AT91SAM3X8E) based on FreeRTOS 10.1.1. Online: https://github.com/bdmihai/DueFreeRTOS.

[8] Robin Kase, Efficient Scheduling Library for FreeRTOS, Mster Thesis, KTH Royal Institute of Technology, Copenhagen, Online: https://github.com/RobinK2/ESFree.git, http://www.diva-portal.org/smash/get/diva2:1085303/FULLTEXT01.pdf

[9] Online: https://github.com/sachin-ik/EDF_FreeRTOS

[10] University of Michigan, EECS473 Lab Course, LAB 3: Using a Real-time Operating System, Online: http://eecs.umich.edu/courses/eecs473/Labs/Lab3F19.pdf

[11] Code Inside Out, A simple implementation of a Task Scheduler, Online: https://www.codeinsideout.com/blog/stm32/task-scheduler/

[12] Peter Marwedel, Embedded System Design. Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things, Springer, 2021, Open Access: https://link.springer.com/content/pdf/10.1007/978-3-030-60910-8.pdf