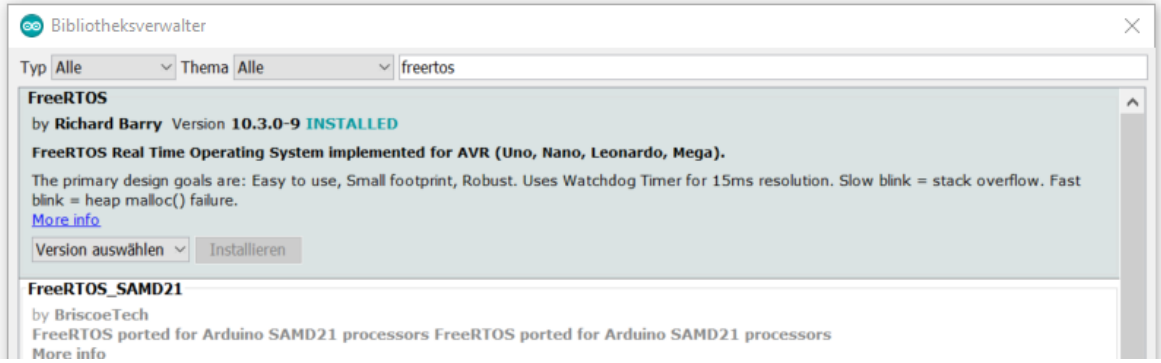
Übung 09: Embedded Betriebssysteme

Aufgabe 1: Einbindung von FreeRTOS

a) Screenshot nach der Installation:



b) Ohne Betriebssystem benötigt ein leeres Programm ca. 686 Byte.

c) Sobald man den FreeRTOS Code einbindet, benötigt man 6810 Byte. Das veranschaulicht den Overhead, den ein Embedded OS verursacht. Komponenten wie ein Scheduler müssen eben auch erst einmal implementiert werden und benötigen Code-Speicherplatz.

Aufgabe 2: Programm mit 1 Task

a) Der Task hat die Priorität 2, siehe Aufruf von `xTaskCreate(.)`. Die Stackgröße ist 128 Byte (die Wortgröße beim Atmega2560 ist 1 Byte).

b) Das Kommando setzt den Task für 1000 ms in den Zustand `BLOCKED`. Unmittelbar davor war er im Zustand `RUNNING`. Nach 100 ms wird der Task automatisch wieder zurück in den Zustand READY gesetzt. In der Zwischenzeit könnten andere Tasks arbeiten, wenn es denn welche gäbe.

c) Lösung, siehe Quelltext!

```
#include <Arduino_FreeRTOS.h>

// definition of tasks, prototypes
void vBlinkLedTask(void *pvParameters );

void setup() {

  xTaskCreate(                    // set up the blink task
    vBlinkLedTask
    ,  "Blinking LED Task"        // human-readable name
    ,  128                        // stack size
    ,  NULL
    ,  2                          // priority
    ,  NULL );

  // Now the task scheduler, which takes over control of scheduling individual tasks, is
automatically started.
}

void loop() {                     // Empty: Things are done in tasks
}
```

```
// implementation of tasks
void vBlinkLedTask(void *pvParameters)  {
  DDRB |= (1 << 7);                          // switch PB 7 / LED to output

  for (;;)  {                                // a task does not return or exit
    PORTB ^= (1 << 7);                       // toggle PB 7 / LED
    vTaskDelay(pdMS_TO_TICKS(1000));         // block / wait for 1 second
  }
}
```

Aufgabe 3: Zwei Tasks mit verschiedenen Prioritäten

a) Die interne LED ändert weiterhin jede Sekunde ihren Zustand, toggelt also. Der Grund ist, dass der Blinking Task eine höhere Priorität hat als der Busy Task. Der Scheduler wird den Blinking Task also bevorzugt zum „RUNNING“ Task machen.

b) Man beobachtet äußerlich keinen Unterschied zu a). Beide Tasks haben die gleiche Priorität. Deshalb wird der Scheduler beiden Tasks abwechselnd im Round Robin Rechenzeit einräumen. Da der Scheduler sehr schnell zwischen den Tasks wechselt, kommt der Blinking Task oft genug dran.
Hinweis: Man kann FreeRTOS auch so konfigurieren, dass bei gleichen Prioritäten der aktuelle laufende Task nicht unterbrochen wird.

c) Auch in diesem Fall blinkt die LED wie gewohnt. Der „Busy Task“ geht jeweils 1 Sekunde in den BLOCKED Zustand. Das ist genügend Zeit für den „Blinking Task“, um die LED zu toggeln.

d) Im letzten Fall blinkt die LED nicht mehr. Das delay(.) im „Busy Task“ führt nicht dazu, dass der hoch-priore „Busy Task“ in den Blocked Zustand wechselt. Er bleibt am laufen. Da er höhere Priorität als der „Blinking Task“ hat, kommt der „Blinking Task“ nicht zum Zug.

Aufgabe 4: Kommunikation zwischen Tasks - Semaphore

a) Wenn der Taster gedrückt ist, liegt LOW an. Der Eingang PA2 ist dann direkt mit GND verbunden.

b) Siehe Quelltext, gelb markiert.

c) Siehe Quelltext, gelb markiert

d) Es findet eine implizite Entprellung statt, da der „Button Task“ nur einmal jede Sekunde aufgerufen wird. Dazwischen ist er im „Blocked“ Zustand. Mechanische Schwingungen werden auf diese Weise indirekt rausgefiltert.

```
#include <Arduino_FreeRTOS.h>
#include "semphr.h"

// definition of tasks
void vBlinkLedTask(void *pvParameters);
void vButtonTask(void *pvParameters);

SemaphoreHandle_t xButtonSemaphore;    // stores handle to semaphore

void setup() {

  xTaskCreate(                    // set up Blinking task
    vToggleLedTask
    ,  "Toggle LED Task"
    ,  128  // Stack size
    ,  NULL
    ,  2
```

```
    ,  NULL );

  xTaskCreate(                    // set up Button task
    vButtonTask
    ,  "Button Task"
    ,  128  // Stack size
    ,  NULL
    ,  2
    ,  NULL );
}

void loop() {
}

// implementation of "Blinking task"
void vToggleLedTask(void *pvParameters)  {

  DDRB |= (1 << 7);      // internal LED output -> digital pin 13 / PB 7

  for (;;)  {
    if (xButtonSemaphore != NULL && xSemaphoreTake(xButtonSemaphore, 1)) {
        PORTB ^= (1 << 7); // toggle
    }
    vTaskDelay(pdMS_TO_TICKS(1000));
  }
}

void vButtonTask(void *pvParameters)  {

  DDRA &= ~(1 << 2);      // PA2 == Digital PIn 24 as input
  PORTA |= (1 << 2);    // activate internal pull-up -> p68 of manual

  xButtonSemaphore = xSemaphoreCreateBinary();      // create semaphore

  for (;;) {
     if (!(PINA & (1 << 2))) { // button pressed
         xSemaphoreGive(xButtonSemaphore);
     }
     vTaskDelay(pdMS_TO_TICKS(100));               // check every 100 ms
  }
}
```

Aufgabe 5: Kommunikation zwischen Tasks - Queues

Lösung, siehe Quelltext

```
#include <Arduino_FreeRTOS.h>
#include "queue.h"

// definition of tasks
void vUARTTask(void *pvParameters);

QueueHandle_t rxQueue;

void setup() {
  // configure UART, set up UART Receive interrupt
  UBRR0L = (unsigned char) 103;    // set buadrate
  UBRR0H = (unsigned char) (103 >> 8);

  // Enable receiver and transmitter p220
  UCSR0B |= (1<<TXEN0) | (1<<RXEN0);
```

```
  // Set frame format: 8 data, 1 stop bit, p221
  UCSR0C |=  (1<<UCSZ01) | (1<<UCSZ00);

  // activate RX interrupt
  UCSR0B |= (1<<RXCIE0);
  sei();

  xTaskCreate(   // set up the button task
    vUARTTask
    ,  "UART Task"
    ,  128  // Stack size
    ,  NULL
    ,  2                            // priority
    ,  NULL );

}

void loop() {
}

// implementation of "UART Task"
void vUARTTask(void *pvParameters)  {
  rxQueue = xQueueCreate(128,1);

  char c;

  for (;;) {
   if (xQueueReceive(rxQueue, &c, 1) == pdPASS)  { // sth in receiv queue
      while (!(UCSR0A & (1 << UDRE0)));   // echo back
      UDR0 = c;
   }
  }
}

//ISR when UART character has been received
ISR(USART0_RX_vect) {
  unsigned char ch = UDR0;
  xQueueSendFromISR(rxQueue, &ch, NULL);
}
```