Assignment 5- Implementing a closed loop control application in Zephyr

V1.0

Generated by Doxygen 1.8.17

Chapter 1

Bug List

File main.h

No known bugs.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 CMakeLists.txt File Reference

Functions

- cmake_minimum_required (VERSION 3.20.0) find_package(Zephyr REQUIRED HINTS $ENV

3.1.1 Function Documentation

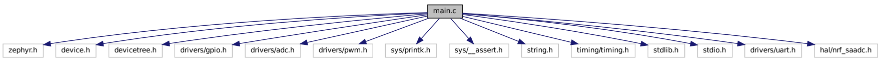
3.1.1.1 cmake_minimum_required()

```
cmake_minimum_required (
          VERSION 3.20.   0 )
```

3.2 main.c File Reference

```
#include <zephyr.h>
#include <device.h>
#include <devicetree.h>
#include <drivers/gpio.h>
#include <drivers/adc.h>
#include <drivers/pwm.h>
#include <sys/printk.h>
#include <sys/__assert.h>
#include <string.h>
#include <timing/timing.h>
#include <stdlib.h>
#include <stdio.h>
#include <drivers/uart.h>
#include <hal/nrf_saadc.h>
```

Include dependency graph for main.c:



Macros

- #define len_dados 10
- #define STACK_SIZE 1024
- #define thread_A1_prio 1
- #define thread_A_prio 1
- #define thread_B_prio 1
- #define thread_C_prio 1
- #define thread_D_prio 1
- #define SAMP_PERIOD_MS 1000
- #define ADC_NID DT_NODELABEL(adc)
- #define ADC_RESOLUTION 10
- #define ADC_GAIN ADC_GAIN_1_4
- #define ADC_REFERENCE ADC_REF_VDD_1_4
- #define ADC_ACQUISITION_TIME ADC_ACQ_TIME(ADC_ACQ_TIME_MICROSECONDS, 40)
- #define ADC_CHANNEL_ID 1
- #define ADC_CHANNEL_INPUT NRF_SAADC_INPUT_AIN1
- #define BUFFER_SIZE 1
- #define GPIO0_NID DT_NODELABEL(gpio0)
- #define PWM0_NID DT_NODELABEL(pwm0)
- #define BOARDLED1 0x0d
- #define BOARDBUT1 0xb /∗ Pin at which BUT1 is connected. Addressing is direct (i.e., pin number) ∗/
- #define BOARDBUT2 0xc
- #define BOARDBUT2 0xc
- #define BOARDBUT4 0x19
- #define FATAL_ERR -1 /∗ Fatal error return code, app terminates ∗/
- #define UART_NID DT_NODELABEL(uart0) /∗ UART Node label, see dts ∗/
- #define RXBUF_SIZE 60 /∗ RX buffer size ∗/
- #define TXBUF_SIZE 60 /∗ TX buffer size ∗/
- #define RX_TIMEOUT 1000 /∗ Inactivity period after the instant when last char was received that triggers an rx event (in us) ∗/
- #define MAIN_SLEEP_TIME_MS 10 /∗ Time between main() activations ∗/

Functions

- K_THREAD_STACK_DEFINE (thread_A1_stack, STACK_SIZE)
- K_THREAD_STACK_DEFINE (thread_A_stack, STACK_SIZE)
- K_THREAD_STACK_DEFINE (thread_B_stack, STACK_SIZE)
- K_THREAD_STACK_DEFINE (thread_C_stack, STACK_SIZE)
- K_THREAD_STACK_DEFINE (thread_D_stack, STACK_SIZE)
- void thread_A1_code (void ∗argA, void ∗argB, void ∗argC)
- void thread_A_code (void ∗argA, void ∗argB, void ∗argC)

  Lê o valor da ADC e guarda numa variável global (shared memory between tasks A/B) no nosso Código denominada por "ab" e no final faz give do semáforo AB.

- void thread_B_code (void ∗argA, void ∗argB, void ∗argC)

  Neste Script, feito o take do semáforo AB É efetuada a filtragem, em que é realizado a média das últimas 10 amostras calculadas na thread A e o filtro rejeita todos os valores que estejam abaixo ou acima de 10% da média destas amostras e faz give do semáforo BD.

- void thread_C_code (void ∗argA, void ∗argB, void ∗argC)

  Modo Manual, sempre que se carregar no botão 2 incrementa a luminosidade e ao clicar no botão 4 a luminosidade do led decrementa igualmente, ou seja, sempre que um dos botões for pressionado, o PWM varia em ± 10% do período deste, dependendo do botão que for pressionado.

- void thread_D_code (void ∗argA, void ∗argB, void ∗argC)

Modo Automatico: Faz o take que vem do semáforo BD (após a filtragem). Se o valor for menor que 500, significa que existe muita luz no meio, entao o led apaga. Quando estamos à luz ambiente, (valores lidos entre 500 e 900), o led está com uma intensidade de luz intermédia, se não existir luminosidade, (ambiente escuro) o led acende com a máxima intensidade.

- void but1press_cbfunction (const struct device ∗dev, struct gpio_callback ∗cb, uint32_t pins)

    If button 1 is pressed, Update Flag 1.
- void but2press_cbfunction (const struct device ∗dev, struct gpio_callback ∗cb, uint32_t pins)

    If button 2 is pressed, Update Flag 2.
- void but4press_cbfunction (const struct device ∗dev, struct gpio_callback ∗cb, uint32_t pins)

    If button 4 is pressed, Update Flag 4.
- void main (void)

    Main funtion: Initialize semaphores and configure GPIO_PIN.

Variables

- struct k_thread thread_A1_data
- struct k_thread thread_A_data
- struct k_thread thread_B_data
- struct k_thread thread_C_data
- struct k_thread thread_D_data
- k_tid_t thread_A1_tid
- k_tid_t thread_A_tid
- k_tid_t thread_B_tid
- k_tid_t thread_C_tid
- k_tid_t thread_D_tid
- int a1a =0
- int ab = 0
- int bc = 0
- int bd = 0
- struct k_sem sem_a1a
- struct k_sem sem_ab
- struct k_sem sem_bc
- struct k_sem sem_bd
- struct k_timer my_timer
- const struct device ∗ adc_dev = NULL
- volatile int Flag_1 = 0
- volatile int Flag_3 = 0
- volatile int Flag_2 = 0
- volatile int Flag_4 = 0
- volatile bool flag_flag = 0
- const struct device ∗ gpio0_dev
- const struct uart_config uart_cfg
- const struct device ∗ uart_dev
- volatile int uart_rx_rdy_flag
- uint8_t welcome_mesg [ ] = "UART demo: Type a few chars in a row and then pause for a little while ...\n\r"
- uint8_t rep_mesg [TXBUF_SIZE]

3.2.1 Macro Definition Documentation

3.2.1.1 ADC_ACQUISITION_TIME

```
#define ADC_ACQUISITION_TIME ADC_ACQ_TIME(ADC_ACQ_TIME_MICROSECONDS, 40)
```

3.2.1.2 ADC_CHANNEL_ID

```
#define ADC_CHANNEL_ID 1
```

3.2.1.3 ADC_CHANNEL_INPUT

```
#define ADC_CHANNEL_INPUT NRF_SAADC_INPUT_AIN1
```

3.2.1.4 ADC_GAIN

```
#define ADC_GAIN ADC_GAIN_1_4
```

3.2.1.5 ADC_NID

```
#define ADC_NID DT_NODELABEL(adc)
```

ADC definitions and includes

3.2.1.6 ADC_REFERENCE

```
#define ADC_REFERENCE ADC_REF_VDD_1_4
```

3.2.1.7 ADC_RESOLUTION

```
#define ADC_RESOLUTION 10
```

3.2.1.8 BOARDBUT1

```
#define BOARDBUT1 0xb /* Pin at which BUT1 is connected.  Addressing is direct (i.e., pin
number) */
```

3.2.1.9 BOARDBUT2 [1/2]

```
#define BOARDBUT2 0xc
```

3.2.1.10 BOARDBUT2 [2/2]

```
#define BOARDBUT2 0xc
```

3.2.1.11 BOARDBUT4

```
#define BOARDBUT4 0x19
```

3.2.1.12 BOARDLED1

```
#define BOARDLED1 0x0d
```

3.2.1.13 BUFFER_SIZE

```
#define BUFFER_SIZE 1
```

3.2.1.14 FATAL_ERR

```
#define FATAL_ERR -1 /* Fatal error return code, app terminates */
```

3.2.1.15 GPIO0_NID

```
#define GPIO0_NID DT_NODELABEL(gpio0)
```

Refer to dts file

3.2.1.16 len_dados

```
#define len_dados 10
```

Number of samples for the average

3.2.1.17 MAIN_SLEEP_TIME_MS

```
#define MAIN_SLEEP_TIME_MS 10 /* Time between main() activations */
```

3.2.1.18 PWM0_NID

```
#define PWM0_NID DT_NODELABEL(pwm0)
```

3.2.1.19 RX_TIMEOUT

```
#define RX_TIMEOUT 1000 /* Inactivity period after the instant when last char was received
that triggers an rx event (in us) */
```

3.2.1.20 RXBUF_SIZE

```
#define RXBUF_SIZE 60 /* RX buffer size */
```

3.2.1.21 SAMP_PERIOD_MS

```
#define SAMP_PERIOD_MS 1000
```

Therad periodicity (in ms)

3.2.1.22 STACK_SIZE

```
#define STACK_SIZE 1024
```

Size of stack area used by each thread (can be thread specific, if necessary)

3.2.1.23 thread_A1_prio

```
#define thread_A1_prio 1
```

Thread scheduling priority

3.2.1.24 thread_A_prio

```
#define thread_A_prio 1
```

3.2.1.25 thread_B_prio

```
#define thread_B_prio 1
```

3.2.1.26 thread_C_prio

```
#define thread_C_prio 1
```

3.2.1.27 thread_D_prio

```
#define thread_D_prio 1
```

3.2.1.28 TXBUF_SIZE

```
#define TXBUF_SIZE 60 /* TX buffer size */
```

3.2.1.29 UART_NID

```
#define UART_NID DT_NODELABEL(uart0) /* UART Node label, see dts */
```

3.2.2 Function Documentation

3.2.2.1 but1press_cbfunction()

```
void but1press_cbfunction (
              const struct device * dev,
              struct gpio_callback * cb,
              uint32_t pins )
```

If button 1 is pressed, Update Flag 1.

```
  void but1press_cbfunction(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    Flag_1 = 1;
    flag_flag=!flag_flag;
    printk("but1 \n\r");
}
```

Parameters

arg3	const struct device ∗dev, struct gpio_callback ∗cb, uint32_t pins.

Returns

No returns

3.2.2.2 but2press_cbfunction()

```
void but2press_cbfunction (
             const struct device * dev,
             struct gpio_callback * cb,
             uint32_t pins )
```

If button 2 is pressed, Update Flag 2.

```
 void but1press_cbfunction(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    Flag_2 = 1;
}
```

Parameters

arg3	const struct device ∗dev, struct gpio_callback ∗cb, uint32_t pins.

Returns

No returns

3.2.2.3 but4press_cbfunction()

```
void but4press_cbfunction (
             const struct device * dev,
             struct gpio_callback * cb,
             uint32_t pins )
```

If button 4 is pressed, Update Flag 4.

```
 void but1press_cbfunction(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    Flag_4 = 1;
}
```

Parameters

arg3	const struct device ∗dev, struct gpio_callback ∗cb, uint32_t pins.

Returns

No returns

3.2.2.4 K_THREAD_STACK_DEFINE() [1/5]

```
K_THREAD_STACK_DEFINE (
             thread_A1_stack ,
             STACK_SIZE  )
```

Create thread stack space

3.2.2.5 K_THREAD_STACK_DEFINE() [2/5]

```
K_THREAD_STACK_DEFINE (
             thread_A_stack ,
             STACK_SIZE  )
```

3.2.2.6 K_THREAD_STACK_DEFINE() [3/5]

```
K_THREAD_STACK_DEFINE (
             thread_B_stack ,
             STACK_SIZE  )
```

3.2.2.7 K_THREAD_STACK_DEFINE() [4/5]

```
K_THREAD_STACK_DEFINE (
             thread_C_stack ,
             STACK_SIZE  )
```

3.2.2.8 K_THREAD_STACK_DEFINE() [5/5]

```
K_THREAD_STACK_DEFINE (
             thread_D_stack ,
             STACK_SIZE  )
```

3.2.2.9 main()

```
void main (
            void )
```

Main funtion: Initialize semaphores and configure GPIO_PIN.

```
 void main(void) {
  int err=0;
    printf("\n\r Illustration of the use of shmem + semaphores\n\r");
    int ret = 0;
    ret = gpio_pin_configure(gpio0_dev, BOARDBUT1, GPIO_INPUT | GPIO_PULL_UP);
    ret = gpio_pin_configure(gpio0_dev, BOARDBUT2, GPIO_INPUT | GPIO_PULL_UP);
    ret = gpio_pin_configure(gpio0_dev, BOARDBUT4, GPIO_INPUT | GPIO_PULL_UP);
    if (ret < 0) {
        printk("Error %d: Failed to configure BUT 1 \r", ret);
    return;
    }
    ret = gpio_pin_interrupt_configure(gpio0_dev, BOARDBUT1, GPIO_INT_EDGE_TO_ACTIVE);
    ret = gpio_pin_interrupt_configure(gpio0_dev, BOARDBUT2, GPIO_INT_EDGE_TO_ACTIVE);
    ret = gpio_pin_interrupt_configure(gpio0_dev, BOARDBUT4, GPIO_INT_EDGE_TO_ACTIVE);
    if (ret != 0) {
    printk("Error %d: failed to configure interrupt on BUT1 pin \r", ret);
    return;
    }
    gpio_init_callback(&but1_cb_data, but1press_cbfunction, BIT(BOARDBUT1));
    gpio_add_callback(gpio0_dev, &but1_cb_data);
    gpio_init_callback(&but2_cb_data, but2press_cbfunction, BIT(BOARDBUT2));
    gpio_add_callback(gpio0_dev, &but2_cb_data);
    gpio_init_callback(&but4_cb_data, but4press_cbfunction, BIT(BOARDBUT4));
    gpio_add_callback(gpio0_dev, &but4_cb_data);


    err=0;
    uint8_t welcome_mesg[] = "UART demo: Type a few chars in a row and then pause for a little while
       ...\n\r";
    uint8_t rep_mesg[TXBUF_SIZE];


    k_sem_init(&sem_a1a, 0, 1);
    k_sem_init(&sem_ab, 0, 1);
    k_sem_init(&sem_bc, 0, 1);
     k_sem_init(&sem_bd, 0, 1);


     thread_A1_tid = k_thread_create(&thread_A1_data, thread_A1_stack,
        K_THREAD_STACK_SIZEOF(thread_A1_stack), thread_A1_code,
        NULL, NULL, NULL, thread_A1_prio, 0, K_NO_WAIT);
    thread_A_tid = k_thread_create(&thread_A_data, thread_A_stack,
        K_THREAD_STACK_SIZEOF(thread_A_stack), thread_A_code,
        NULL, NULL, NULL, thread_A_prio, 0, K_NO_WAIT);
    thread_B_tid = k_thread_create(&thread_B_data, thread_B_stack,
        K_THREAD_STACK_SIZEOF(thread_B_stack), thread_B_code,
        NULL, NULL, NULL, thread_B_prio, 0, K_NO_WAIT);
    thread_C_tid = k_thread_create(&thread_C_data, thread_C_stack,
        K_THREAD_STACK_SIZEOF(thread_C_stack), thread_C_code,
        NULL, NULL, NULL, thread_C_prio, 0, K_NO_WAIT);
    thread_D_tid = k_thread_create(&thread_D_data, thread_D_stack,
        K_THREAD_STACK_SIZEOF(thread_D_stack), thread_D_code,
        NULL, NULL, NULL, thread_D_prio, 0, K_NO_WAIT);

    return;
}
```

Parameters

NO_args	without arguments

Returns

No returns

Welcome message

Create and init semaphores

Create tasks

3.2.2.10 thread_A1_code()

```
void thread_A1_code (
             void * argA,
             void * argB,
             void * argC )
```

Thread code prototypes

3.2.2.11 thread_A_code()

```
void thread_A_code (
             void * argA,
             void * argB,
             void * argC )
```

Lê o valor da ADC e guarda numa variável global (shared memory between tasks A/B) no nosso Código denominada por “ab” e no final faz give do semáforo AB.

```
void thread_A_code(void *argA , void *argB, void *argC)
{
    int err=0;

    printk("Thread A init\n");

    adc_dev = device_get_binding(DT_LABEL(ADC_NID));
    if (!adc_dev) {
        printk("ADC device_get_binding() failed\n");
    }
    err = adc_channel_setup(adc_dev, &my_channel_cfg);
    if (err) {
        printk("adc_channel_setup() failed with error code %d\n", err);
    }


    NRF_SAADC->TASKS_CALIBRATEOFFSET = 1;
    while(1) {
        k_sem_take(&sem_a1a,  K_FOREVER);

        err=adc_sample();
        if(err) {
            printk("adc_sample() failed with error code %d\n\r",err);
        }
        else {
            if(adc_sample_buffer[0] > 1023) {
                printk("adc reading out of range\n\r");
            }
            else {

                ab=adc_sample_buffer[0];
            }
        }
        printk("Thread A set ab value to: %d ",ab);

        k_sem_give(&sem_ab);


    }
}
```

Parameters

arg3	void *argA , void *argB, void *argC.

Returns

No returns

3.2.2.12 thread_B_code()

```
void thread_B_code (
              void * argA,
              void * argB,
              void * argC )
```

Neste Script, feito o take do semáforo AB É efetuada a filtragem, em que é realizado a média das últimas 10 amostras calculadas na thread A e o filtro rejeita todos os valores que estejam abaixo ou acima de 10% da média destas amostras e faz give do semáforo BD.

```
void thread_B_code(void *argA , void *argB, void *argC)
{
    int Array_dados[len_dados]={0};
    int k=0;
    printk("Thread B init (sporadic, waits on a semaphore by task A)\n");
    while(1) {
        int sumador=0,somador_2=0,media=0, media_filtered=0;
        int contador=0;

        k_sem_take(&sem_ab,  K_FOREVER);

        printk("Task B read ab value: %d\n",ab);
        for(int k=len_dados-1; k>0;k--){

        Array_dados[k]= Array_dados[k-1];
        }
        Array_dados[0]= ab;

       for(int i = 0; i < len_dados; i++){
            if(Array_dados[i] != 0){
                sumador = sumador + Array_dados[i];
            }
        }
        media=sumador/len_dados;
        contador=0;

        for(int j = 0; j < len_dados; j++){
            if(Array_dados[j] < (media - media*0.1) || Array_dados[j] > (media + media*0.1))
                somador_2=somador_2;
            else{
                somador_2 = somador_2 + Array_dados[j];
                contador =contador +1;

            }
        }
        if(somador_2 != 0)
            media_filtered=somador_2/contador;
        else
            media_filtered = 0;
        bd=ab;
        printk("Thread B set bc value to: %d\n",bc);
        k_sem_give(&sem_bd);
  }
}
```

Parameters

arg3	void ∗argA , void ∗argB, void ∗argC.

Returns

No returns

array de dados da adc

3.2.2.13 thread_C_code()

```
void thread_C_code (
              void * argA,
              void * argB,
              void * argC )
```

Modo Manual, sempre que se carregar no botão 2 incrementa a luminosidade e ao clicar no botão 4 a luminosidade do led decrementa igualmente, ou seja, sempre que um dos botões for pressionado, o PWM varia em ± 10% do período deste, dependendo do botão que for pressionado.

```
 void thread_C_code(void *argA , void *argB, void *argC)
{
    const struct device *gpio0_dev;
    const struct device *pwm0_dev;
    int ret=0;
    unsigned int dcValue[]={100,90,80,70,60,50,40,30,20,10,0};
    unsigned int dcIndex=0;
    unsigned int pwmPeriod_us = 100;
    printk("Thread C init (sporadic, waits on a semaphore by task B)\n");

    gpio0_dev = device_get_binding(DT_LABEL(GPIO0_NID));
    if (gpio0_dev == NULL) {
        printk("Error: Failed to bind to GPIO0\n\r");
    return;
    }

    pwm0_dev = device_get_binding(DT_LABEL(PWM0_NID));
    if (pwm0_dev == NULL) {
    printk("Error: Failed to bind to PWM0\n r");
    return;
    }

    while(1) {
        k_sem_take(&sem_bc, K_FOREVER);
        ret=0;

        if(Flag_2) {
            dcIndex++;
            if(dcIndex == 11)
                dcIndex = 0;
            Flag_2 = 0;
            printk("PWM DC value set to %u %%\n\r",dcValue[dcIndex]);
            ret = pwm_pin_set_usec(pwm0_dev, BOARDLED1,
              pwmPeriod_us,(unsigned int)((pwmPeriod_us*dcValue[dcIndex])/100), PWM_POLARITY_NORMAL);
            if (ret) {
                printk("Error %d: failed to set pulse width\n", ret);
        return;
            }
        }
        if(Flag_4) {

            if(dcIndex == 0)
                dcIndex = 11;
            dcIndex--;
            Flag_4 = 0;
            printk("PWM DC value set to %u %%\n\r",dcValue[dcIndex]);
            ret = pwm_pin_set_usec(pwm0_dev, BOARDLED1,
              pwmPeriod_us,(unsigned int)((pwmPeriod_us*dcValue[dcIndex])/100), PWM_POLARITY_NORMAL);
            if (ret) {
                printk("Error %d: failed to set pulse width\n", ret);
        return;
            }
        }

        printk("Task C - PWM: %u % n", (unsigned int)(((pwmPeriod_us*bc)/1023)/10));
    }
}
```

Parameters

arg3	void *argA , void *argB, void *argC.

Returns

No returns

Prints dutty-cycle

3.2.2.14 thread_D_code()

```
void thread_D_code (
               void * argA,
               void * argB,
               void * argC )
```

Modo Automatico: Faz o take que vem do semáforo BD (após a filtragem). Se o valor for menor que 500, significa que existe muita luz no meio, entao o led apaga. Quando estamos à luz ambiente, (valores lidos entre 500 e 900), o led está com uma intensidade de luz intermédia, se não existir luminosidade, (ambiente escuro) o led acende com a máxima intensidade.

```
 void thread_D_code(void *argA , void *argB, void *argC)
{
    const struct device *gpio0_dev;
    const struct device *pwm0_dev;
    int ret=0;
    unsigned int dcValue[]={100,90,80,70,60,50,40,30,20,10,0};
    unsigned int dcIndex=0;
    unsigned int pwmPeriod_us = 100;
    printk("Thread C init (sporadic, waits on a semaphore by task B)\n");


    gpio0_dev = device_get_binding(DT_LABEL(GPIO0_NID));
    if (gpio0_dev == NULL) {
        printk("Error: Failed to bind to GPIO0\n\r");
    return;
    }

    pwm0_dev = device_get_binding(DT_LABEL(PWM0_NID));
    if (pwm0_dev == NULL) {
    printk("Error: Failed to bind to PWM0\n r");
    return;
    }

    while(1) {
        k_sem_take(&sem_bd, K_FOREVER);
        printk("Valor lido para automatico %d\n\r",bd);
        ret=0;

        if(bd<500) {

            ret = pwm_pin_set_usec(pwm0_dev, BOARDLED1,
              pwmPeriod_us,(unsigned int)(pwmPeriod_us), PWM_POLARITY_NORMAL);
            if (ret) {
                printk("Error %d: failed to set pulse width\n", ret);
        return;
            }
        }
        else if(bd>500 && bd<900) {

            ret = pwm_pin_set_usec(pwm0_dev, BOARDLED1,
              pwmPeriod_us,(unsigned int)(pwmPeriod_us*0.5), PWM_POLARITY_NORMAL);
            if (ret) {
                printk("Error %d: failed to set pulse width\n", ret);
        return;
            }
        }
        else {

            ret = pwm_pin_set_usec(pwm0_dev, BOARDLED1,
              pwmPeriod_us,(unsigned int)(0), PWM_POLARITY_NORMAL);
            if (ret) {
                printk("Error %d: failed to set pulse width\n", ret);
        return;
            }
        }

 }
 }
```

Parameters

NO_args	without arguments
arg3	void ∗argA , void ∗argB, void ∗argC.

Takes one adc_sample

```
  static int adc_sample(void)
{
    int ret;
    const struct adc_sequence sequence = {
        .channels = BIT(ADC_CHANNEL_ID),
        .buffer = adc_sample_buffer,
        .buffer_size = sizeof(adc_sample_buffer),
        .resolution = ADC_RESOLUTION,
    };
    if (adc_dev == NULL) {
            printk("adc_sample(): error, must bind to adc first \r");
            return -1;
    }
    ret = adc_read(adc_dev, &sequence);
    if (ret) {
            printk("adc_read() failed with code %d\n", ret);
    }
    return ret;
}
```

Parameters

NO_args	without arguments

Returns

Read ADC_sample value (static int)

3.2.3 Variable Documentation

3.2.3.1 a1a

```
int a1a =0
```

Global vars (shared memory between tasks A/B and B/C, resp)

3.2.3.2 ab

```
int ab = 0
```

3.2.3.3 adc_dev

```
const struct device* adc_dev = NULL
```

3.2.3.4 bc

```
int bc = 0
```

3.2.3.5 bd

```
int bd = 0
```

3.2.3.6 Flag_1

```
volatile int Flag_1 = 0
```

3.2.3.7 Flag_2

```
volatile int Flag_2 = 0
```

3.2.3.8 Flag_3

```
volatile int Flag_3 = 0
```

3.2.3.9 Flag_4

```
volatile int Flag_4 = 0
```

3.2.3.10 flag_flag

```
volatile bool flag_flag = 0
```

3.2.3.11 gpio0_dev

```
const struct device* gpio0_dev
```

3.2.3.12 my_timer

```
struct k_timer my_timer
```

Global vars

3.2.3.13 rep_mesg

```
uint8_t rep_mesg[TXBUF_SIZE]
```

3.2.3.14 sem_a1a

```
struct k_sem sem_a1a
```

Semaphores for task synch

3.2.3.15 sem_ab

```
struct k_sem sem_ab
```

3.2.3.16 sem_bc

```
struct k_sem sem_bc
```

3.2.3.17 sem_bd

```
struct k_sem sem_bd
```

3.2.3.18 thread_A1_data

```
struct k_thread thread_A1_data
```

Create variables for thread data

3.2.3.19 thread_A1_tid

```
k_tid_t thread_A1_tid
```

Create task IDs

3.2.3.20 thread_A_data

```
struct k_thread thread_A_data
```

3.2.3.21 thread_A_tid

```
k_tid_t thread_A_tid
```

3.2.3.22 thread_B_data

```
struct k_thread thread_B_data
```

3.2.3.23 thread_B_tid

```
k_tid_t thread_B_tid
```

3.2.3.24 thread_C_data

```
struct k_thread thread_C_data
```

3.2.3.25 thread_C_tid

```
k_tid_t thread_C_tid
```

3.2.3.26 thread_D_data

```
struct k_thread thread_D_data
```

3.2.3.27 thread_D_tid

```
k_tid_t thread_D_tid
```

3.2.3.28 uart_cfg

```
const struct uart_config uart_cfg
```

Initial value:

```
= {
        .baudrate = 115200,
        .parity = UART_CFG_PARITY_NONE,
        .stop_bits = UART_CFG_STOP_BITS_1,
        .data_bits = UART_CFG_DATA_BITS_8,
        .flow_ctrl = UART_CFG_FLOW_CTRL_NONE
}
```

3.2.3.29 uart_dev

```
const struct device* uart_dev
```

3.2.3.30 uart_rx_rdy_flag

```
volatile int uart_rx_rdy_flag
```

3.2.3.31 welcome_mesg

```
uint8_t welcome_mesg[ ] = "UART demo:  Type a few chars in a row and then pause for a little
while ...\n\r"
```

Main function

3.3 main.h File Reference

Application to control the light intensity of a given region. The system comprises a light sensor, an illumination system and a Human-Machine Interface. The system can operate in two modes: • Automatic: programmable via the terminal. Should allow setting On/Off periods and the corresponding light intensity; • Manual: interface via the DevKit buttons. Allows to turn the system On/Off (when in “Off” the system operates in automatic mode), via two of the buttons. The other two buttons allow to set (increase/decrease) the desired intensity.

Functions

- void main (void)

  Main funtion: Initialize semaphores and configure GPIO_PIN.
- void thread_A_code (void ∗argA, void ∗argB, void ∗argC)

  Lê o valor da ADC e guarda numa variável global (shared memory between tasks A/B) no nosso Código denominada por "ab" e no final faz give do semáforo AB.
- void thread_A1_code (void ∗argA, void ∗argB, void ∗argC)
- void thread_B_code (void ∗argA, void ∗argB, void ∗argC)

  Neste Script, feito o take do semáforo AB É efetuada a filtragem, em que é realizado a média das últimas 10 amostras calculadas na thread A e o filtro rejeita todos os valores que estejam abaixo ou acima de 10% da média destas amostras e faz give do semáforo BD.
- void thread_C_code (void ∗argA, void ∗argB, void ∗argC)

  Modo Manual, sempre que se carregar no botão 2 incrementa a luminosidade e ao clicar no botão 4 a luminosidade do led decrementa igualmente, ou seja, sempre que um dos botões for pressionado, o PWM varia em ± 10% do período deste, dependendo do botão que for pressionado.
- void thread_D_code (void ∗argA, void ∗argB, void ∗argC) static int adc_sample(void)

  Modo Automatico: Faz o take que vem do semáforo BD (após a filtragem). Se o valor for menor que 500, significa que existe muita luz no meio, entao o led apaga. Quando estamos à luz ambiente, (valores lidos entre 500 e 900), o led está com uma intensidade de luz intermédia, se não existir luminosidade, (ambiente escuro) o led acende com a máxima intensidade.
- void but1press_cbfunction (const struct device ∗dev, struct gpio_callback ∗cb, uint32_t pins)

  If button 1 is pressed, Update Flag 1.
- void but2press_cbfunction (const struct device ∗dev, struct gpio_callback ∗cb, uint32_t pins)

  If button 2 is pressed, Update Flag 2.
- void but4press_cbfunction (const struct device ∗dev, struct gpio_callback ∗cb, uint32_t pins)

  If button 4 is pressed, Update Flag 4.

3.3.1 Detailed Description

Application to control the light intensity of a given region. The system comprises a light sensor, an illumination system and a Human-Machine Interface. The system can operate in two modes: • Automatic: programmable via the terminal. Should allow setting On/Off periods and the corresponding light intensity; • Manual: interface via the DevKit buttons. Allows to turn the system On/Off (when in "Off" the system operates in automatic mode), via two of the buttons. The other two buttons allow to set (increase/decrease) the desired intensity.

Author

Frederico Moreira, Ana Sousa, Pedro Rodrigues

Date

21 June 2022

Bug No known bugs.

3.3.2 Function Documentation

3.3.2.1 but1press_cbfunction()

```
void but1press_cbfunction (
             const struct device ∗ dev,
             struct gpio_callback ∗ cb,
             uint32_t pins )
```

If button 1 is pressed, Update Flag 1.

```
  void but1press_cbfunction(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    Flag_1 = 1;
    flag_flag=!flag_flag;
    printk("but1 \n\r");
}
```

Parameters

arg3	const struct device ∗dev, struct gpio_callback ∗cb, uint32_t pins.

Returns

No returns

3.3.2.2 but2press_cbfunction()

```
void but2press_cbfunction (
             const struct device ∗ dev,
             struct gpio_callback ∗ cb,
             uint32_t pins )
```

If button 2 is pressed, Update Flag 2.

```
 void but1press_cbfunction(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    Flag_2 = 1;
}
```

Parameters

arg3	const struct device ∗dev, struct gpio_callback ∗cb, uint32_t pins.

Returns

No returns

3.3.2.3 but4press_cbfunction()

```
void but4press_cbfunction (
             const struct device ∗ dev,
             struct gpio_callback * cb,
             uint32_t pins )
```

If button 4 is pressed, Update Flag 4.

```
 void but1press_cbfunction(const struct device *dev, struct gpio_callback *cb, uint32_t pins)
{
    Flag_4 = 1;
}
```

Parameters

arg3	const struct device ∗dev, struct gpio_callback ∗cb, uint32_t pins.

Returns

No returns

3.3.2.4 main()

```
void main (
               void )
```

Main funtion: Initialize semaphores and configure GPIO_PIN.

```
 void main(void) {
  int err=0;
    printf("\n\r Illustration of the use of shmem + semaphores\n\r");
    int ret = 0;
    ret = gpio_pin_configure(gpio0_dev, BOARDBUT1, GPIO_INPUT | GPIO_PULL_UP);
    ret = gpio_pin_configure(gpio0_dev, BOARDBUT2, GPIO_INPUT | GPIO_PULL_UP);
    ret = gpio_pin_configure(gpio0_dev, BOARDBUT4, GPIO_INPUT | GPIO_PULL_UP);
    if (ret < 0) {
        printk("Error %d: Failed to configure BUT 1 \r", ret);
    return;
    }
    ret = gpio_pin_interrupt_configure(gpio0_dev, BOARDBUT1, GPIO_INT_EDGE_TO_ACTIVE);
    ret = gpio_pin_interrupt_configure(gpio0_dev, BOARDBUT2, GPIO_INT_EDGE_TO_ACTIVE);
    ret = gpio_pin_interrupt_configure(gpio0_dev, BOARDBUT4, GPIO_INT_EDGE_TO_ACTIVE);
    if (ret != 0) {
    printk("Error %d: failed to configure interrupt on BUT1 pin \r", ret);
    return;
    }
    gpio_init_callback(&but1_cb_data, but1press_cbfunction, BIT(BOARDBUT1));
    gpio_add_callback(gpio0_dev, &but1_cb_data);
    gpio_init_callback(&but2_cb_data, but2press_cbfunction, BIT(BOARDBUT2));
    gpio_add_callback(gpio0_dev, &but2_cb_data);
    gpio_init_callback(&but4_cb_data, but4press_cbfunction, BIT(BOARDBUT4));
    gpio_add_callback(gpio0_dev, &but4_cb_data);


    err=0;
    uint8_t welcome_mesg[] = "UART demo: Type a few chars in a row and then pause for a little while
       ...\n\r";
    uint8_t rep_mesg[TXBUF_SIZE];


    k_sem_init(&sem_a1a, 0, 1);
    k_sem_init(&sem_ab, 0, 1);
    k_sem_init(&sem_bc, 0, 1);
     k_sem_init(&sem_bd, 0, 1);


     thread_A1_tid = k_thread_create(&thread_A1_data, thread_A1_stack,
        K_THREAD_STACK_SIZEOF(thread_A1_stack), thread_A1_code,
        NULL, NULL, NULL, thread_A1_prio, 0, K_NO_WAIT);
    thread_A_tid = k_thread_create(&thread_A_data, thread_A_stack,
        K_THREAD_STACK_SIZEOF(thread_A_stack), thread_A_code,
        NULL, NULL, NULL, thread_A_prio, 0, K_NO_WAIT);
    thread_B_tid = k_thread_create(&thread_B_data, thread_B_stack,
        K_THREAD_STACK_SIZEOF(thread_B_stack), thread_B_code,
        NULL, NULL, NULL, thread_B_prio, 0, K_NO_WAIT);
    thread_C_tid = k_thread_create(&thread_C_data, thread_C_stack,
        K_THREAD_STACK_SIZEOF(thread_C_stack), thread_C_code,
        NULL, NULL, NULL, thread_C_prio, 0, K_NO_WAIT);
    thread_D_tid = k_thread_create(&thread_D_data, thread_D_stack,
        K_THREAD_STACK_SIZEOF(thread_D_stack), thread_D_code,
        NULL, NULL, NULL, thread_D_prio, 0, K_NO_WAIT);

    return;
}
```

Parameters

NO_args	without arguments

Returns

No returns

Welcome message

Create and init semaphores

Create tasks

3.3.2.5 thread_A1_code()

```
void thread_A1_code (
              void * argA,
              void * argB,
              void * argC )
```

Neste script faz-se o toggle entre o sistema manual e o sistema automático através do botão 1, sendo esta thread periódica. Caso estejamos perante o caso Modo Manual, a próxima thread a ser executada é a thread ‘C’, caso contrário, é a thread ‘A’ a ser executada.

```
void thread_A1_code(void *argA , void *argB, void *argC)
{

    int64_t fin_time=0, release_time=0;
    int err=0;
    printk("Thread A1 init (periodic)\n");

    release_time = k_uptime_get() + SAMP_PERIOD_MS;
    while(1) {


        if (flag_flag==0){
         printk("Modo manual\n");
         printk("Modo key 2 e 4\n");
         k_sem_give(&sem_bc);}

        else{

           printk("Modo automatico\n");

          k_sem_give(&sem_a1a);
          }
        fin_time = k_uptime_get();
        if( fin_time < release_time) {
            k_msleep(release_time - fin_time);
            release_time += SAMP_PERIOD_MS;
        }
    }

}
```

Parameters

arg3	void ∗argA , void ∗argB, void ∗argC.

Returns

No returns

Thread code prototypes

3.3.2.6 thread_A_code()

```
void thread_A_code (
            void * argA,
            void * argB,
            void * argC )
```

Lê o valor da ADC e guarda numa variável global (shared memory between tasks A/B) no nosso Código denominada por “ab” e no final faz give do semáforo AB.

```
void thread_A_code(void *argA , void *argB, void *argC)
{
    int err=0;

    printk("Thread A init\n");

    adc_dev = device_get_binding(DT_LABEL(ADC_NID));
    if (!adc_dev) {
        printk("ADC device_get_binding() failed\n");
    }
    err = adc_channel_setup(adc_dev, &my_channel_cfg);
    if (err) {
        printk("adc_channel_setup() failed with error code %d\n", err);
    }


    NRF_SAADC->TASKS_CALIBRATEOFFSET = 1;
    while(1) {
        k_sem_take(&sem_a1a,  K_FOREVER);

        err=adc_sample();
        if(err) {
            printk("adc_sample() failed with error code %d\n\r",err);
        }
        else {
            if(adc_sample_buffer[0] > 1023) {
                printk("adc reading out of range\n\r");
            }
            else {

                ab=adc_sample_buffer[0];
            }
        }
        printk("Thread A set ab value to: %d ",ab);

        k_sem_give(&sem_ab);


    }
}
```

Parameters

arg3	void ∗argA , void ∗argB, void ∗argC.

Returns

No returns

3.3.2.7 thread_B_code()

```
void thread_B_code (
              void * argA,
              void * argB,
              void * argC )
```

Neste Script, feito o take do semáforo AB É efetuada a filtragem, em que é realizado a média das últimas 10 amostras calculadas na thread A e o filtro rejeita todos os valores que estejam abaixo ou acima de 10% da média destas amostras e faz give do semáforo BD.

```
void thread_B_code(void *argA , void *argB, void *argC)
{
    int Array_dados[len_dados]={0};
    int k=0;
    printk("Thread B init (sporadic, waits on a semaphore by task A)\n");
    while(1) {
        int sumador=0,somador_2=0,media=0, media_filtered=0;
        int contador=0;

        k_sem_take(&sem_ab,  K_FOREVER);

        printk("Task B read ab value: %d\n",ab);
        for(int k=len_dados-1; k>0;k--){

        Array_dados[k]= Array_dados[k-1];
        }
        Array_dados[0]= ab;

       for(int i = 0; i < len_dados; i++){
            if(Array_dados[i] != 0){
                sumador = sumador + Array_dados[i];
            }
        }
        media=sumador/len_dados;
        contador=0;

        for(int j = 0; j < len_dados; j++){
            if(Array_dados[j] < (media - media*0.1) || Array_dados[j] > (media + media*0.1))
                somador_2=somador_2;
            else{
                somador_2 = somador_2 + Array_dados[j];
                contador =contador +1;

            }
        }
        if(somador_2 != 0)
            media_filtered=somador_2/contador;
        else
            media_filtered = 0;
        bd=ab;
        printk("Thread B set bc value to: %d\n",bc);
        k_sem_give(&sem_bd);
  }
}
```

Parameters

arg3	void *argA , void *argB, void *argC.

Returns

No returns

array de dados da adc

3.3.2.8 thread_C_code()

```
void thread_C_code (
               void * argA,
               void * argB,
               void * argC )
```

Modo Manual, sempre que se carregar no botão 2 incrementa a luminosidade e ao clicar no botão 4 a luminosidade do led decrementa igualmente, ou seja, sempre que um dos botões for pressionado, o PWM varia em ± 10% do período deste, dependendo do botão que for pressionado.

```
 void thread_C_code(void *argA , void *argB, void *argC)
{
    const struct device *gpio0_dev;
    const struct device *pwm0_dev;
    int ret=0;
    unsigned int dcValue[]={100,90,80,70,60,50,40,30,20,10,0};
    unsigned int dcIndex=0;
    unsigned int pwmPeriod_us = 100;
    printk("Thread C init (sporadic, waits on a semaphore by task B)\n");

    gpio0_dev = device_get_binding(DT_LABEL(GPIO0_NID));
    if (gpio0_dev == NULL) {
        printk("Error: Failed to bind to GPIO0\n\r");
    return;
    }

    pwm0_dev = device_get_binding(DT_LABEL(PWM0_NID));
    if (pwm0_dev == NULL) {
    printk("Error: Failed to bind to PWM0\n r");
    return;
    }

    while(1) {
        k_sem_take(&sem_bc, K_FOREVER);
        ret=0;

        if(Flag_2) {
            dcIndex++;
            if(dcIndex == 11)
                dcIndex = 0;
            Flag_2 = 0;
            printk("PWM DC value set to %u %%\n\r",dcValue[dcIndex]);
            ret = pwm_pin_set_usec(pwm0_dev, BOARDLED1,
              pwmPeriod_us,(unsigned int)((pwmPeriod_us*dcValue[dcIndex])/100), PWM_POLARITY_NORMAL);
            if (ret) {
                printk("Error %d: failed to set pulse width\n", ret);
        return;
            }
        }
        if(Flag_4) {

            if(dcIndex == 0)
                dcIndex = 11;
            dcIndex--;
            Flag_4 = 0;
            printk("PWM DC value set to %u %%\n\r",dcValue[dcIndex]);
            ret = pwm_pin_set_usec(pwm0_dev, BOARDLED1,
              pwmPeriod_us,(unsigned int)((pwmPeriod_us*dcValue[dcIndex])/100), PWM_POLARITY_NORMAL);
            if (ret) {
                printk("Error %d: failed to set pulse width\n", ret);
        return;
            }
        }

        printk("Task C - PWM: %u % n", (unsigned int)(((pwmPeriod_us*bc)/1023)/10));
    }
}
```

Parameters

arg3	void *argA , void *argB, void *argC.

Returns

No returns

Prints dutty-cycle

3.3.2.9 thread_D_code()

```
void thread_D_code (
               void * argA,
               void * argB,
               void * argC )
```

Modo Automatico: Faz o take que vem do semáforo BD (após a filtragem). Se o valor for menor que 500, significa que existe muita luz no meio, entao o led apaga. Quando estamos à luz ambiente, (valores lidos entre 500 e 900), o led está com uma intensidade de luz intermédia, se não existir luminosidade, (ambiente escuro) o led acende com a máxima intensidade.

```
 void thread_D_code(void *argA , void *argB, void *argC)
{
    const struct device *gpio0_dev;
    const struct device *pwm0_dev;
    int ret=0;
    unsigned int dcValue[]={100,90,80,70,60,50,40,30,20,10,0};
    unsigned int dcIndex=0;
    unsigned int pwmPeriod_us = 100;
    printk("Thread C init (sporadic, waits on a semaphore by task B)\n");


    gpio0_dev = device_get_binding(DT_LABEL(GPIO0_NID));
    if (gpio0_dev == NULL) {
        printk("Error: Failed to bind to GPIO0\n\r");
    return;
    }

    pwm0_dev = device_get_binding(DT_LABEL(PWM0_NID));
    if (pwm0_dev == NULL) {
    printk("Error: Failed to bind to PWM0\n r");
    return;
    }

    while(1) {
        k_sem_take(&sem_bd, K_FOREVER);
        printk("Valor lido para automatico %d\n\r",bd);
        ret=0;

        if(bd<500) {

            ret = pwm_pin_set_usec(pwm0_dev, BOARDLED1,
              pwmPeriod_us,(unsigned int)(pwmPeriod_us), PWM_POLARITY_NORMAL);
            if (ret) {
                printk("Error %d: failed to set pulse width\n", ret);
        return;
            }
        }
        else if(bd>500 && bd<900) {

            ret = pwm_pin_set_usec(pwm0_dev, BOARDLED1,
              pwmPeriod_us,(unsigned int)(pwmPeriod_us*0.5), PWM_POLARITY_NORMAL);
            if (ret) {
                printk("Error %d: failed to set pulse width\n", ret);
        return;
            }
        }
        else {

            ret = pwm_pin_set_usec(pwm0_dev, BOARDLED1,
              pwmPeriod_us,(unsigned int)(0), PWM_POLARITY_NORMAL);
            if (ret) {
                printk("Error %d: failed to set pulse width\n", ret);
        return;
            }
        }

 }
 }
```

Parameters

NO_args	without arguments
arg3	void ∗argA , void ∗argB, void ∗argC.

Takes one adc_sample

```
  static int adc_sample(void)
{
    int ret;
    const struct adc_sequence sequence = {
        .channels = BIT(ADC_CHANNEL_ID),
        .buffer = adc_sample_buffer,
        .buffer_size = sizeof(adc_sample_buffer),
        .resolution = ADC_RESOLUTION,
    };
    if (adc_dev == NULL) {
            printk("adc_sample(): error, must bind to adc first \r");
            return -1;
    }
    ret = adc_read(adc_dev, &sequence);
    if (ret) {
            printk("adc_read() failed with code %d\n", ret);
    }
    return ret;
}
```

Parameters

NO_args	without arguments

Returns

Read ADC_sample value (static int)

Index