# SOTER – Final Work

JOSÉ SILVA - 1130352

DANIEL FREIRE - 1130858

# Task Creation

```
void startAllTasks(void){

    I2CtoUSARTQueue=xQueueCreate(10,sizeof(I2C_data_type));
    I2CtoPongQueue=xQueueCreate(1,sizeof(float));

    //I2C and USARTSend have higher priorities to make sure the data is always logged.
    //Pong never runs at the same time as UI, and has higher priority than debugLED to keep up the framerate
    //UI has the same priority as debugLED because the screen is only updated at 1Hz
    xTaskCreate((void*)debugLED,    "debugLED",     task_stacks[DebugLEDTaskEnum], NULL,   tskIDLE_PRIORITY+1,     &task_handlers[DebugLEDTaskEnum]);
    xTaskCreate((void*)I2C,         "I2C",          task_stacks[I2CEnum],          NULL,   tskIDLE_PRIORITY+3,     &task_handlers[I2CEnum]);
    xTaskCreate((void*)USARTSend,   "USARTSend",    task_stacks[USARTSendEnum],    NULL,   tskIDLE_PRIORITY+3,     &task_handlers[USARTSendEnum]);
    xTaskCreate((void*)UI,          "UI",           task_stacks[UIEnum],           NULL,   tskIDLE_PRIORITY+1,     &task_handlers[UIEnum]);
    xTaskCreate((void*)Pong,        "Pong",         task_stacks[PongEnum],         NULL,   tskIDLE_PRIORITY+2,     &task_handlers[PongEnum]);
}
```

# Task Creation

```
enum taskenum{
    DebugLEDTaskEnum,
    I2CEnum,
    USARTSendEnum,
    UIEnum,
    PongEnum,
    NumberOfTasks
};

extern unsigned short task_stacks[NumberOfTasks];

TaskHandle_t task_handlers[NumberOfTasks];
```

```
//File-scope typedef
typedef struct I2C_data{
    float gx,gy,gz;
    uint32_t timestamp;
}I2C_data_type;

//File-scope Queues
QueueHandle_t I2CtoUSARTQueue,I2CtoPongQueue;

//Global-scope task stacks
unsigned short task_stacks[NumberOfTasks]={
        configMINIMAL_STACK_SIZE,       //DebugLEDTask
        configMINIMAL_STACK_SIZE,       //I2CTask
        configMINIMAL_STACK_SIZE+100,   //USARTSend
        configMINIMAL_STACK_SIZE+62,    //UI
        configMINIMAL_STACK_SIZE+100    //Pong
};
```

# UI Task

```
//Print queues to the screen
lcd_draw_string(0,0,"Messages in queues:",WHITE,1);
sprintf(buffer,"I2CtoUSARTQueue: %lu  ",uxQueueMessagesWaiting(I2CtoUSARTQueue));
lcd_draw_string(0,10,buffer,WHITE,1);
sprintf(buffer,"I2CtoPongQueue: %lu  ",uxQueueMessagesWaiting(I2CtoPongQueue));
lcd_draw_string(0,20,buffer,WHITE,1);


//Print all tasks to the screen
lcd_draw_string(0,40,"Worst free stack:",0xFFFF,1);
for(i=0;i<NumberOfTasks;i++){
    sprintf(buffer,"%s:%lu of %hu ",pcTaskGetName(task_handlers[i]),uxTaskGetStackHighWaterMark(task_handlers[i]),task_stacks[i]*2);
    lcd_draw_string(0,50+i*10,buffer,WHITE,1);
}
```

# UI Task

```
//"Poll" 4 times for the switches and wait a total for 1000ms
//This was necessary to make the transition to pong smoother, as only checking once very 1 second felt very slow
for(i=0;i<4;i++){

    vTaskDelayUntil(&previousWakeTime, 250/portTICK_PERIOD_MS);

    if(getSwitchAction(0)==CenterEnum){
        //Switch out of this task and enter the Pong task
        lcd_draw_fillrect(0,0,LCD_WIDTH,LCD_HEIGHT,BLACK);
        vTaskResume(task_handlers[PongEnum]);
        vTaskSuspend(NULL);

        //If the code gets here, pong has selected this task to run
        vTaskDelay(100/portTICK_PERIOD_MS);
        flushSwitchAction();

        //Get new execution time
        previousWakeTime=xTaskGetTickCount();
        break;
    }

}
```

# USARTSend

```
xQueueReceive(I2CtoUSARTQueue,&I2Cdata,portMAX_DELAY);
sprintf(buffer,"X=%0.2fg|Y=%0.2fg|Z=%0.2fg|TSTAMP=%ld\n",I2Cdata.gx,I2Cdata.gy,I2Cdata.gz,I2Cdata.timestamp);
USARTPutString(buffer);

//Adding a slight offset between the USART wait and I2C wait will increase the I2CtoUSART queue slowly.
vTaskDelayUntil(&previousWakeTime,50/portTICK_PERIOD_MS);
```

# I2C (Task)

```
static void I2C(void){
    static TickType_t previousWakeTime;
    static I2C_data_type I2Cdata;

    previousWakeTime=xTaskGetTickCount();

    startupAccel();

    watchdogStart();

    while(1){

        getAccelData(&I2Cdata.gx,&I2Cdata.gy,&I2Cdata.gz);
        I2Cdata.timestamp=xTaskGetTickCount();

        //Flush old data if the USART failed to send the data
        if(xQueueSendToBack(I2CtoUSARTQueue,&I2Cdata,0)!=pdPASS){
            xQueueReset(I2CtoUSARTQueue);
            xQueueSendToBack(I2CtoUSARTQueue,&I2Cdata,0);
        }

        xQueueOverwrite(I2CtoPongQueue,&I2Cdata.gx);

        //Reset watchdog before 1 second passes
        watchdogReset();
        //Get data at a 20Hz rate
        vTaskDelayUntil(&previousWakeTime,50/portTICK_PERIOD_MS);
    }
}
```

# Watchdog reset


```
int main(void) {

    initLowLevel(10,10,115200);

    lcd_init();

    if(systemWasResetByWatchdog()){

        lcd_draw_string(25,0,"System",RED,2);
        lcd_draw_string(20,20,"Failure",RED,2);
        lcd_draw_string(20,120,"Watchdog reset",WHITE,1);
        while(1);
    }

    startAllTasks();

    lcdIntroScreen();

    vTaskStartScheduler();

    //Will get stuck in this loop if there wasn't enough heap space for idle task
    while(1);
}
```

# Pong (Task)

```
while(1){

    xQueueReceive(I2CtoPongQueue,&gx,0);

    logic(gx);

    render(&previousWakeTime);

    //50ms = 20FPS
    vTaskDelayUntil(&previousWakeTime,50/portTICK_PERIOD_MS);

    //State transition
    if(getSwitchAction(0)==CenterEnum){
        //Switch back to the UI task
        lcd_draw_fillrect(0,0,LCD_WIDTH,LCD_HEIGHT,BLACK);
        vTaskResume(task_handlers[UIEnum]);
        vTaskSuspend(NULL);

        //Flush all switch actions in case a double click on SW5 occurred
        vTaskDelay(100/portTICK_PERIOD_MS);
        flushSwitchAction();
        previousWakeTime=xTaskGetTickCount();
        gameInit();
    }
}
```

# Idle Hook

```
#define configUSE_PREEMPTION        1
#define configUSE_IDLE_HOOK         1
#define configUSE_TICK_HOOK         0
#define configCPU_CLOCK_HZ          ( ( unsigned long ) 72000000 )
#define configTICK_RATE_HZ          ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES        ( 5 )
#define configMINIMAL_STACK_SIZE    ( ( unsigned short ) 128 )
#define configTOTAL_HEAP_SIZE       ( ( size_t ) ( 14 * 1024 ) )
#define configMAX_TASK_NAME_LEN     ( 16 )
#define configUSE_TRACE_FACILITY    0 //for vTaskList set to 1
#define configUSE_16_BIT_TICKS      0
#define configIDLE_SHOULD_YIELD     1
#define configUSE_MUTEXES           1
```

```
//Idle hook
void vApplicationIdleHook( void ){
    //Wait for interrupt to wake up
    __WFI();
}
```

# Pong


Messages in queues:
I2CtoUSARTQueue: 0
I2CtoPongQueue: 1
Worst free stack:
debugLED:104 of 256
I2C:92 of 256
USARTSend:51 of 456
UI:41 of 380
Pong:89 of 456
Press SW5 for PONG!

# Pong

# “Standard” Peripheral Library

- Terrible documentation
  - PDF version does not match website version
  - Both have macros that do not exist in the actual library
- Inconsistency
  - All macros about GPIO outputs are GPIO_Mode_Out_xx, but inputs can be GPIO_Mode_IN_FLOATING or GPIO_Mode_IPU or _IPD.... what??? Also, uppercase “IN” but no uppercase “Out”?!
- Bad code
  - Disabling any USART interrupt actually disables **all** USART interrupts.
- Deprecated
  - New “Hardware abstraction layer” is prefered over the “standard” peripheral library, as it is no longer ported to new ST microcontrollers!
  - Congratulations! **Almost everything you learned in SISEM is now useless!**

# CMSIS - Cortex Microcontroller Software Interface Standard

- Standard way of accessing **all ARM Cortex microcontroller registers.**
  - Same programming style, doesn't matter if the micro is from ST, NXP, Infineon, etc...
- Standard way of configuring all interrupts in all ARM Cortex microcontrollers
  - NVIC is a part of the ARM processor and not part of the peripherals
- Only portable to the same series of microcontrolers because the registers are different, so it's less portable than the standard peripheral library...
  - **Except if the standard peripheral library changed between microcontroller families.... *Which it does!***
- The best documentation you can get
  - Which is the reference manual of the microcontroller you want to program.
- If something doesn't work, it's always ***your fault***!*

*Except when it isn't and the hardware is broken in very peculiar ways, such as the I2C events. But that's why you always check the errata for your microcontroller! ☺

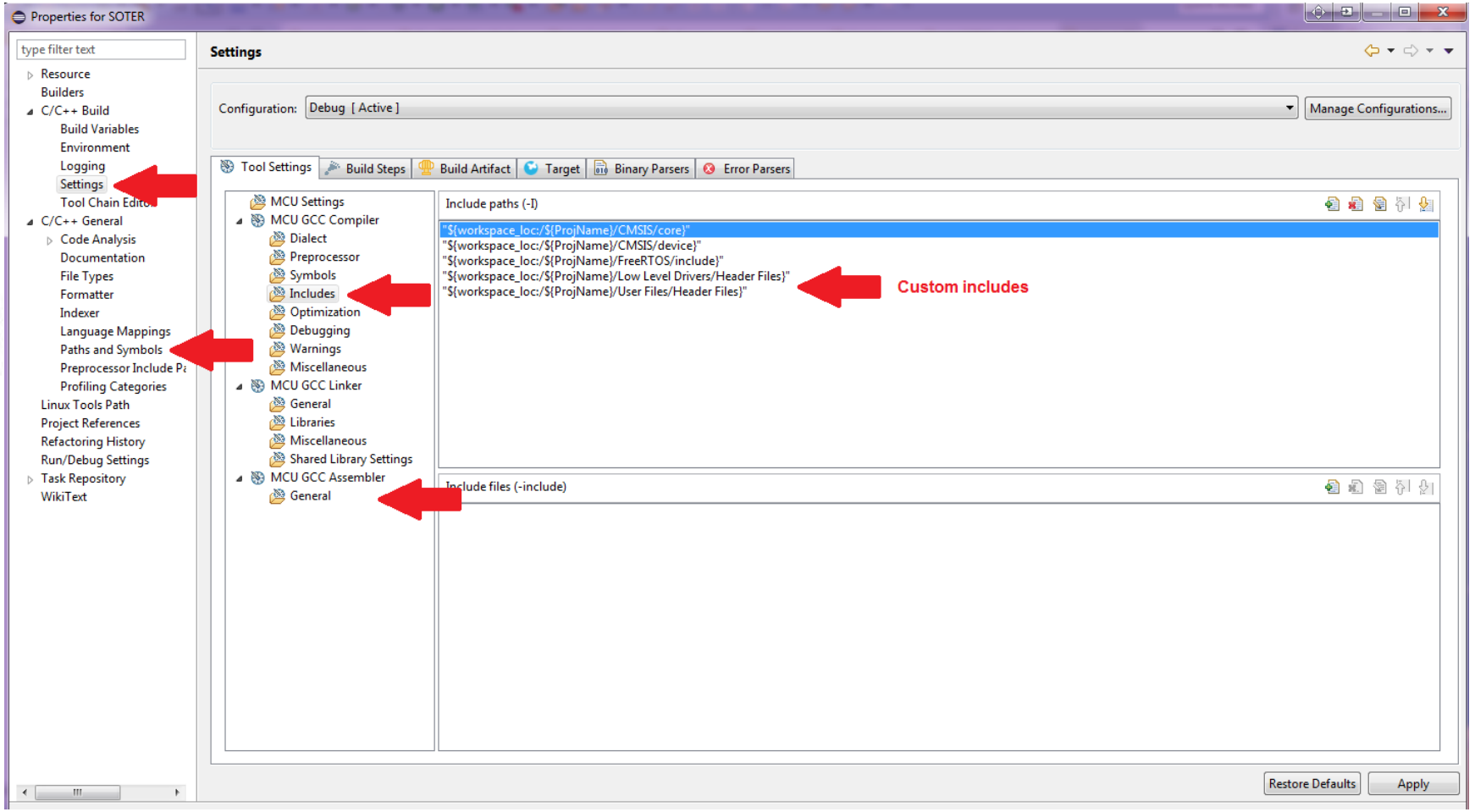
# How to decrapify a project?

- ST doesn’t want you using CMSIS – otherwise you wouldn’t be stuck to their microcontrollers!
  - Because of this, ST doesn’t provide a direct download, but all Cortex microcontrollers have it!
- 1) Start a project with the HAL or Standard Library. Choose the option to “Add low level drivers as sources in the application project”. This will create a separate CMSIS folder.
- 2) Remove every file related to the standard library/HAL. Leave only the assembly startup file.
  - This startup file has weak aliases for the interrupt handlers.
- 3) Go to the project properties and remove all the includes from the standard library/HAL in C/C++ Build > Settings.
- 4) Go to C/C++ General > Paths and Symbols and remove all include directories for the standard library/HAL for assembly and GNU GCC.
- 5) In the same place, go to the “Symbols” tab and remove all macros related to the libraries (such as USE_STDPERIPH_DRIVER).

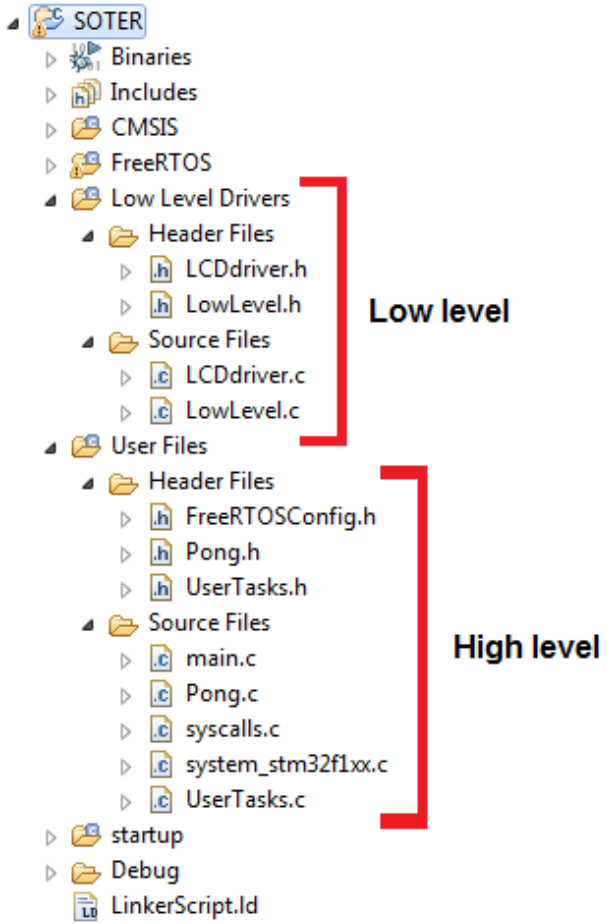
# How to decrapify a project?

- Pro tips:
  - You can also add your own include paths for custom folders!
  - You can add custom configuration-dependent macros! Want to blink a LED in a debug version? Easy! Just define a DEBUG macro in the “DEBUG” configuration and use the preprocessor to check for this macro with #ifdef
  - You can also change the optimization level in the C/C++ Build > Settings > MCU GCC Compiler > Settings

# How to decrapify a project?



# Getting organized



# Watchdog? Easy

```
//Independent watchdog functions
//*************************************************************
void watchdogStart(void){
    IWDG->KR|=0xCCCC;
}

void watchdogReset(void){
    IWDG->KR|=0xAAAA;
}

uint8_t systemWasResetByWatchdog(void){
    return ( (RCC->CSR&RCC_CSR_IWDGRSTF) != 0 );
}

//*************************************************************
```

# I2C? Not so easy

```
//See if SB bit is set
if((I2C2->SR1&I2C_SR1_SB)!=0){
    if(I2C_struct.flag==0){
        //Send address with LSB reset to enter transmitter mode
        I2C2->DR= (I2C_struct.slave_address<<1);
        //Reading SR1 and writing to DR resets the SB flag
    }else{
        //This is a re-start condition, so send the device address again, now in receiver mode
        I2C2->DR= (I2C_struct.slave_address<<1) | 0x1;
        //TX is now clear because of the start condition. Enable the RX and TX interrupts to receive data
        I2C2->CR2|= (I2C_CR2_ITBUFEN);
    }
}

//When an address is sent to the slave, this interrupt happens
if( (I2C2->SR1&I2C_SR1_ADDR) !=0){
    //SR2 read is necessary to reset the ADDR flag
    I2C_struct.data[0]=I2C2->SR2;
    if(I2C_struct.flag==0){
        //Send register address to read
        I2C2->DR=I2C_struct.register_address;
    }else if(I2C_struct.registers_to_read==1){
        //Just one byte to receive means the NACK must be configured in here
        I2C2->CR1 &= ~(I2C_CR1_ACK);
        I2C2->CR1 |= I2C_CR1_STOP;
    }
    //If the flag here is 1, it means that we've already sent the device address twice
    //so we can't write nothing to DR because we are about to receive data
}

//Register address was sent
if( (I2C2->SR1&I2C_SR1_TXE)!= 0){
    //Generate a repeated start
    I2C2->CR1|=I2C_CR1_START;
    I2C2->DR|=0;
    //Disable TX interrupts because nothing can be written to DR
    I2C2->CR2&= ~(I2C_CR2_ITBUFEN);
    I2C_struct.flag=1;
}

//Data received
if( (I2C2->SR1&I2C_SR1_RXNE)!= 0){
    I2C_struct.data[I2C_struct.data_counter]=I2C2->DR;

    if(I2C_struct.data_counter==(I2C_struct.registers_to_read-1)){
        //Last byte was sent, I2C interrupts are no longer necessary
        NVIC_DisableIRQ(I2C2_EV_IRQn);
        //A stop condition was already generated, so just resume the current task
        vTaskResume(BlockedI2CTask);
    }else if(I2C_struct.data_counter==(I2C_struct.registers_to_read-2)){
        //Prepare NACK and stop condition at second-last byte
        I2C2->CR1 &= ~(I2C_CR1_ACK);
        I2C2->CR1 |= I2C_CR1_STOP;
    }

    I2C_struct.data_counter++;

}
```

```
//Start condition
if((I2C2->SR1&I2C_SR1_SB)!=0){

    I2C2->DR= (I2C_struct.slave_address<<1);

}

//Slave address sent and ACK received (ADDR bit set)
if( (I2C2->SR1&I2C_SR1_ADDR) !=0){
    //Read SR2 just to to clear ADDR bit. data[1] is unused for sending data
    I2C_struct.data[1]=I2C2->SR2;
    //Send register to write to slave
    I2C2->DR=I2C_struct.register_address;
}

//Transmission buffer is empty due to ADDR clear or sent byte ACK received
if( (I2C2->SR1&I2C_SR1_TXE)!= 0){
    if(I2C_struct.flag==0){
        //Send data
        I2C2->DR=I2C_struct.data[0];
        I2C_struct.flag=1; //Indicates that the next TXE interrupt is end of transmission
    }else{
        //ACK received, send stop condition
        I2C2->CR1 |= I2C_CR1_STOP;
        NVIC_DisableIRQ(I2C2_EV_IRQn);
        vTaskResume(BlockedI2CTask);
    }
}
```

# I2C – File-scope struct data setup

```
uint8_t writetoI2C(uint8_t slave_address, uint8_t slave_register, uint8_t data){
    uint8_t i=0;

    //All the logic here is similar to the read function

    if(xSemaphoreTake(I2Csemaphore,portMAX_DELAY)!=pdPASS){
        return I2C_ERR;
    }

    for(i=0;i<5;i++){
        if( (I2C2->SR2&I2C_SR2_BUSY)==0){
            break;
        }
    }

    if(i==5){
        return I2C_ERR;
    }

    I2C_struct.slave_address=slave_address;
    I2C_struct.register_address=slave_register;
    I2C_struct.data[0]=data;
    I2C_struct.data_counter=0;
    I2C_struct.action_type=I2C_SendEnum;
    I2C_struct.flag=0;

    BlockedI2CTask=xTaskGetCurrentTaskHandle();

    //Enable I2C's IRQ
    NVIC_EnableIRQ(I2C2_EV_IRQn);

    //Generate a start condition and turn on ACKs
    I2C2->CR1|=I2C_CR1_START|I2C_CR1_ACK;

    //Block task until I2C communication is over
    vTaskSuspend(BlockedI2CTask);

    xSemaphoreGive(I2Csemaphore);

    return I2C_OK;
}
```

# Switch internal queues

```
swenum getSwitchAction(TickType_t xTicksToWait){
    swenum SWvalue;

    if(xQueueReceive(SWQueue,&SWvalue,xTicksToWait)!=pdPASS){
        return NoAction;
    }else{
        return SWvalue;
    }
}
```

```
//Switch enumerator type
typedef enum swenum{
    RightEnum,
    LeftEnum,
    UpEnum,
    DownEnum,
    CenterEnum,
    NoAction
}swenum;
```

# Switch internal queues

```
void flushSwitchAction(void){
    xQueueReset(SWQueue);
}

void EXTI1_IRQHandler(void){
    static BaseType_t pxHigherPriorityTaskWoken;
    static swenum SWvalue;

    //SW5
    SWvalue=CenterEnum;
    xQueueSendToBackFromISR(SWQueue,&SWvalue,&pxHigherPriorityTaskWoken);

    if(pxHigherPriorityTaskWoken==pdTRUE){
        taskYIELD();
    }

    EXTI->PR |= EXTI_PR_PR1;
    NVIC_ClearPendingIRQ(EXTI1_IRQn);
}
```

# USART

```
BaseType_t USARTPutString(char* str){
    uint8_t i=0;

    if(xSemaphoreTake(USARTSemaphore,portMAX_DELAY)!=pdPASS){
        return pdFAIL;
    }

    while(str[i]){
        if(xQueueSendToBack(TXQueue,&str[i],0)!=pdPASS){

            USART2->CR1|= USART_CR1_TXEIE;
            BlockedUSART2Task=xTaskGetCurrentTaskHandle();
            vTaskSuspend(BlockedUSART2Task);
            xQueueSendToBack(TXQueue,&str[i],0);
        }
        i++;
    }

    USART2->CR1|= USART_CR1_TXEIE;

    xSemaphoreGive(USARTSemaphore);

    return pdPASS;
}
```

# USART

```
if(xQueueReceiveFromISR(TXQueue,&data2,&pxHigherPriorityTaskWoken)==pdPASS){
    USART2->DR=data2;
    if( pxHigherPriorityTaskWoken == pdTRUE ){
        taskYIELD();
    }
}else if (xQueueIsQueueEmptyFromISR(TXQueue)!=pdFALSE){
    //If the queue is empty, stop all transmition by disabling the interrupt
    USART2->CR1&= ~(USART_CR1_TXEIE);

    if(BlockedUSART2Task!=NULL){
        vTaskResume(BlockedUSART2Task);
        BlockedUSART2Task=NULL;
    }
}
```

# USART

```
//USARTDIV = DIV_Mantissa + (DIV_Fraction / 16) -> Page 794 RM0008
//Tx/Rx baud rate = fPCLK1 / (USARTDIV *16) -> Page 803 RM0008

//Example:

//Desired baud rate: 115200 -> desired USARTDIV is 19.53125
// DIV_Fraction = 16 * 0.53125 = 8.5 -> round to 9 = 0x9
// Mantisa is just 19
// real USARTDIV = 19 + (9 / 16) = 19.5625

//Configuration would look like this:
//USART2->BRR|=(19<<4)|9;
//Remember to shift the mantissa by 4 bits! Page 825 RM0008
```

# USART

```
uint32_t mantissa_final,fractional_final;
float usartdiv, fractional;

//WARNING: DON'T FORGET TO TYPECAST TO FLOAT! Otherwise division by integer returns no fractional part!!!!!
usartdiv = (float) 36000000/(USART2baudrate*16);

//Get fractional part of USARTDIV
fractional = usartdiv - (long)usartdiv;

//Get whole part of USARTDIV
mantissa_final = usartdiv-fractional;

//Get fractional part by multiplying by 16 and rounding up. (fractional*16)+0.5 will never be higher than 16.499(9)
//meaning that fractonal_final will never be higher than 16!
fractional_final = (uint32_t) ((fractional*16)+0.5);

//If the fraction is bigger than 4bits (i.e. it's 0d16), carry 1 to the mantissa and subtract 0d15 or 0xF to the fractional
if(fractional_final>0xF){
    fractional_final -= 0xF;
    mantissa_final++;
}

USART2->BRR |= (mantissa_final<<4)|fractional_final;
```

# Want to go down the same path?

- Check it out:
  - https://github.com/Kagehiko/CMSIS-FreeRTOS-STM32F103

# Questions?