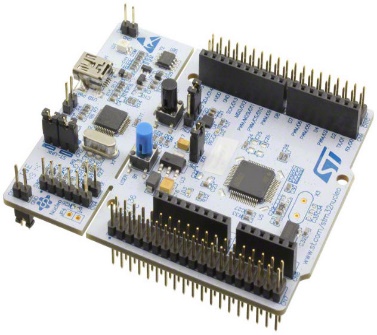
***REAL-TIME Operating System for STM32F103 based Microcontroller***

***Background***

The core objective of this project was to design and implement some libraries that will act as a Real-time Operating System for STM32f103 based microcontrollers. It incorporates multithreading along with other functionalities that will help the embedded programmers to write codes with great comfort.The library includes various on-chip resources like Timers, UART, SPI, ADC, DAC, GPIOS easily.This will make the usage of complex STM based Micro-Controllers as simple as Arduino.

Moreover, you can create/delete threads by calling functions.

All these libraries have been written on Keil uVision4 and tested on Nucleo STM32F103RB Micro-Controller.

***Introduction***

The first thing you need to do before including any library in you code, is to copy *preprocessor.h* file and paste it in your project folder. This file contains preprocessors, that are used by other libraries. If this file is not in your project folder, you will get errors.

Now that this is aside, let’s explore the libraries.

***Threads*** (thread.h)

Multi-Threading is the most important features of this library. It allows you to create/delete threads merely by calling simple functions.

First thing you want to do, before creating threads is to call the Thread\_Init() function from main(). The Thread\_Init() function is mandatory to call as it initializes the SysTick Timer. Without initializing the timer, there will be no scheduling between threads.

Each thread has a time slice assigned to it. The time slice for a process or thread is the time the process is allowed to run. Once the time slice is expired next process will run. Round Robin Scheduling Algorithm is used to schedule between the threads.

Note:

0x20004A38 to 0x20004EE8 is a memory region that stores Thread Control Blocks(TCB). DO NOT WRITE TO THIS MEMORY REGION.

***MULTI-THREADING FUNCTIONS***

1. ***Creating a Thread:***

In order to create a thread, call the following function:



It takes the address of the function you want to run as a thread. That function must not have any return type or parameters. In other words, the return type and parameter of the function must be void. This is one of the limitations of Create\_Threads() i.e. it does not handle functions with parameters or return type.

You can create up to 10 threads. The function that you want to run as a thread must contain an infinite loop. In other world the function must not return. Also note that the prototype of the function must include,

\_\_attribute\_\_((noreturn));

This tells the compiler that the function will not return (see example below).

The function returns 0 if unsuccessful, else it returns thread ID. You can create threads inside other threads also.

Here’s an example of creating a thread,

***Example:***

******

This small program will create a thread running Task\_1 function and that thread will get scheduled anytime. Each thread has its own stack. The size of the stack is 120 bytes but 80 bytes are used by the OS and remaining 40 bytes are used for local variables. Keep this in mind when declaring variables in thread. YOU HAVE 40 BYTES FOR LOCAL VARIABLE IN THREADS. If you exceed stack size, it will cause undefined behavior.

1. ***Deleting a Thread:***

In order to delete a thread, the function is:



And it takes the ID of the thread.

The function returns 0 if unsuccessful, and returns 1 on successful.

Thread can only be deleted in main() function. Threads cannot be deleted in its own function, nor it can be deleted from other thread.

***Example:***

Continuing the previous example.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***THREAD-SYNCHRONIZATION FUNCTIONS*** (synchronization.h)

Thread synchronization is the concurrent execution of two or more threads that share critical resources. However, conflicts may arise when parallel-running threads attempt to modify a shared variable at the same time.

In order to avoid these conflicts, the RTOS library provides mutex/semaphores functions.

Following are the functions of semaphores and mutex.

***SEMAPHORES:***

******

***MUTEX:***

******

This Example Demonstrates how to use **Mutex**:



Similarly, you can use semaphores in the same way. This example shows how to initialize and use semaphore variable.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***GPIOs (General Purpose Input Output)*** (gpio.h)

A General Purpose Input/output (**GPIO**) is an interface available on most modern **microcontrollers** (MCU) including Stm32f103, to provide an ease of access to the devices internal properties. Generally, there are multiple **GPIO** pins on a single MCU for the use of multiple interaction so simultaneous application.

STM32F103 GPIO are generic pins that can be configured as input or output. These GPIO pins can be configured in 4 different modes (input mode, output mode, analog input mode, and alternate function mode).

I’ve provided a library that allow you to configure the GPIOs merely by calling simple functions with appropriate arguments. Following are the functions that are provided by the library.



**Description:**

Call this function to configure GPIO pins before using it. You can call this function multiple times if you want to configure multiple pins.

**Parameters:**

-> **port** = A,B,C,D

-> **mode** = 0,1,2,3

0: Input Mode

1: Output mode, max speed 10 Mhz

2: Output mode, max speed 2 MHZ

3: Output mode, max speed 50 MHZ

-> **Operation** = 0,1,2,3

(In case of Input Mode)

0: Analog Mode

1: Floating Input(reset state)

2: Input with pull-up/pull-down

3: Reserved

(In case of Output Mode)

0: General Purpose output push-pull

1: General Purpose output open-drain

2: Alternate function output push-pull

3: Alternate function output open-drain

-> **pinNumber**: 0-15



**Description:**

Call this function to enable a pin.

**Parameters:**

-> **port** = A,B,C,D

-> **pinNumber**: 0-15 (The pin number that you want to enable)



**Description:**

Call this function to disable a pin.

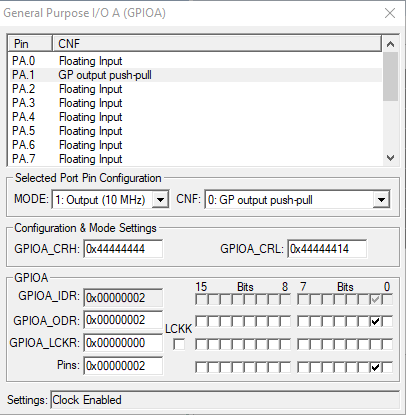
**Parameters:**

-> **port** = A,B,C,D

-> **pinNumber**: 0-15 (The pin number that you want to enable)

Following is an example program that configures PA1 as output, and then enables that pin.





\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Timers*** (timers.h)

Three General purpose timers (TIM2, TIM3, TIM4) in STM32F103 are used to provide the functions to delay the execution of a process/thread for a specified time in milliseconds. All the three timers are independent from one another.

Also note that these timers are configured to run on 8 MHz HSI clock (default in STM32F103) or 48 MHz PLL (HSI feed) clock. Any other clock frequency will produce inaccurate results.

The library provides three Delay functions,

Following is an example program that uses Delay\_1 function.



Delay\_2 and Delay\_3 works in exactly the same way.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***UART*** (uart.h)

Uart stands for Universal Asynchronous Receiver/Transmitter. It’s main purpose is to transmit and receive serial data.

The library provides 3 Uart (Uart1, Uart2, Uart3) interfaces. Uart1 is the fastest, and it operates at 48Mhz Pll (with HSI feed) clock source and must be used independently. Uart2 and Uart3 operates on default 8Mhz HSI clock source.

Following are the functions that are provided in the library.

1. **int UART1\_Init(int baudrate, short parity, short stop)**

**Description:**

Call this function before using UART\_1. It configures the USART1 registers (Usart1 is used for uart1). It also changes the clock source to 48Mhz Pll. You also don’t need to configure the GPIOs, as this function does that for you. However, you do need to hook up the wires.

* port A pin 9(TX)
* port A pin 10(RX)

On successful configuration, UART will transmit “Link Ok” (see image below).

**Parameters:**

-> **baudrate** = Enter the required baud rate e.g. 9600, 112500 etc

-> **parity** = 0,1,2

0: No Parity

1: Even Parity

2: Odd Parity

If parity is enabled, and parity error occurs, then a variable ***\_parityError\_1*** is set to 1. You can use this variable and compare it with 1 to see if parity error has occurred or not. For Uart2 and Uart3 the variables are, ***\_parityError\_2 and \_parityError\_3*** respectively.

-> **stop** = 1,2

1: 1 Stop Bit

2: 2 Stop Bit

1. **void UART1\_Send(char\* data)**
2. **char UART1\_Receive()**
3. **void UART1\_Finish()**

**Description:**

Calling UART1\_Finish() will reset all UART1 registers, and reset the clock source back to 8MHz HSI.

Other Uart interfaces (Uart2 and Uart3) also has the same functions. The only differences are that both of these (Uart2 and Uart3) operates on default 8Mhz clock, and of course, both have different tx and rx pins.

For Uart2,

* USART2 TX -> PA2
* USART2 RX -> PA3

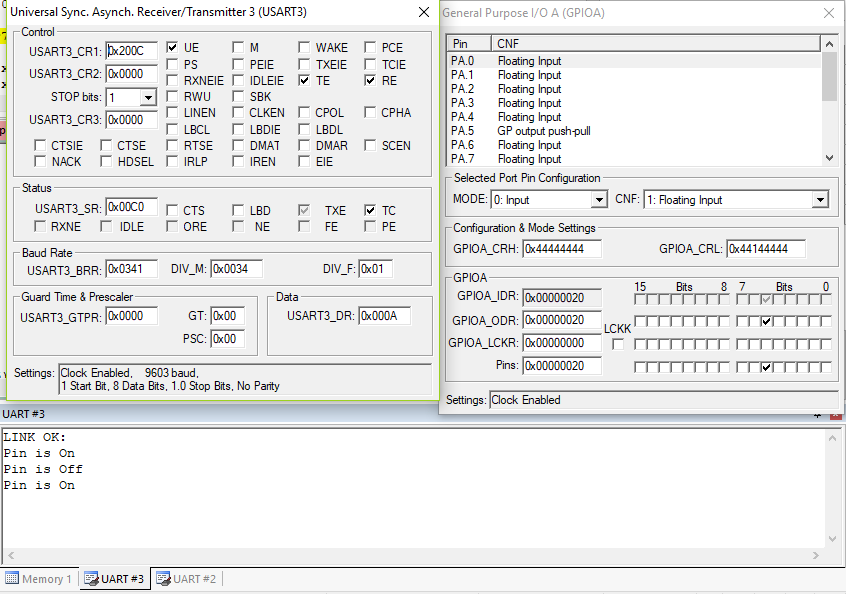
Also note that by default PA2 and PA3 pins (i.e. D1 and D0) on STM32 Nucleo are connected to the ST-LINK/V2-1 interface, and therefore not usable without adding and removing some bridges first.

For Uart3,

* USART3 TX -> PB10
* USART3 RX -> PB11

Here’s the program that demonstrates the Uart (Uart3 is used)





Uart1 and Uart2 are similar.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***SPI (Serial Peripheral Interface)*** (spi.h)

The SPI bus is a serial communication protocol for controlling and communicating with almost any digital electronics device that accepts a clocked serial stream of bits. SPI is typically used for short distance communication and commonly found in embedded devices.

SPI devices communicate in full duplex mode, meaning communication can travel in two directions - in this case Master-Slave. This Master-Slave architecture enables a single master to control multiple slaves (max 3 slaves in this library).

The library allows both master and slave capability. Following are the functions provided by library.

1. **void Init\_Master(short baudrate, short LSB\_First, short mode, int numbOfSlaves)**

**Description:**

Call this function if you want your micro-controller as master. You don't need to configure GPIOs. This function does that for you.

Also note that SPI1 runs at 48Mhz PLL clock source (provided by HCI) instead of usual 8Mhz HCI. Using SPI1 might affect other peripherals that are configured to run with default 8Mhz HSI clock source.

In master mode, you can attach up to 3 slaves,

->**Slave 1**:

Connect SS\_1(Slave Select for 1st Slave) to PA4

Connect SCK(source clock) to PA5

Connect MISO(Master In Slave Out) to PA6

Connect MOSI(Master Out Slave In) to PA7

->**Slave 2**:

Connect SS\_2 to PB6

(SCK, MISO, MOSI are common)

->**Slave 3**:

Connect SS\_3 to PC7

(SCK, MISO, MOSI are common)

Connect the slaves in order, e.g. if you want to connect only one slave, use Slave 1 settings, if you want to connect 2 slaves use slave 1 and slave 2 settings and so on.

**Parameters:**

-> baudrate = 0,1,2,3,4,5,6,7

0: fPCLK/2

1: fPCLK/4

2: fPCLK/8

3: fPCLK/16

4: fPCLK/32

5: fPCLK/64

6: fPCLK/128

7: fPCLK/256

**fPCLK = 48Mhz**

-> **LSB\_First** = 0,1

0: MSB

1: LSB

-> **mode** = 0,1,2,3

0: CPOL = 0, CPHA = 0

1: CPOL = 0, CPHA = 1

2: CPOL = 1, CPHA = 0

3: CPOL = 1, CPHA = 1

CPOL = Clock Polarity, CPHA = Clock Phase

-> **numbOfSlaves**: 1,2,3 (Maximum 3 slaves can be attached)

Enter the number of slaves connected to the master.

1: GPIO of first slave will be initlized

2: GPIO of first and second slaves will be initlized

3: GPIO of first and second and third slaves will be initlized

If you want to attach 1 slave then used slave 1 settings, if you want to attach 2 slaves use slave 1 and slave 2 settings. For three slaves using slave 1 and slave 2 and slave 3 settings (See function description above).

1. **void Init\_Slave(short nss, short LSB\_First, short mode, short multiSlave)**

**Description:**

Call this function if you want your micro-controller as slave.

**Parameters:**

-> **nss** = 0,1

0: Hardware Nss

1: Software Nss

-> **LSB\_First** = 0,1

0: MSB

1: LSB

-> **mode** = 0,1,2,3

0: CPOL = 0, CPHA = 0

1: CPOL = 0, CPHA = 1

2: CPOL = 1, CPHA = 0

3: CPOL = 1, CPHA = 1

CPOL = Clock Polarity, CPHA = Clock Phase

-> **multiSlave** = 0,1

0 = if this device is the only slave connected to master (Point to point)

1 = if this device is one of the many slaves are connected to master

1. **unsigned char SPI\_Transfer(unsigned char data)**

**Description**:

Only 8 bit data will be sent/receive. The function sends/receives data simultaneously.

1. **void Enable\_Slave(short slaveNumb)**

**Description:**

Use this function in master mode to enable a slave before transmission/reception

**Parameter:**

**->slaveNumb**: 1,2,3

Enter the number of slave you want to enable

1: First Slave will be enabled

2: Second Slave will be enabled

3: Third Slave will be enabled

1. **void Disable\_Slave(short slaveNumb)**

**Description:**

Use this function in master mode to disable a slave after transmission/reception.

1. **void SPI\_Finish()**

**Description:**

Use this function at the end to terminate the SPI. This will reset the source clock as well (in case of master).

Here’s the example program where STM32F103 acts as a master.



Here’s an example where STM32F103 acts as a slave.



In this program, slave receives the value from master, and then send a confirmation message via UART.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***ADC (Analog to Digital)*** (spi.h)

An Analog to Digital Converter (ADC) is a very useful feature that converts an analog voltage on a pin to a digital number. By converting from the analog world to the digital world, we can begin to use electronics to interface to the analog world around us.

STM32’s ADC converter is a complete beast. Some of the features it has are scan mode, self-calibration, channel by channel programmable sampling time, discontinuous mode to name a few.

This library however provides two features only, single conversion mode and continuous mode (I might extend this library to include all ADC features in future, depending on overall feedbacks).

Also note that Only ADC 1 is supported. Fixed Resolution set to 12-bits. After every conversion, conversion result is stored in a variable *conversionResult*. Read the result from this variable.

Following are the functions provided by library.

1. **int ADC\_Init(short mode, short alignment)**  //return 0 if unsuccessful, return 1 for success.

**Description:**

Call this function before using ADC 1. You don't need to configure the gpios, this function does that for you.

* Only first 5 [0-5] channels are supported.
* Regular channels are supported only, no Injected channels.
* Only First Five External Channels (Channel 0-4) is supported.
* Use PA0, PA1, PA2, PA3, PA4 for channels 0, 1, 2, 3, 4 respectively. Only one channel can be configured and used at a time.
* After each conversion, read the result from *conversionResult* variable

**Parameters:**

-> **mode** = 0,1

0: Single Conversion Mode (For Regular channel only)

1: Continous Mode (For Regular Channel Only)

-> **alignment** = 0,1

0: right

1: left

1. **void ADC\_ChannelConfig(short channelNumber, short sampleTime);**

**Description:**

Call this function after Init to configure the channels.

**Parameters:**

-> **channelNumber** = 0,1,2,3,4 (The channel you want to configure)

-> **sampleTime** = 0,1,2,3,4,5,6,7

0: 1.5 cycles

1: 7.5 cycles

2: 13.5 cycles

3: 28.5 cycles

4: 41.5 cycles

5: 55.5 cycles

6: 71.5 cycles

7: 239.5 cycles

1. **void ADC\_StartConversion()**

**Description:**

After calling this function, read the result from variable *conversionResult*.

Here’s an example demonstrating ADC.