Embedded Real Time Systems

Design and Implementation of a Clock-Counter system using a RTOS.

Marios Mitalidis
mmitalidis@gmail.com

30/01/2017

Contents

Application Requirements

The following system regards the implementation of a clock/counter. The system should run a RTOS and the application should have minimum size in bytes. More specifically, from the problem description, we have the following requirements:

The system should print the current time with HH:MM:SS format (24-hour) or the current counter time.

It should have as input the following commands:

- R : reset counter to zero
- S : start, stop, (re-)start and stop the counter
- P : pause, print only the current counter value, or continue, (the counter should continue to count during the the display of the value)
- T : toggle, change appearance from clock to counter
- H : increment time by 1 hour, only for the clock mode
- M : increment time by 1 minute, only for the clock mode
- Z : reset seconds counter, only for the clock mode
- Changing from clock to counter should preserve the state of each application.

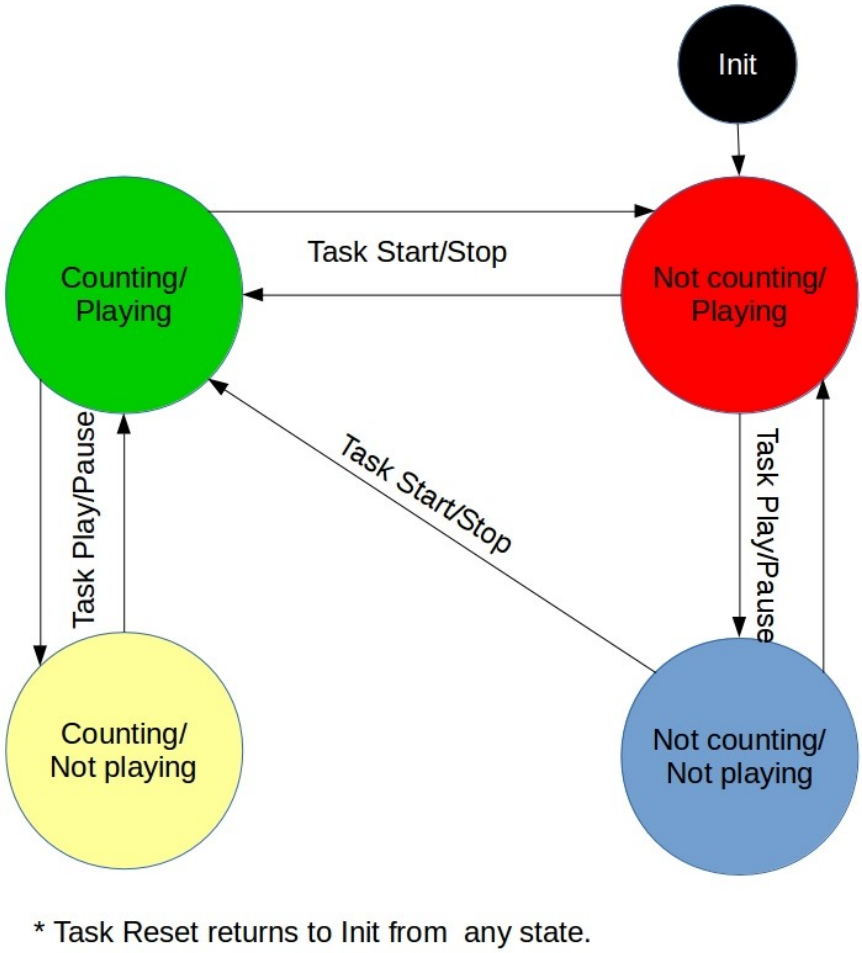
High level system description

The system consists of two applications, the clock and the counter, which run parallel and are independent from each other. We use two counters, where each of them determines the time step for each application.

The clock starts counting from t=0. Using the appropriate inputs we can change the clock time, as described on "Application Requirements".

For the counter application we should note, that we define two variables "*time*" and "*print_time*". The "*time*" variable is incremented for each counter time step, while the "*print_time*" variable stores the value that should printed and has the same value as the "*time*" variable, unless the user has pressed *pause*.

Moreover, we define the states *"Counting/Not counting"* and *"Playing/Not playing"*. The case *"Counting/Not counting"* determines if the counter is used (update its state). On the other hand, the *"Playing/Not playing"* state, determines if the "*print_time*" variable, should be updated.



Drawing 1: System States

The above diagram shows the states transitions, for each task.

Hardware

The system was developed using STM32F401RE Nucleo development board. The microcontroller includes the *ARM 32-bit Cortex-M4* central processing unit with 512KB Flash/96KB SRAM, along with a lot of peripherals for interconnection with other systems.

The advantages of this development board are the following:

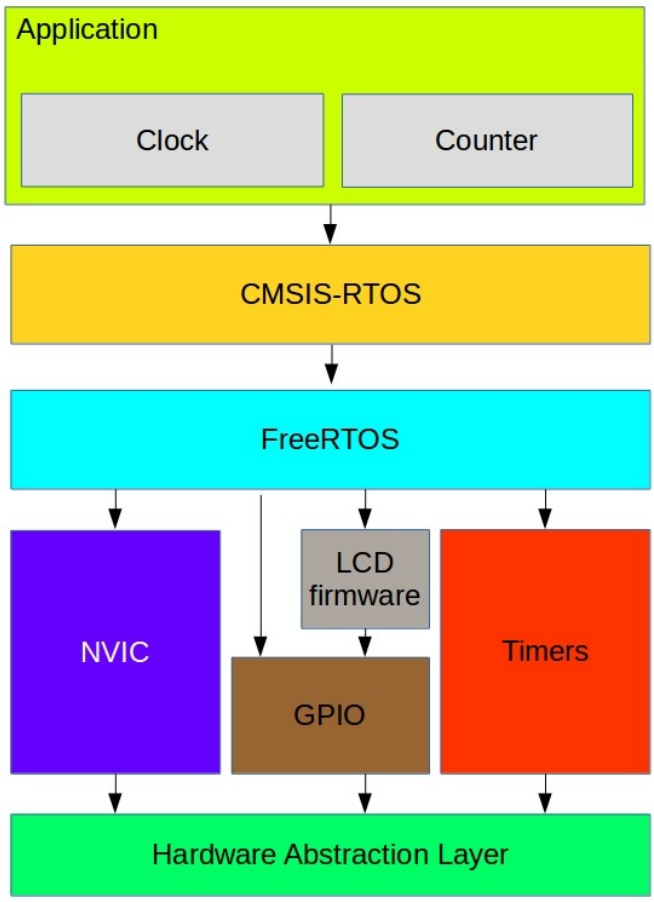
1. small development cost
2. a powerful IDE which is helpful for a fast prototype implementation
3. it is educational, since ARM processors are important in the embedded systems industry
4. small power consumption

Furthermore, an LCD monitor of type HD44780 was used.

Software

Software layers description

The structure of the system is presented on the following picture. We further describe each layer.



Drawing 2: Software Layers

Hardware Abstraction Layer

Starting the software description from the bottom layer, we used the ST Hardware Abstraction Layer. This part of the code is a clean interface for the connection with the hardware.

The specific HAL gives designers the flexibility to use another ST microcontroller of the stm32f4xx series (where xx denotes the model), without changing the application code.

As a drawback we should note that using the HAL, we make the executable bigger in size and the execution time increases, due to a few more function calls.

Nested Vector Interrupt Controller

Here we set up which interrupts (such as timers or externals) will be activated so that ISR table is generated. During the code development phase the developer writes the callback functions that get executed. They are mostly signals to OS threads.

General Purpose Input Output

It refers to the pin state and the interrupt routines for input/output. More specifically, using GPIO pins we get user commands from the push buttons. Moreover, they are used for printing results to the LCD monitor.

Timers

Timers are an import part of the system. After, pre-defined time intervals they activate interrupts regarding:

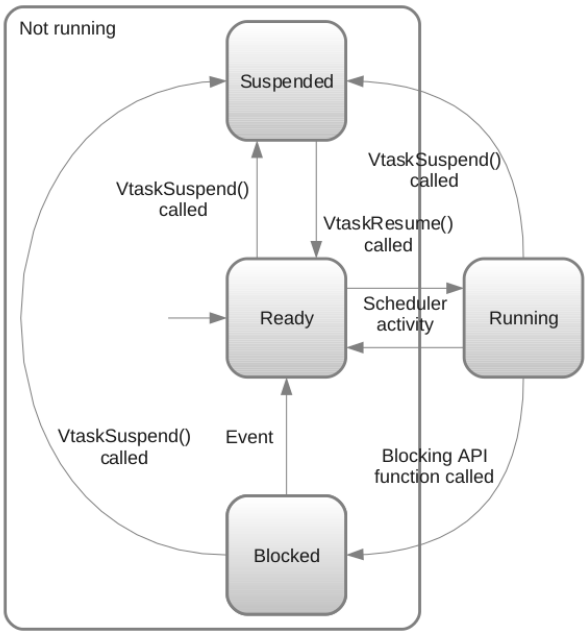
1. Clock tick for application “clock” (and increment of the corresponding counter) each second.
2. Clock tick for application “counter” (and increment of corresponding counter – change of “print_time”) each 1ms.
3. Clock tick for execution of the next LCD task.
4. Clock tick for refreshing of the LCD print message.

The values of the last two timers were found experimentally, so that it neither the refresh rate value is too big, resulting in a monitor that does not display any value, nor too small giving the sense of a slow response time.

FreeRTOS

FreeRTOS is a minimalistic RTOS, free, open-source, developed by Real Time Engineers Ltd. It offers a single kernel with a kernel with basic functionality for job scheduling and memory management, while it does not include drivers for communication with other devices.

In the following diagram the states in which a process can be found and the corresponding transitions are presented.



Drawing 3: Task states and transitions

Assuming that the system has a single processor the scheduler is responsible to choose which (from the pool of "Ready") task should be executed the next time slot. It uses a system timer, and at each clock tick it chooses the task with the highest priority. When there are multiple tasks with the same priority the scheduler alternates between them on every clock tick.

It is worth to mention that the tasks' priorities are defined statically and do not change permanently, unless the developer explicitly decides to do so.

Moreover, there is no provision from the kernel for low priority tasks starvation.

A FreeRTOS capability that was used for accessing shared memory was mutexes. In that case, using priority inheritance, the task that uses a shared resource inherits the greatest priority of all the tasks waiting for that resource.

Finally, queues for communication and synchronization are supported, as well as semaphores and dynamic memory management.

CMSIS-RTOS

CMSIS stands for Cortex Microcontroller System Interface Standard. It is a manufacturer independent HAL, for Cortex M microcontrollers. Thanks to this API, the function calls and function definitions of the OS, ore specifically tasks and signals, are RTOS independent and makes the development process easier and less error prone.

Application

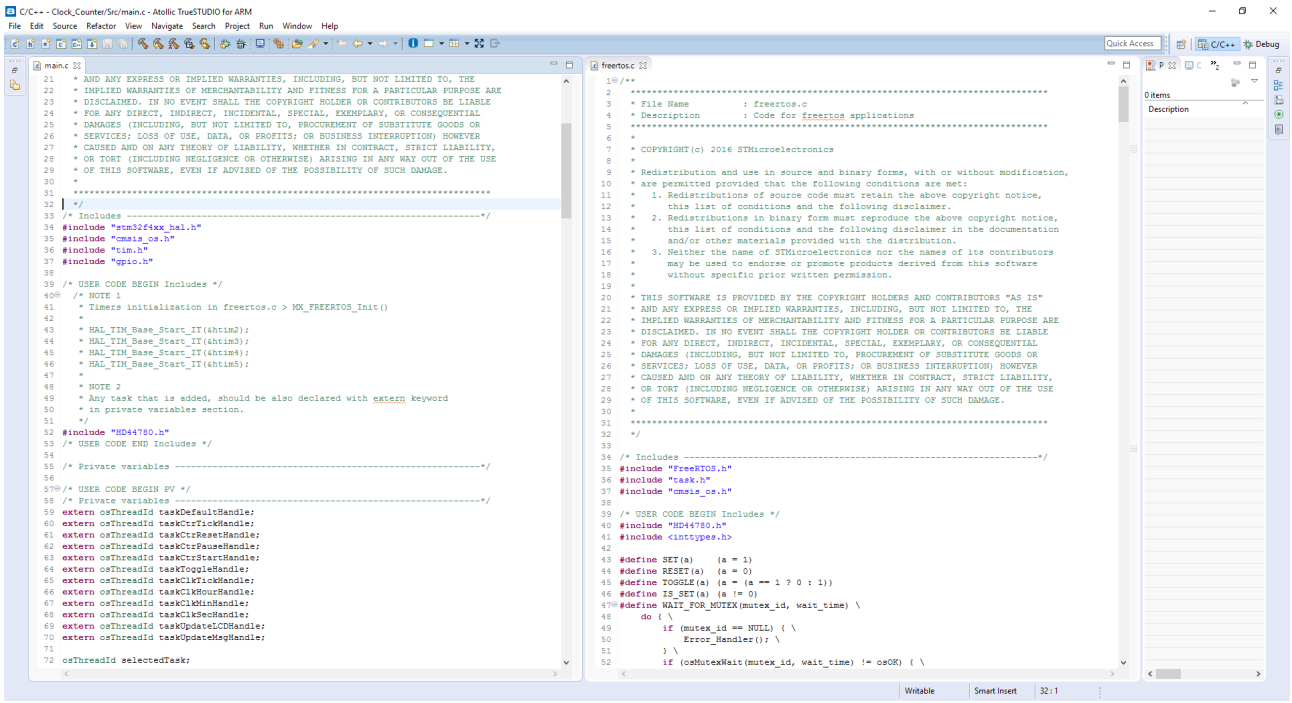
This layer is about the implementation of the embedded system functionality. It consists of two basic parts, the clock and the counter, that are described in a preceding section.

System Development

Integrated Desktop Environment

For the development of the system we used ST tools. More specifically, the company supports some toolchains and provides a C code-generation system. The software generated is responsible for the initialization and interconnection of the microcontroller peripherals, while the developer focuses on the application development.

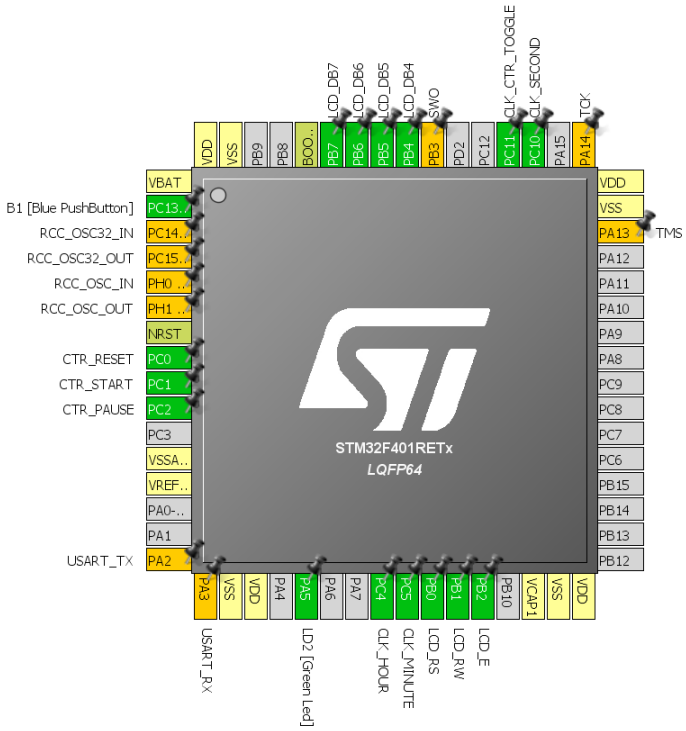
The IDE that was used was TrueSTUDIO from Atollic [1], which uses toolchain *arm-atollic-gnu* for the compilation. It is a product based on eclipse which can be used to program the microcontroller and for debugging.



Drawing 4: Atollic TrueSTUDIO IDE

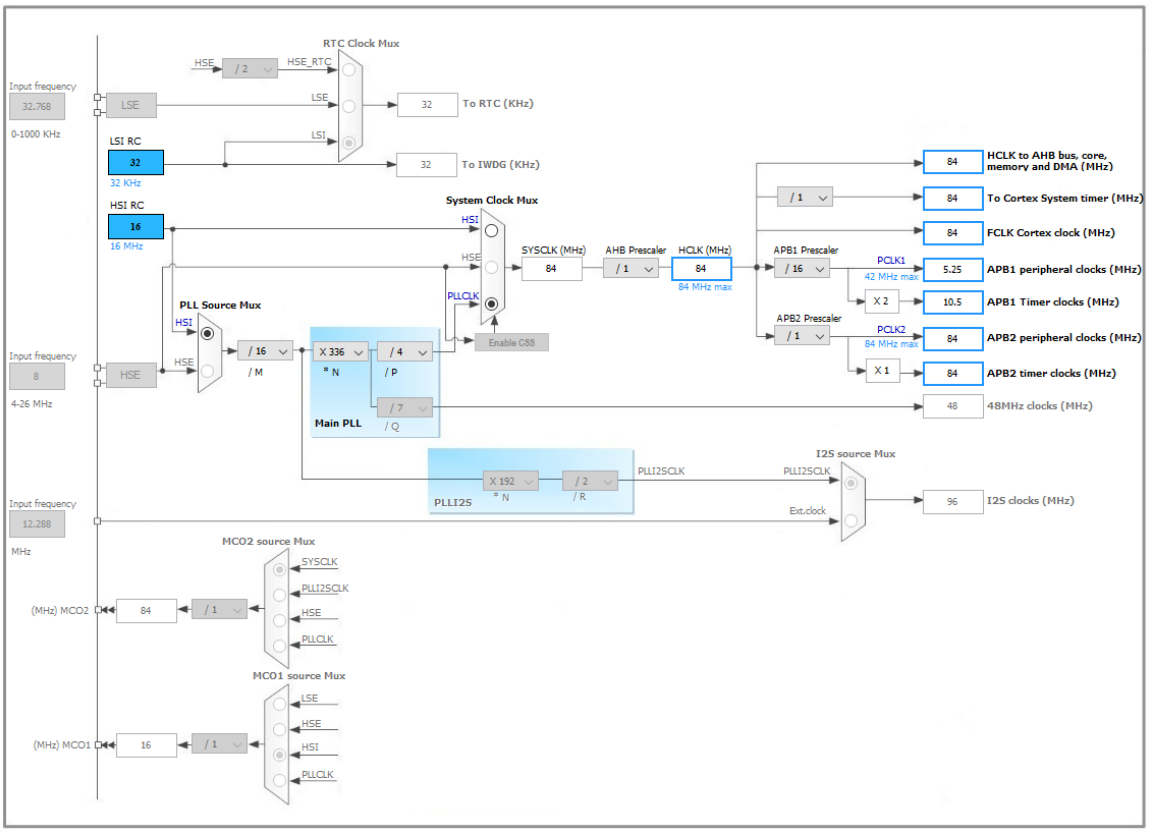
The software for C code generation is STM32CubeMX [2]. It is a very useful application that helps to develop the system fast and accurately.

First of all, we choose the microcontroller or the development board that is going to be used. The using the *Pins* tab the functionality of the microcontroller's pins is defined.



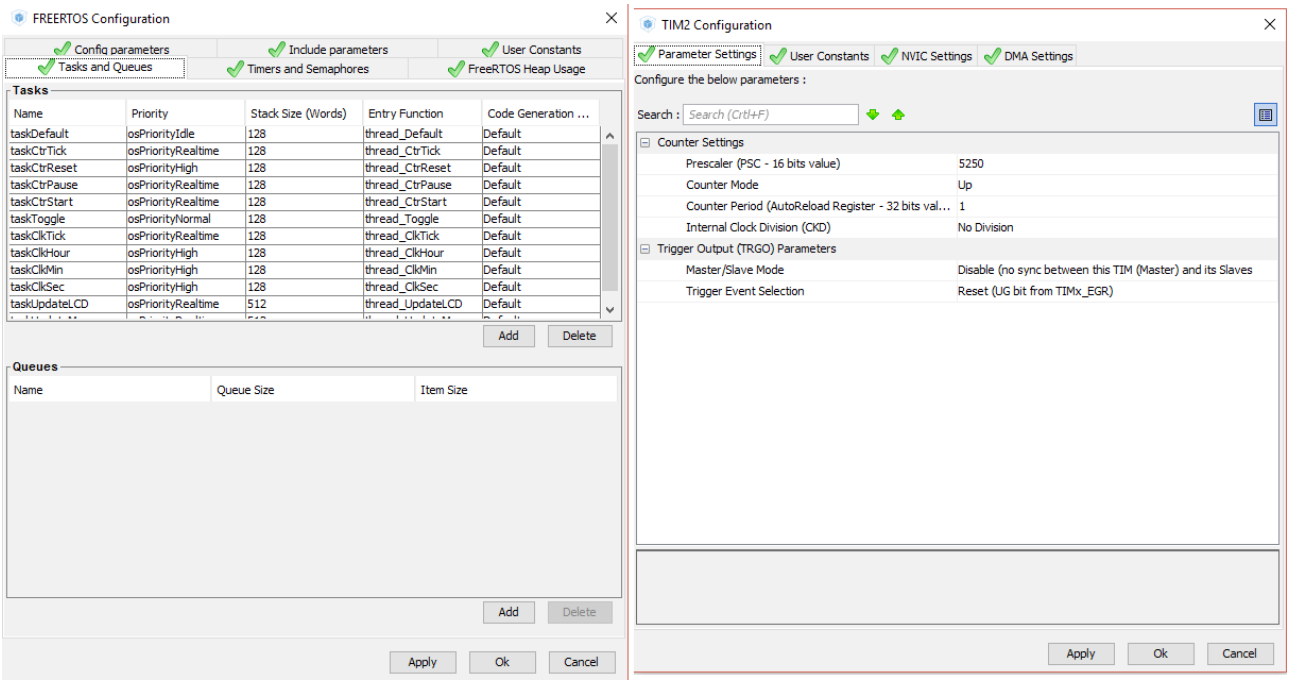
Drawing 5: Microcontroller Pins

Obviously, the *Clock* tab presents with a graph the production and distribution of the clock timers, along with options for various prescalers.



Drawing 6: Microcontroller Clock tab

Moreover, from the *Configuration* tab, we choose the parameters of the supported components that will be used. We present some the options for the RTOS and the timer-2.



Drawing 7: RTOS - timer 2 options

Finally, using the *Power Consumption* tab, we can predict the power consumption of the system.

Problems and Solutions

Deadlocks between threads.

Problem
During the development of the system more than one shared variables were used. Some threads were found in a deadlock state, as one of them holded a mutex for a control variable, while trying to acquire mutex for a data variable.

Solution
As soon as a thread uses a shared resource it releases the corresponding mutex, before asking for another.

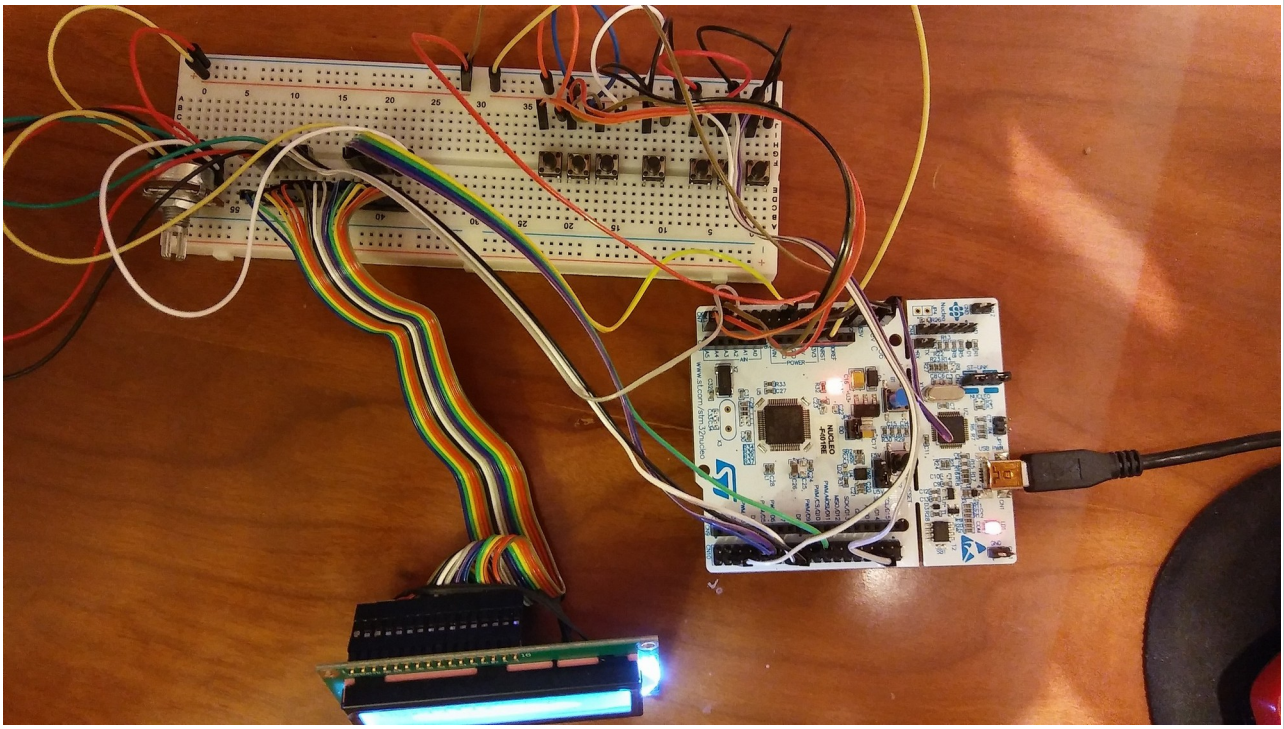
Connecting switches to the development board.

Problem
For the user input, we connected push buttons to the GPIO pins of the microcontroller. Thus, using interrupts the corresponding callback function is executed. The problem was that pressing a push button once, triggered multiple interrupts, possibly due to high variation of the input voltage.

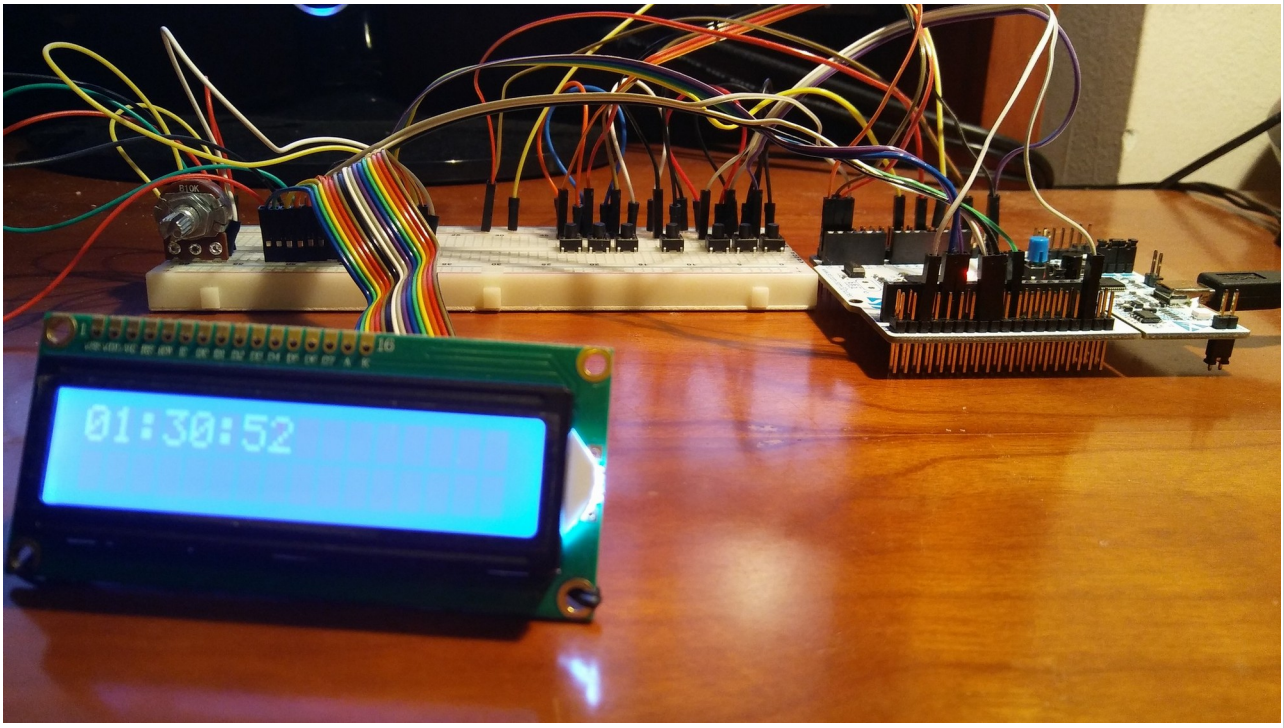
Solution
A solution would be to use a low-pass filter to cut off the high frequency part of the input signal. As an immediate solution, we chose to define that pressing a button (for functionality A) changes the functionality of the on-board blue button (to functionality A).

System prototype presentation

In the following pictures the prototype is presented.



Drawing 8: Prototype - top view



Drawing 9: Prototype - front view

On the following link [3] a video demo is presented on YouTube, where all the operations are executed.

System Efficiency

In an effort to design an efficient system, the priorities of the tasks were chosen such that the user gets the most accurate result. The priorities of the tasks are presented on the following table.

Task	Priority
taskDefault	osPriorityIdle
taskCtrTick	osPriorityRealTime
taskCtrReset	osPriorityHigh
taskCtrPause	osPriorityRealTime
taskCtrStart	osPriorityRealTime
taskToggle	osPriorityNormal
taskClkTick	osPriorityRealTime
taskClkHour	osPriorityHigh
taskClkMin	osPriorityHigh
taskClkZero	osPriorityHigh
taskUpdateLCD	osPriorityRealTime
taskUpdateMsg	osPriorityRealTime

As a first test, the response of the system was checked, as well as the inaccuracy of the measured values was measured.

The system has been developed so that each task executes its functionality and then gets in sleep mode, after calling the function osSignalWait(SIGNAL_WAKE_UP, osWaitForever). The interrupts (external or timers) send a signal to the corresponding task in order to start its operation. As long as none of the tasks is executed the scheduler chooses only the defaultTask, that is the idle state, where the power consumption is low.

A more holistic approach, would require profiling from the RTOS and data output through the debugging port. FreeRTOS has this capability, if a separate timer is used, which periodically increases a time variable, so that the CPU usage of each task can be calculated.

Using this MCU usage profile it is possible to predict the power consumption. More specifically, giving as input to the STM32CubeMX tool a periodic sequence of CPU and peripherals usage, a power consumption profile can be calculated, hence predict the expected lifetime of the power source.

The requirement for a minimum binary size is achieved, using the *"-Os"* during compilation. The binary size is: 36800 bytes.

Further Research

Task Profiling and Power Consumption calculation

The first step is to measure the CPU usage time and the power consumption. Thus, improvements could be made and a suitable battery for the system would be chosen.

Real Time Clock

The microcontroller datasheet reveals that the clock is a RC oscillator (which is input to a PLL), that diverges by 1% at 25℃. It would be interesting to connect a high precision real time clock and/or use the external dev-board oscillator, so that the system efficiency is compared for each case.

Test cases development

This part refers to the development of test cases to find bugs in the systems and confirm works as expected.

PCB Manufacturing

PCB manufacturing of the system would reveal some other problems, while we would have to take into consideration parameters concerning the size, durability electromagnetic compliance and more.

Cost Estimation

Finally, it would be interesting to make an initial cost prediction for production of the system.