

Developer Guide

UNSW DESN2000 24T2 (COMP)

Group: G

Members: Aditya Muthukattu, Linfeng Cai, Kavisha Chandraratne

Table of Contents

- [Developer Guide](#)
 - [Table of Contents](#)
 - [Introduction](#)
 - [Development Environment](#)
 - [Hardware Requirements](#)
 - [Software Requirements](#)
 - [Code Structure](#)
 - [System Architecture](#)
 - [Key Features](#)
 - [User Interface](#)
 - [State Management](#)
 - [Libraries](#)
 - [HAL](#)
 - [State](#)
 - [LCD](#)
 - [Code structure](#)
 - [Initialisation](#)
 - [LCD Commands and Data Functions](#)
 - [Display String and Cursor Functions](#)
 - [Helper Functions](#)
 - [LED Bar](#)
 - [Initialisation](#)
 - [LEDBar Load](#)
 - [Keypad](#)
 - [Initialization](#)
 - [Keypad Input Functions](#)
 - [Standard Clock and Alarm](#)
 - [Code Structure](#)
 - [Initialization](#)
 - [STD_CLK_Init\(\)](#)
 - [Primary Logic Handling](#)
 - [STD_CLK_Update\(\)](#)
 - [STD_CLK_SetTimeAndDate\(\)](#)
 - [STD_CLK_SyncTime\(\)](#)
 - [STD_CLK_DisplayCurrentTime\(\)](#)
 - [Alarm Handling](#)
 - [STD_CLK_AlarmEventHandle\(\)](#)
 - [STD_CLK_AlarmDeactivateHandle\(\)](#)
 - [STD_CLK_SetAlarm\(\)](#)
 - [Helper Functions](#)
 - [STD_CLK_GetWeekday\(\)](#)
 - [getWeekday\(\)](#)
 - [Extra content \(External script\)](#)
 - [Standard timer](#)

- [Structure Overview](#)
- [Stopwatch](#)
- [Extending the Code](#)

Introduction

This document provides a comprehensive guide for developers who wish to contribute to the Custom Clock with Laboratory Timer (CCLT) project. The CCLT is a versatile device that integrates parallel timer functionality with standard clock features, as well as additional features tailored specifically to enhance the efficiency and accuracy of lab operations. This guide will provide an overview of the project structure, the development environment, and the process for contributing to the project.

Development Environment

Hardware Requirements

The CCLT project is designed to run on the STM32F303RET6 microcontroller on the NUCLEO-F303RE development board. In addition to this you will need the UNSW coaST education board.

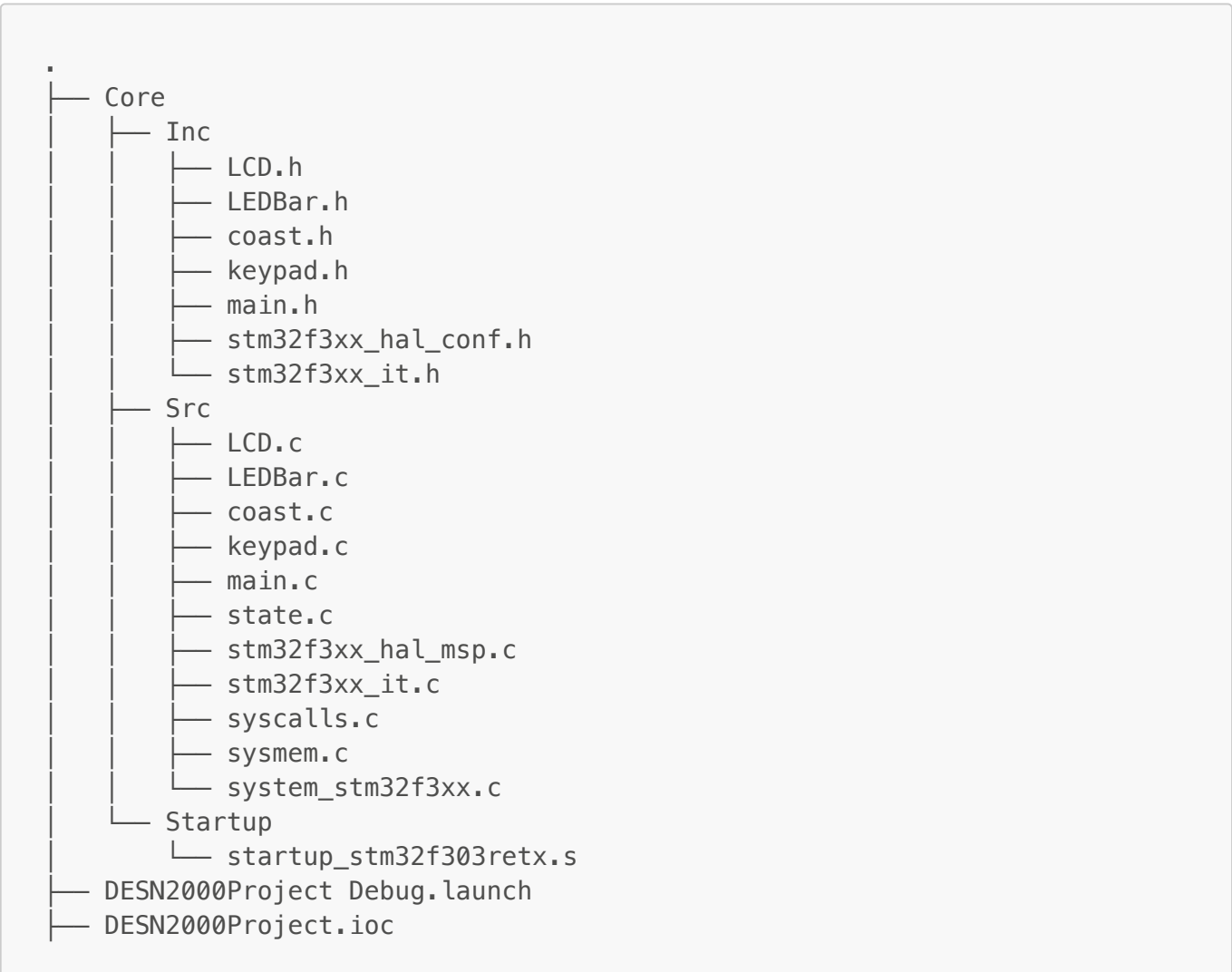
Software Requirements


This project has been developed entirely on the STM32CubeIDE platform.

Code Structure

[Describe the overall structure of the codebase, including the main modules, classes, or components used.]

The CCLT project is structured as follows:



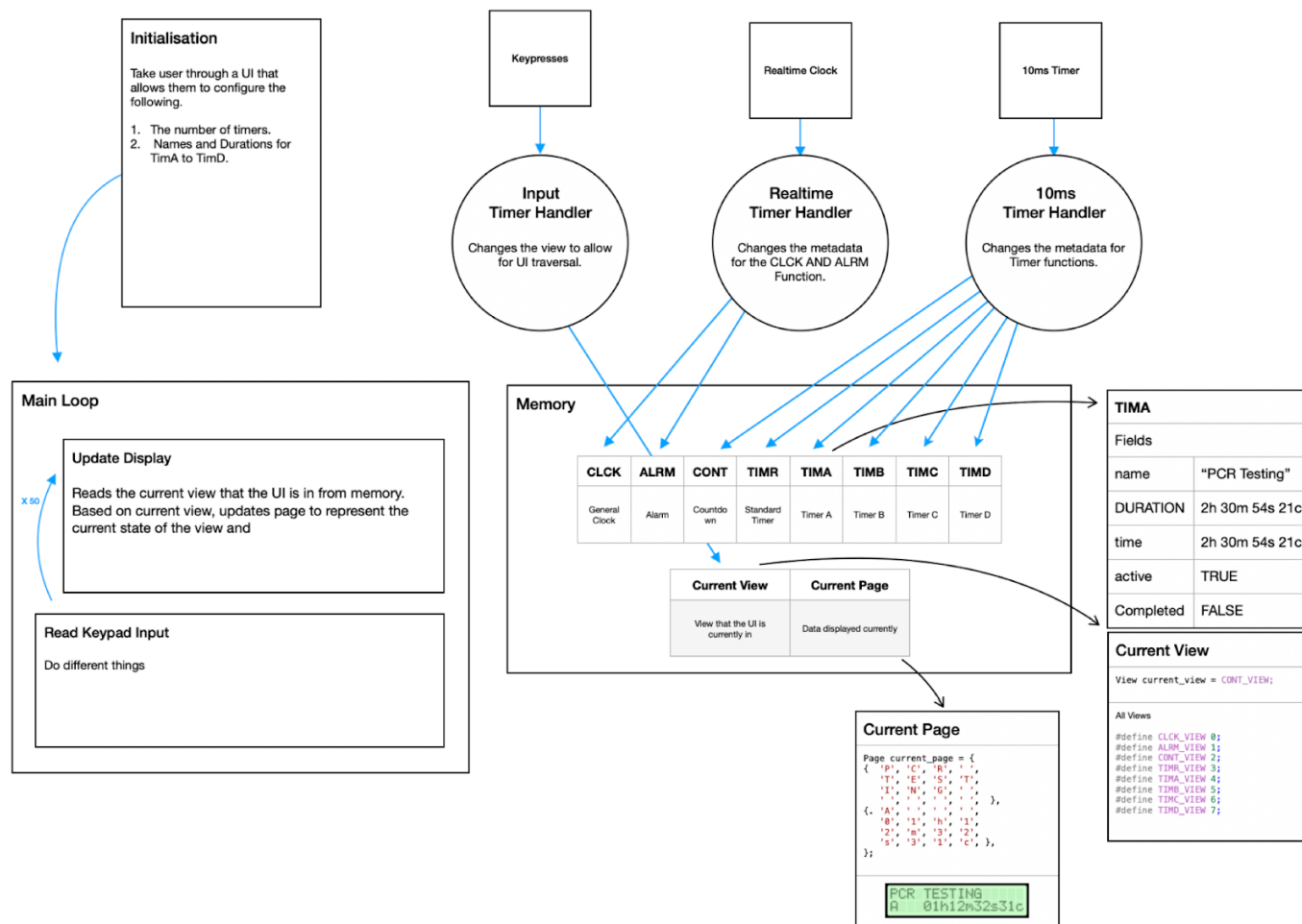
 Debug
...

The project is divided into several modules, each responsible for a specific aspect of the device's functionality. The main modules can be found in the [Core/Src/](#) directory, and accompanying header files are found in the [Core/Inc](#).

The main modules of the implementation are:

- [main.c](#): The main entry point of the program, responsible for initialising the system and running the main loop.
- [state.c](#): Contains functions that enable our finite state machine which manages the state of the device.
- [memory.c](#): Contains functions and data structures to store data
- [lcd.c](#): Contains functions to interface with the LCD display
- [LEDBar.c](#): Contains functions to interface with the LED bar
- [keypad.c](#): Contains functions to interface with the keypad
- [STD_CLK.c](#): Contains the implementation of the standard clock functionality

System Architecture



TODO: Add figure caption in Markdown

Key Features

The above diagram illustrates the design of the system.

Once the device is powered on it will display the boot screen with device information. It will be followed by the initialisation process. During the initialisation subroutine the user can set the number of timers in the scientific mode and set the labels and durations for each of the timers.

Once initialised the UI will proceed to the regular UI mode where the user can switch between scientific and standard modes, and cycle through their individual functions. Once traversed into the Scientific mode the user can select through the 4 (or fewer) timers by using the up and down arrow keys. The user can also start a specific timer when inside each of the timers by using the key and can reset the timer using # to its duration.

The user can change the labels and durations for each of the timers by traversing further by using the "SET" key (c on the keypad). However the duration cannot be edited while the timer is active.

Inside the Standard mode the user can traverse through four functions. The clock, stopwatch, alarm and standard timer. They can be initiated in a similar fashion to the scientific timers.

The implementation has a global object, Memory, which stores the current view of the UI, the current page loaded on the display and the metadata for each of the functions such as active status, if completed, duration and the current time.

User Interface

User interface implementation is done mainly via the main loop. This design decision coincides with the button layout for the implementation as all UI traversal buttons are assigned to the keypad. Inside the main loop there exists a state check which displays the corresponding view that the UI is currently in. The loop then proceeds to listen for button inputs and changes memory accordingly.

Pressing the MODE button changes the `current_view` state in memory. During the next iteration of the loop it recognises the updates view and generates a page that reflects the data for that function.

The `smart_load` function moves the cursor of the LCD display and only updates locations where characters will change. This is implemented by storing the last page to be displayed on the display in memory. This implementation methodology has significantly reduced the amount of flickering that otherwise is usually visible on the display.

An extension of the UI is when data is to be changed by pressing the SET button. Pressing SET and RESET on each function runs a unique subroutine which allows the user to configure labels and/or durations.

State Management

The state management logic located in `state.c` is responsible for managing the state of the device. The state machine is implemented using a finite state machine (FSM) approach, where each state represents a different mode of operation for the device. The state machine transitions between states based on user input and system events. See the documentation in `state.c` for more information on the state machine implementation.

Libraries

HAL

The Hardware Abstraction Layer (HAL) library provides a higher level interface to the STM32 and its functionality. It simplifies the process of configuring and using the microcontroller peripherals, allowing developers to focus on their implementation rather than the low-level hardware details.

State

`state.c` contains the implementation of the state machine that manages the state of the device. The FSM is implemented using an enum of states, an enum of events, and a transition table that maps states and events to the next state. The `handleEvent` function is responsible for processing events and transitioning between states based on the current state and event.

Enums

```
typedef enum {  
    STATE_STD_CLK,  
    STATE_STD_ALM,
```

```
    STATE_STD_CTD,  
    STATE_STD_STP,  
    STATE_SCI_T1,  
    STATE_SCI_T2,  
    STATE_SCI_T3,  
    STATE_SCI_T4,  
    STATE_EDIT  
} State;
```

```
typedef enum {  
    EVENT_MODE,  
    EVENT_FUNC,  
    EVENT_SET,  
    EVENT_RESET,  
} Event;
```

The **Transition** struct defines a state transition, including the current state, the event that triggers the transition, and the next state.

```
typedef struct {  
    State fromState;  
    Event event;  
    State nextState;  
} Transition;
```

The **transitions** table defines the state transitions for the FSM. Each element in the array represents a transition from one state to another based on a specific event.

LCD

This library aims to bring developer both convenience and flexibility by having both a lower-level bit manipulation functions as well as higher level helper functions that allows for quick and easy development for it. For instance, if the developer does not want to pay too much attention to how to initialise and send instruction to the LCD, they can simply use our pre-made Init function and then use LCD_PrintLine1 and LCD_PrintLine2 to display any content on the screen.

On the other hand, developer who wants to customize their own initialisation for their own reason be it compatibility or optimisation reasons can modify the initialise function as well as using the lower-level functions such as PutNibble and Pulse to create their own sequence of instructions.

Code structure

This library is consisting of 3 main parts. One single large Initialisation function that setup the data pins and send the required pulse to start the LCD screen. It is then followed by LCD commands and send data function that handles the lower-level operation of the library allows developer to tamper with the lower-level operation with less difficulty. Then it is the highest level of helper function that makes use of the said lower-level functions to complete some of the commonly used tasks for the convenience of developer.

Initialisation

LCD_init()

The `LCD_init()` function initializes the LCD display. It configures the GPIO pins connected to the LCD and sets the display to 4-bit mode. It then sends a series of commands to set up the display. Detailed steps below.

Steps:

- Enable the clock for the GPIO port connected to the LCD.
- Initialize the data pins (D4, D5, D6, D7) as output.
- Initialize the control pins (E, RW, RS) as output.
- Wait for the LCD to stabilize.
- Send the initialization sequence to set the LCD to 4-bit mode.
- Configure the LCD with function set, display control, clear display, and entry mode set commands.

LCD Commands and Data Functions

LCD_PutNibble(uint8_t nibble)

This function sends a 4-bit nibble to the LCD data pins.

LCD_Pulse()

This function sends an enable pulse to the LCD, signaling it to read the data/command on the data lines.

```
LCD_SendCmd(uint8_t cmd)
```

This function sends a command to the LCD.

```
LCD_SendData(uint8_t data)
```

This function sends data to the LCD.

Display String and Cursor Functions

```
LCD_DisplayString(char* str)
```

This function sends a string to the LCD, character by character.

```
LCD_SetCursor(uint8_t row, uint8_t col)
```

This function sets the cursor to a specific position on the LCD.

Helper Functions

```
LCD_PrintLine1(char* str)
```

This function prints a string on the first line of the LCD.

```
LCD_PrintLine2(char* str)
```

This function prints a string on the second line of the LCD.

LED Bar

The `LEDBar.c` file contains functions to initialise and control an LED bar using shift registers.

Initialisation

```
void LEDBar_Init(void);
```

The `LEDBar_Init()` function initialises the GPIO pins used to control the shift registers connected to the LED bar. The following GPIO pins are used to control the shift registers:

- `SR_RCLK_Pin`: Register Clock Pin
- `SR_SRCLK_Pin`: Shift Register Clock Pin
- `SR_SER_Pin`: Serial Data Input Pin

Usage: Call this function once during the system initialisation to set up the GPIO pins for the shift registers.

```
LEDBar_Init();
```

LEDBar Load

```
void LEDBar_Load(uint16_t data);
```

The `LEDBar_Load` function loads a 16-bit data value into the shift registers, which in turn controls the LEDs on the LED bar. It shifts the data bit by bit into the shift register and triggers the clock and latch pins to update the LED states.

Usage: Call this function to update the LED bar with a new pattern.

```
uint16_t led_pattern = 0xFFFF; // Example pattern  
  
LEDBar_Load(led_pattern);
```

Keypad

This library aims to bring developers both convenience and flexibility by providing both lower-level bit manipulation functions and higher-level helper functions that allow for quick and easy development. For instance, if the developer does not want to focus too much on how to initialize and read input from the keypad, they can simply use our pre-made initialization function and then use the provided functions to read keypresses.

On the other hand, developers who want to customize their own initialization for compatibility or optimization reasons can modify the initialization function and use the lower-level functions to create their own sequence of instructions.

This library consists of two main parts:

1. An initialization function that sets up the row and column pins.
2. A function to read keypresses from the keypad.

Initialization

`Keypad_Init()`

The `Keypad_Init()` function initializes the keypad. It configures the GPIO pins connected to the keypad rows as outputs and the columns as inputs with pull-up resistors.

Steps:

Initialize the row pins (ROW1, ROW2, ROW3, ROW4) as output.

Initialize the column pins (COL1, COL2, COL3, COL4) as input with pull-up resistors.

Keypad Input Functions

Keypad_GetKey()

The `Keypad_GetKey()` function reads the keypad input and returns the corresponding character. It includes debouncing and registers the key press on release.

Steps:

- Define a keymap matrix to map the row and column indices to their respective characters.
- Iterate through each row, setting it to low and checking the state of each column.
- If a column pin is low, wait until it is released (debounce).
- Return the character corresponding to the pressed key.
- If no key is pressed, return '\0'.

Standard Clock and Alarm

This library provides a comprehensive solution for handling a standard clock and alarm system on an STM32 microcontroller. It offers both convenience and flexibility by allowing developers to use higher-level functions for quick setup and operation, while also providing the ability to customize alarm handling and other functions to fit specific requirements.

For instance, developers can use the provided initialization and display functions for a straightforward implementation. However, those who need more control can modify the initialization process and customize the alarm handling logic to integrate with other system components or specific application needs.

Code Structure

This library consists of several main parts:

- Initialisation function to set up the clock and alarm system.
- Primary logic handling for updating the display and setting the clock or alarm.
- Functions to set time and date manually or via serial communication.
- Functions to display the current time on the LCD.
- Customisable alarm handling functions.
- Helper functions for date and time calculations.

Initialization

STD_CLK_Init()

The `STD_CLK_Init()` function initialises the standard clock system. It sets up the RTC and UART handlers, initialises the LCD, and clears the display.

Steps:

1. Assign the RTC and UART handlers.
2. Initialize the LCD.

Clear the LCD display and return the cursor home.

Primary Logic Handling

STD_CLK_Update()

The `STD_CLK_Update()` function handles the main logic for updating the display and setting the clock or alarm based on keypad input.

Steps:

1. Read key input from the keypad.
2. Depending on the current state (setting date, time, or alarm), handle the input accordingly.
3. Update the display and internal state based on the input.
4. Setting Time and Date

STD_CLK_SetTimeAndDate()

The `STD_CLK_SetTimeAndDate()` function sets the time and date in the RTC.

Steps:

1. Parse the input values for hour, minute, second, day, month, and year.
2. Set the RTC time and date using the parsed values.

STD_CLK_SyncTime()

The `STD_CLK_SyncTime()` function uses serial communication to receive a string representing the date and time, and updates the RTC accordingly.

Steps:

1. Continuously attempt to receive data via UART.
2. Parse the received string into date and time components.
3. Update the RTC with the parsed values.
4. Displaying Current Time

STD_CLK_DisplayCurrentTime()

The `STD_CLK_DisplayCurrentTime()` function retrieves the current time and date from the RTC and displays it on the LCD.

Steps:

1. Get the current time and date from the RTC.
2. Format the date and time into strings.
3. Display the formatted strings on the LCD.

Alarm Handling

`STD_CLK_AlarmEventHandle()`

The `STD_CLK_AlarmEventHandle()` function handles the actions to be taken when the alarm goes off. This function can be customized to perform different actions.

`STD_CLK_AlarmDeactivateHandle()`

The `STD_CLK_AlarmDeactivateHandle()` function handles the deactivation of the alarm. It turns off the alarm in the RTC and any other indicators like LEDs.

`STD_CLK_SetAlarm()`

The `STD_CLK_SetAlarm()` function sets the alarm in the RTC.

Steps:

1. Set the alarm time and day.
2. Activate the alarm with an interrupt.

Helper Functions

`STD_CLK_GetWeekday()`

The `STD_CLK_GetWeekday()` function calculates the day of the week based on the date using Zeller's congruence.

`getWeekday()`

The `getWeekday()` function returns the string representation of the weekday.

Usage Example

Here is a basic example of how to use the Standard Clock and Alarm library in your main code.

```
#include "STD_CLK.h"
```

```
int main(void) {  
  
    HAL_Init();  
  
    RTC_HandleTypeDef hrtc;  
  
    UART_HandleTypeDef huart;  
  
    STD_CLK_Init(&hrtc, &huart);  
  
  
    while (1) {  
  
        STD_CLK_Update();  
  
    }  
  
}
```

Extra content (External script)

For the serial time sync to work a script that continuously send the current time data from external device to serial port is required.

The below is an example of how a python script can be used to achieve that.

```
import serial  
  
import time  
  
def send_time(serial_port, interval=1):  
  
    try:  
  
        ser = serial.Serial(serial_port, 38400) # Adjust the baud rate  
and select a serial port as necessary  
  
        time.sleep(2) # wait for the serial connection to initialize  
  
        while True:  
  
            current_time = time.localtime()  
  
            time_str = time.strftime("%H%M%S%d%m%y", current_time)
```

```
        print(f"Sending time: {time_str}")

        ser.write(time_str.encode())

        time.sleep(interval) # Wait for the specified interval before
        sending the next time

    except serial.SerialException as e:

        print(f"Error opening serial port {serial_port}: {e}")

    except KeyboardInterrupt:

        print("Stopping time sending.")

    finally:

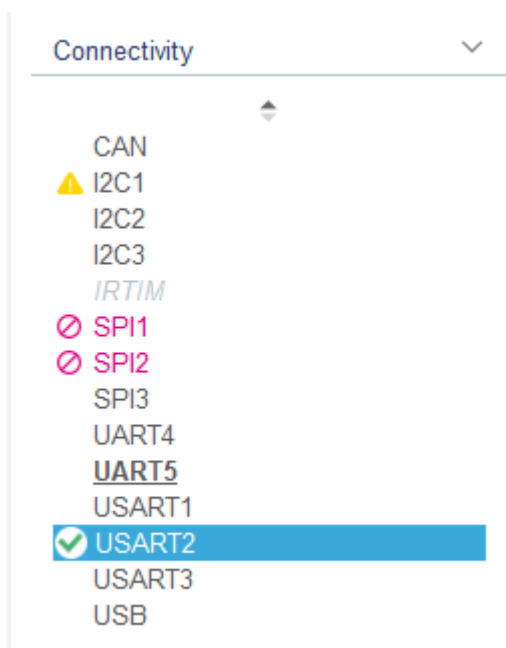
        ser.close()

if __name__ == "__main__":

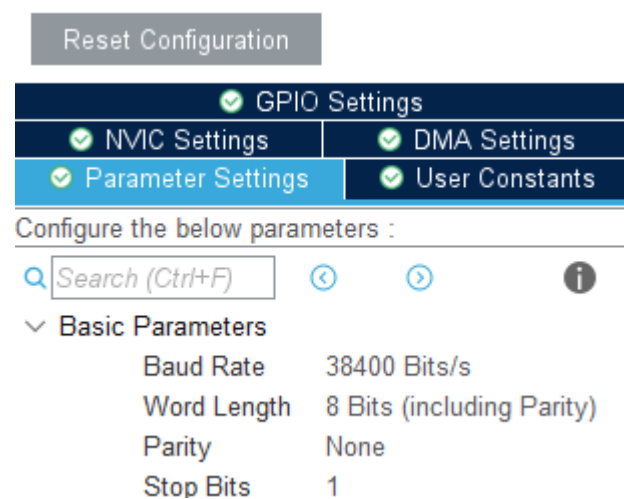
    send_time('COM3', interval=1)
```

For those who want to use the above script for their own project make sure that the baud rate matches the microcontroller setting.

To change the setting for your micro controller find your serial port in connectivity setting as seen in the below picture.



Find your baud rate in the parameter setting as shown below.



Standard timer

Introduction

This guide provides an overview of the timer module, its structure, and guidelines for expanding its functionalities. The timer module is designed to handle a countdown timer that can be set, started, paused, and reset using a keypad. It interfaces with an LCD for display and utilises the RTC for accurate timekeeping.

Structure Overview

The timer module consists of the following files:

timer.h: Header file containing function declarations and necessary includes.

timer.c: Source file implementing the timer logic and functionalities.

Function overview for timer.c

```
*TIMER_Init(RTC_HandleTypeDef hrtc, UART_HandleTypeDef huart)
```

Initialises the timer with the given RTC and UART handles.

TIMER_Update(void)

Updates the timer logic based on keypad input and controls the countdown if the timer is running.

TIMER_Display(void)

Displays the current timer value on the LCD.

TIMER_Set(int hour, int minute, int second)

Sets the timer value in hours, minutes, and seconds.

TIMER_Start(void)

Starts the timer countdown.

`TIMER_Pause(void)`

Pauses the timer countdown.

`TIMER_Cancel(void)`

Cancels the timer setting process and resets the timer.

Stopwatch

The stopwatch makes use of a 1hz timer to keep track of time since the format of hh:mm:ss determines that the highest precision is just ss. It is handled by a simple handle function that runs from the main loop depending on the current state the user is in. Every key press are also handled within the update function of the stopwatch. Since the stopwatch is relatively simple we just need 2 state running and pauses.

Start is handled with a 'C' button press and pressing 'C' again will enter the pause state. Pressing D will reset the stopwatch back to 00:00:00.

All of the above state key presses will be handled within the main update loop.

Extending the Code

The codebase may be easily extended by adhering to the project structure and patterns already being practised. Every new component/subsystem is implemented in a separated C file, and all its function prototypes, macros and other "definitions" go in to the respective header file.