Lab1: Interactive traffic lights

Teaching Assistant: Quang Le

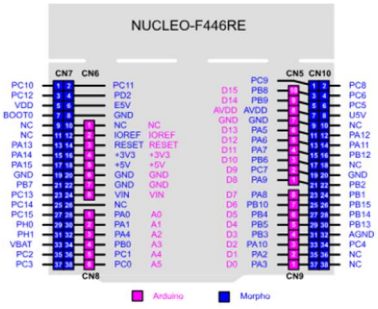
1. Objective

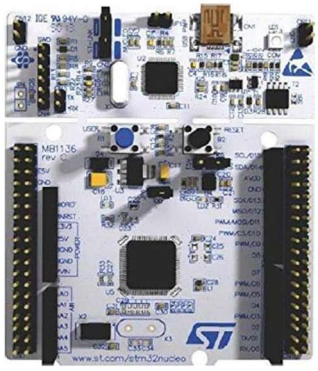
In this lab, students will create different tasks with the **freeRTOS real-time kernel**, using the **STM32CubeIDE** integrated development environment to implement an interactive traffic light system.

2. Development

In this lab, students will use the **nucleo-F446RE development board** shown below. This board is equipped with, among other things, a user **pushbutton** and a reset pushbutton. The nucleo-F446RE board also has **GPIOs to connect LED**s to simulate **interactive traffic lights**. To protect the LEDs, we recommend connecting a **230 to 470 ohm resistor** between the nucleo-f446RE board's GPIO and the LEDs.

Note that this application can be done with a single task, but it is mandatory to use **one task for each LED** and **one for the push button** (for a total of **6 tasks**), in order to get familiar with the **concept of a task**. To do Lab 1, students can use the file **FirstProjectFreeTROS.7z**, which contains a starter project. In this application, three tasks have been created to flash and send a message on the serial port. **USAR2** is used to **send the messages** and **GPIOs PB3, PB4 and PB5** are used to **flash the three LEDs** on the nucleo-F446RE board. The following figure shows the GPIOs on the board.





Part 1

In the first part of the lab, you will turn on three LEDs. They won't all turn on at the same time, of course! In fact, you will build traffic lights.

The control is done in phases. Then, in the second part, you will complete your traffic lights with pedestrian lights. First, let's look at the components list.

Composants nécessaires

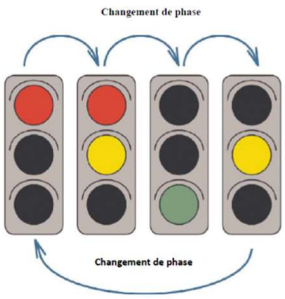
Ce montage nécessite les composants suivants.

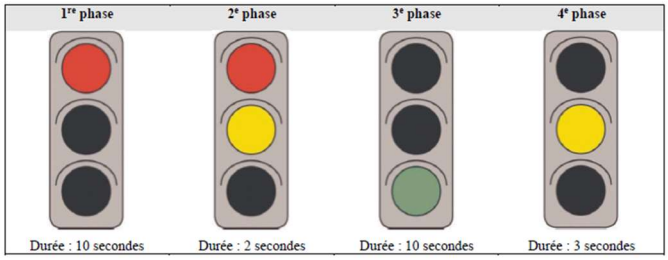
Composant	
2 LED rouges	
1 LED orange	
2 LED vertes	
5 résistances de l'ordre de centaines d'ohms ≈ 330 Ω	
1 bouton-poussoir	Bouton-poussoir bleu de la carte

Signaling Phases

Let's first look at the different possible signaling phases:

The different phases are cycled from left to right, the cycle then starting again at the beginning. Each phase has a defined display duration that can be adjusted individually. Here is an example of lighting durations.

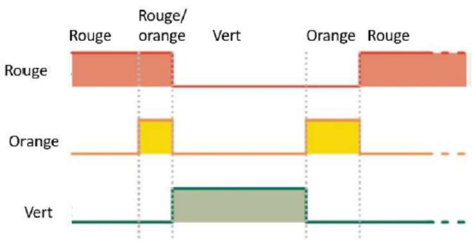


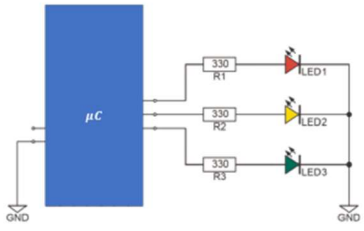


Schematic

The LEDs should be mounted as shown in the following figure. Each LED is driven by a 330 Ω series resistor.

For the traffic lights to work properly, you have to remember not only to turn on the various LEDs, but also to turn them off. The processing is done continuously, from top to bottom, within each task. When moving from phase 1 to phase 2, only one orange LED is added to the red LED. The red one continues to shine. But when moving from phase 2 to phase 3, make sure that the red and orange LEDs go out before the green LED lights up. Then, when moving from phase 4 to phase 1, when the phases start again at the beginning, the orange LED must go out. Take a look at the timing diagram to see how the LEDs are turned on in turn during the different phases.





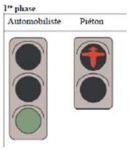
Part 2: Interactive traffic lights

So far, the system has been relatively simple. So you're going to modify it slightly. Let's imagine pedestrian lights installed on a straight stretch of a national road. There's no point in having the phases for motorists constantly changing if no pedestrians want to cross the traffic lane. How should the lights work with their phases? What equipment is needed and how can we extend the logic? Here's what to consider.

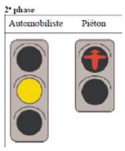
• If no pedestrians come to cross the road, the light remains green for motorists and red for pedestrians.

•If a pedestrian presses the button managing the lights to cross safely, the light changes to orange and then red for motorists. The light then changes to green for pedestrians. After a predefined time, the light changes back to red for pedestrians, and the red light changes to orange and then green for motorists.

The initial situation looks like this:

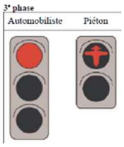


Phase 1: These two light signals remain lit until a pedestrian approaches and presses the button controlling the lights. Only then do the phase changes occur and cause the light to turn red for motorists and green for pedestrians.



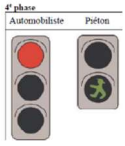
Phase 2: The phase change is triggered by pressing the button that controls the lights. The light turns orange for motorists, which means that it will turn red shortly.

Duration: 3 seconds



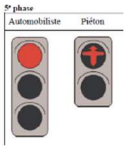
Phase 3: For safety reasons, the light is red first for motorists and for pedestrians. This allows motorists to clear the pedestrian crossing if necessary.

Duration: 1 second



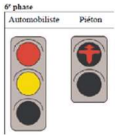
Phase 4: The light turns green for the pedestrian after a short while.

Phase 5: The light turns red for pedestrians.



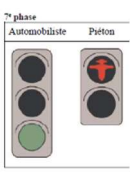
Phase 6: Le feu passe du rouge à l'orange pour les automobilistes, ce qui les avertit que le feu va bientôt passer au vert.

Durée : 2 secondes



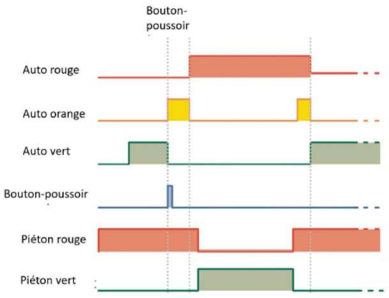
Phase 7: Le feu repasse au vert pour les automobilistes et au rouge pour les piétons. Cette dernière phase est semblable à la première.

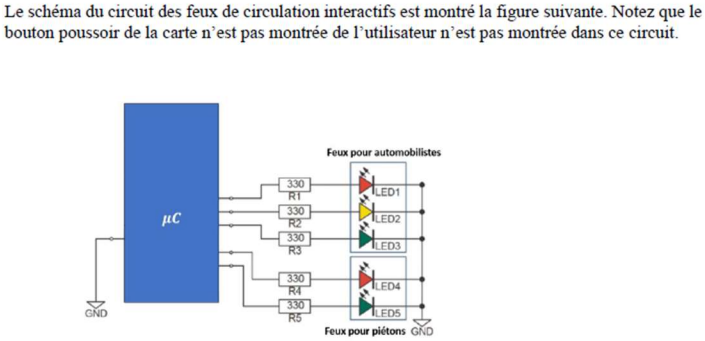
Durée : jusqu'au prochain appui sur le bouton



Schema:

Before we move on to the diagram, here is another timing diagram showing the different lighting durations in relation to each other. The initial situation shows us that the light is green for motorists and red for pedestrians. A pedestrian intending to cross the road at a supposedly safer location presses the button controlling the lights, which initiates the phase changes.





3. Programming

3.1 Software

3.1.1 STM32CubeIDE

STM32CubeIDE is an **advanced C/C++ development platform** for **STM32 microcontrollers** and **microprocessors.**

- Integration: It combines **STM32CubeMX functionalities**, offering an all-in-one tool for device selection, configuration, code generation, compilation, and debugging.
- Development environment: Based on **Eclipse/CDT framework**, it uses **GCC toolchain** for development and GDB for debugging.
- **Multi-OS support**: Available **for Windows, Linux, and Mac operating systems**.
- **Debugging capabilities**: Offers **advanced features** like **CPU core register views**, **memory views**, live variable watch, and **Serial Wire Viewer (SWV) for real-time tracing**.
- **Code generation**: Automatically generates initialization code for selected **STM32 MCU peripherals and middleware**.
- Build and stack analyzers: Provides information about project status and memory requirements.
- Editor features: Includes automated code completion, syntax highlighting, call hierarchy, and code templates.
- **RTOS support**: Offers awareness for **FreeRTOS and AzureRTOS/ThreadX**.
- Peripheral configuration: Allows graphical configuration of microcontroller peripherals and middleware.

- Frequent updates: The IDE receives regular updates and has a supportive community.
- STM32CubeIDE aims to streamline the development process for STM32-based products, offering a comprehensive toolset for embedded systems engineers.

3.1.2 FreeRTOS

FreeRTOS is an open-source real-time operating system designed for embedded microcontrollers and microprocessors. Key features and characteristics of FreeRTOS include:

- Small footprint: FreeRTOS has a tiny memory footprint, typically ranging from 4000 to 9000 bytes for the kernel binary image.
- Multitasking: It provides a **multitasking scheduler** that allows for **efficient task management** and **prioritization**.
- Portability: FreeRTOS supports over 40 **different architectures**, making it highly portable across various **hardware platforms**.
- Simplicity: The kernel comprises only three files, making it easy to understand and implement.
- **Real-time capabilities**: FreeRTOS offers deterministic behavior, ensuring **time-critical events are serviced within defined response times**.
- **Open-source**: It is distributed under the MIT open source license, allowing for **free use in commercial products.**
- Inter-task communication: FreeRTOS provides mechanisms like task notifications, **message queues, semaphores, and stream buffers** for coordination between tasks.
- Memory management: It offers multiple memory allocation options, including the ability to create statically-allocated systems.
- Debugging features: FreeRTOS includes stack overflow detection and memory allocation failure capture to aid in troubleshooting.
- TCP/IP stack: It includes a lightweight networking stack for internet connectivity.

FreeRTOS is widely used in **embedded systems** and **IoT devices** due to its efficiency, small size, and robust feature set. It is maintained by Amazon and offers integration with AWS services for cloud-connected IoT applications.

3.2 Hardware

3.2.1 Nucleo-F446RE

The Nucleo-F446RE is a development board from **STMicroelectronics** featuring the STM32F446RE microcontroller. Key features include:

- Microcontroller

- STM32F446RET6 MCU with **ARM Cortex-M4 core** and FPU
- 180 MHz max CPU frequency
- 512 KB Flash memory and 128 KB SRAM
- Operating voltage range: 1.7V to 3.6V

- Connectivity

- Arduino Uno V3 compatible headers
- ST morpho extension headers
- **On-board ST-LINK/V2-1 debugger/programmer**
- **USB connectivity for programming and power**

- Peripherals

- 3 user LEDs
- 2 push buttons (user and reset)
- Up to 50 **GPIO pins**
- Multiple **communication interfaces** (UART, SPI, I2C, CAN, USB OTG)
- 12-bit ADC and DAC

- **Power Options**

- USB VBUS or external power supply (**3.3V, 5V**, 7-12V)

- Development Support

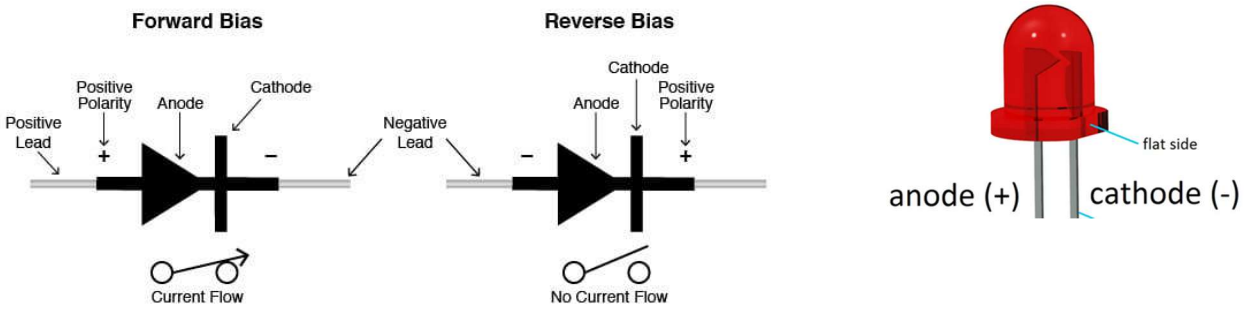
- Compatible with STM32CubeIDE and Mbed OS
- Comprehensive software libraries and examples available

3.2.2 Light Emitting Diodes (LED) and resistor

LEDs (Light Emitting Diodes) and resistors work together in electronic circuits to control current and produce light efficiently.

- Light emission: LEDs convert **electrical energy into light when current flows** through them.
- Efficiency: LEDs are more energy-efficient than incandescent bulbs, producing more lumens per watt.
- Versatility: They come in various colors and are used in diverse applications, from digital displays to traffic signals.

Diode



Resistance Functions in LED Circuits:

- **Current limitation**: **Resistors are crucial for limiting the current flowing through the LED to prevent damage.**
- Voltage regulation: They help manage the voltage drop across the LED, which is approximately constant over a wide range of operating currents.
- **Power management**: Resistors dissipate excess energy as heat, protecting the LED from excessive current.

The relationship between LEDs and resistors is governed by **Ohm's law and Kirchhoff's circuit laws**. The appropriate resistor value is calculated by subtracting the LED's forward voltage (Vf) from the supply voltage and dividing by the desired operating current.

3.2.3 Push Button

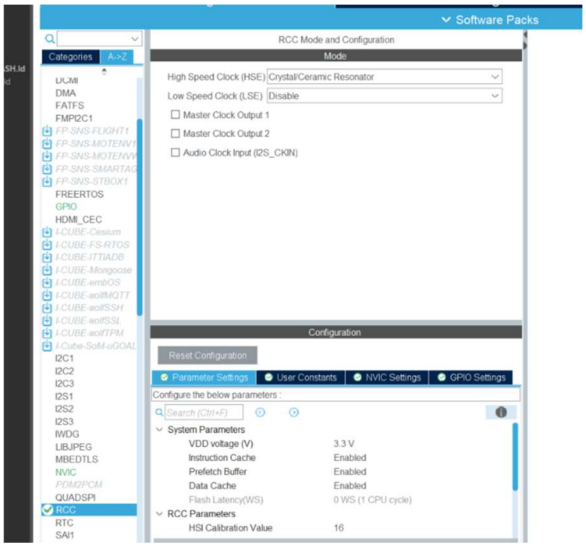
Push button switches are simple, yet versatile electrical components used to **control the flow of current in circuits**. They function by making or breaking an electrical connection when pressed.

- Operation: When pressed, the button actuates an internal mechanism that either closes or opens the circuit, allowing or interrupting current flow.
- **Contact configurations**:
  - **Normally Open (NO):** Circuit closes when pressed.
  - **Normally Closed (NC):** Circuit opens when pressed.

3.3Setting up

Step 1: Open STM32Cube IDE and create a new project

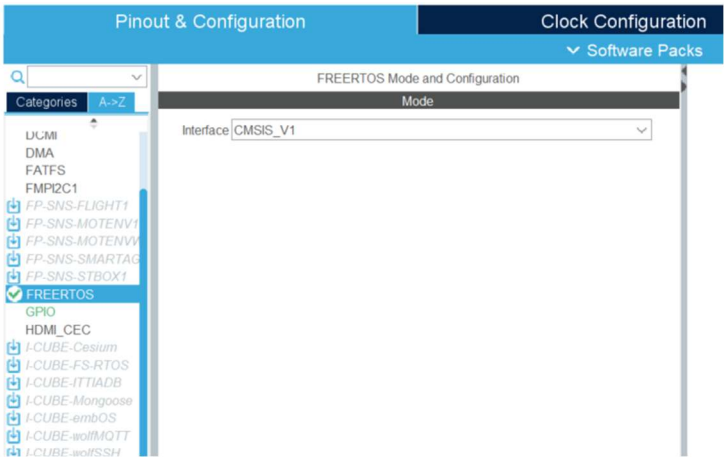
Step 2: Setup RCC (Reset and Clock Control): manages system and peripheral clocks, reset functionality, system initialization and ongoing operation that can be crucial for real-time applications.

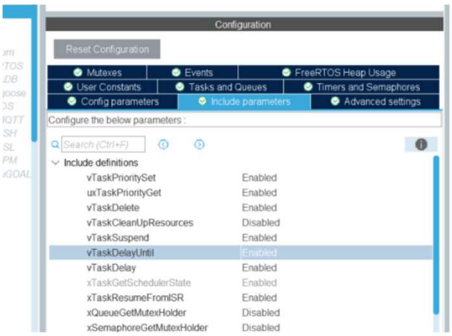


- Crystal and ceramic resonators: generate continuous vibration and electrical feedback. They offer high frequency stability and accuracy, provide wide frequency range (10 kHz – 100 MHz).

Step 3: configure the CMSIS interface

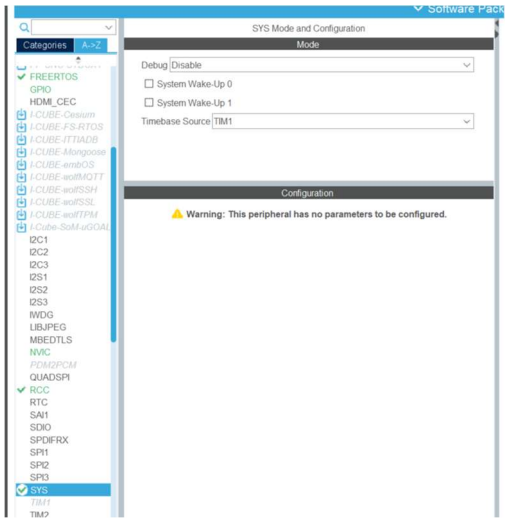
- The FreeRTOS CMSIS interface for STM32 is a wrapper that provides a standardized API for using FreeRTOS on STM32 microcontrollers



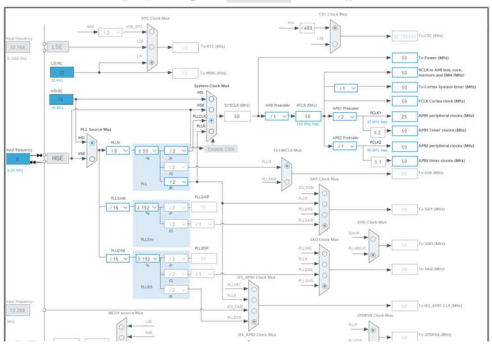


Step 4: Set the timebase source in SYS:

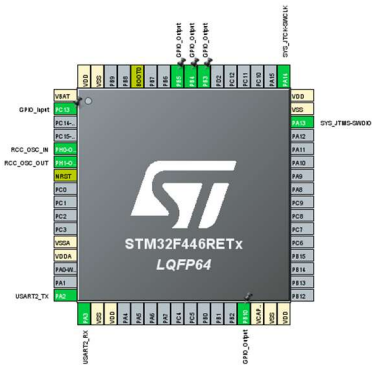
- SysTickTimer: 24-bit timer for simple timing tasks
- TIM groups: 16 to 32-bit timer for high resolution timing tasks
- The timebase is used by the HAL for functions like delays and timeouts.



Step 5 (Optional): clock configuration. The STM32 clock configuration involves several key components and values that can be adjusted to optimize performance and power consumption



Step 6: Define the pinout



3.4 Libraries and Dependencies

3.4.1 HAL (Hardware Abstraction Layer)

- HAL is a high-level driver layer provided by STMicroelectronics for STM32 microcontrollers. It offers a simple, generic set of APIs to interact with the hardware peripherals

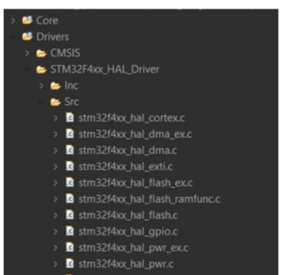
- Provides ready-to-use project templates for supported boards
- Initializes peripherals and handles low-level operations
- Offers a higher level of abstraction, making it easier to write portable code
- Can be more memory-intensive compared to lower-level alternatives

To use HAL:

- **Include** the necessary **HAL header files** in your project
- Initialize the HAL using **HAL_Init()** function
- Configure and use specific **peripheral APIs** as needed

Useful built-in functions *(in this project)*

- **HAL_GPIO_TogglePin():** toggle the state of a specified GPIO pin. If the pin is currently high (1), it changes it to low (0), and vice versa.
  - It can be used to change the state of the LEDs



```
/**
  * @brief  Toggles the specified GPIO pins.
  * @param  GPIOx Where x can be (A..K) to select the GPIO peripheral for STM32F429X device or
  *                      x can be (A..I) to select the GPIO peripheral for STM32F40XX and STM32F427X devices.
  * @param  GPIO_Pin Specifies the pins to be toggled.
  * @retval None
  */
void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
```

Example:

```
HAL_GPIO_TogglePin(GPIOB,GPIO_PIN_5);
```

- **HAL_GPIO_Init(GPIO port, pointer to GPIO_InitTypeDef structure):** initializes GPIO pins according to specified parameters

```
GPIO_InitTypeDef GPIO_InitStruct = {
  .Pin = GPIO_PIN_8,
  .Mode = GPIO_MODE_AF_PP,
  .Pull = GPIO_NOPULL,
  .Speed = GPIO_SPEED_FREQ_VERY_HIGH,
  .Alternate = GPIO_AF1_TIM1
};
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

- **HAL_GPIO_ReadPin(GPIO port, pin number):** reads the state of a specified GPIO pin
  ⇨ Return GPIO_PIN_SET if pin is high, GPIO_PIN_RESET if low

- **HAL_GPIO_WritePin(GPIO port, pin number, desired state):** sets the state of a specified GPIO pin

```
HAL_GPIO_WritePin(IDD_CNT_EN_GPIO_Port, IDD_CNT_EN_Pin, GPIO_PIN_RESET);
```

- **HAL_UART_Transmit():** transmit a specified amount of data over a UART interface. It is commonly utilized for sending strings, bytes, or other data types to a terminal or another device.

```
/**
  * @brief  Sends an amount of data in blocking mode.
  * @note   When UART parity is not enabled (PCE = 0), and Word Length is configured to 9 bits (M1-M0 = 01),
  *         the sent data is handled as a set of u16. In this case, Size must indicate the number
  *         of u16 provided through pData.
  * @param  huart Pointer to a UART_HandleTypeDef structure that contains
  *               the configuration information for the specified UART module.
  * @param  pData Pointer to data buffer (u8 or u16 data elements).
  * @param  Size  Amount of data elements (u8 or u16) to be sent
  * @param  Timeout Timeout duration
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, const uint8_t *pData, uint16_t Size, uint32_t Timeout)
```

3.4.2 CMSIS (Cortex Microcontroller Software Interface Standard)

CMSIS is a vendor-independent hardware abstraction layer for ARM Cortex-M processors, defined by ARM. It is used for **multitasking management (e.g., allowing efficient management of different traffic light states)**. To use CMSIS:

- **Include** the appropriate **CMSIS header files** in your project
- Access hardware registers and peripherals using CMSIS-defined structures and functions: **“MX_GPIO_Init();”, “MX_USARTs_UART_Init();”.**

- Utilize CMSIS-Core functions for system-level operations

3.4.3 Other useful functions

- **osDelay(uint32_t millisec):** The function delays the execution of the current task for a specified number of milliseconds. The actual delay time depends on the RTOS tick rate and system clock frequency

3.4.3 C concepts

3.4.3.1 Variables

Variables in C are used to store data of different types. They must be declared before use, specifying their data type.

```c
int age = 25;
float height = 1.75;
char grade = 'A';
```

Pointers: the variables that store memory addresses

```c
int num = 10;
int *ptr = &num;
printf("Value: %d\n", *ptr); // Outputs: Value: 10
```

3.4.3.2 Headers

Header files in C contain function prototypes, macros, and declarations. They are included using the #include directive.

```c
#include <stdio.h>
#include <math.h>
```

3.4.3.3 Functions

Functions are reusable blocks of code that perform specific tasks. They can take parameters and return values.

```c
int add(int a, int b) {
    return a + b;
}

int main() {
    int result = add(5, 3);
    printf("Sum: %d\n", result);
    return 0;
}
```

3.4.3.4 If statement

If statements are used for decision-making in C, executing code blocks based on conditions.

```c
int x = 10;
if (x > 5) {
    printf("x is greater than 5\n");
} else {
    printf("x is not greater than 5\n");
}
```

3.4.3.4 For loop

For loops are used to execute a block of code repeatedly for a specified number of times.

```c
for (int i = 0; i < 5; i++) {
    printf("Iteration %d\n", i + 1);
}
```

3.4.3.4 while loop

While loops execute a block of code as long as a specified condition is true.

```c
int count = 0;
while (count < 3) {
    printf("Count: %d\n", count);
    count++;
}
```

3.4 Concepts

3.4.1 Task

Tasks are fundamental units of execution in FreeRTOS for STM32 microcontrollers. They allow for **concurrent execution of different parts** of your application.

Task Concepts

- Creation: Tasks are created using the osThreadNew() function in CMSIS-RTOS V2
- Priority: Each task has a priority level, determining its execution order
- Scheduling: The FreeRTOS scheduler manages task execution based on their priorities
- Stack: Each task has its own stack space for local variables and function calls

Main steps

- **Step 1:** Create task handler

```c
/* Definitions for Task1 */
osThreadId_t Task1Handle;
```

- Step 2: Define the entry function for the task

```c
void StartTask1(void *argument);
```

```c
const osThreadAttr_t Task1_attributes = {
  .name = "Task1",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityNormal,
};
```

- Step 3: Define the thread inside the main system
- Step 4: Create the thread and assign the task ID

```
Task1Handle = osThreadNew(StartTask1, NULL, &Task1_attributes);
```

- Step 5: Create supportive functions and variables (optional)
- Step 6: Create the entry function for the task

```
void StartTask1(void *argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {

      HAL_UART_Transmit(&huart2, dataTask1, sizeof(dataTask1), 1000);
     // HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3|GPIO_PIN_4|GPIO_PIN_5, GPIO_PIN_RESET);
      HAL_GPIO_TogglePin(GPIOB,GPIO_PIN_4);//Green LED

      osDelay(1000);
  }
  /* USER CODE END 5 */
}
```

4. Requirements

Total marks: /100

Presentation (25 marks)

- Introduction: Briefly introduce the project and its significance.
- Objectives: Clearly state the goals and objectives.
- Problem Analysis: Identify and analyze the problem.
- Solution Design: Present rough sketches, flowcharts, or diagrams to explain your design process.
- Component Usage: Explain circuit design and component selection.
- Algorithmic Approach: Explain algorithm developed and showcase code snippets.
- Implementation Evaluation: Evaluate implementation, including code compilation.
- Results and Validation: Demonstrate the demo and explain results.
- Address Challenges: Discuss challenges faced and resolutions.
- Teamwork: Ensure each team member contributes equally to presentation and implementation.

Maximum Presentation Time: 10-15 minutes

Tip: Provides a concise overview of your report to the TA.

Code (15 marks)

- Code Explanation: Explain methods, initialization, and pin setup.
- Code Compilation: Demonstrate code compilation and loading.
- Implementation Detail: Provide insights into code implementation.

- Developed Algorithm: Explain the algorithm developed to follow the design approach proposed. The algorithm should realize your design. There needs to be alignment between your planned design and actual implementation.

Circuit Design (10 marks)

- Demo Setup: Showcase demo setup, inputs, outputs, and connections.
- Component Presentation: Present used components.

Demo (25 marks)

Working Demo: Showcase demo functionality to the TA.

Design (25 marks)

- Flowcharts and Diagrams: Present design flowcharts and diagrams.
- Design Explanation: Explain design principles, logic, and rationale.

Remember, while code implementation is crucial, the foundation laid by your design is even more critical. A well-thought-out design ensures a smoother and more effective implementation process.