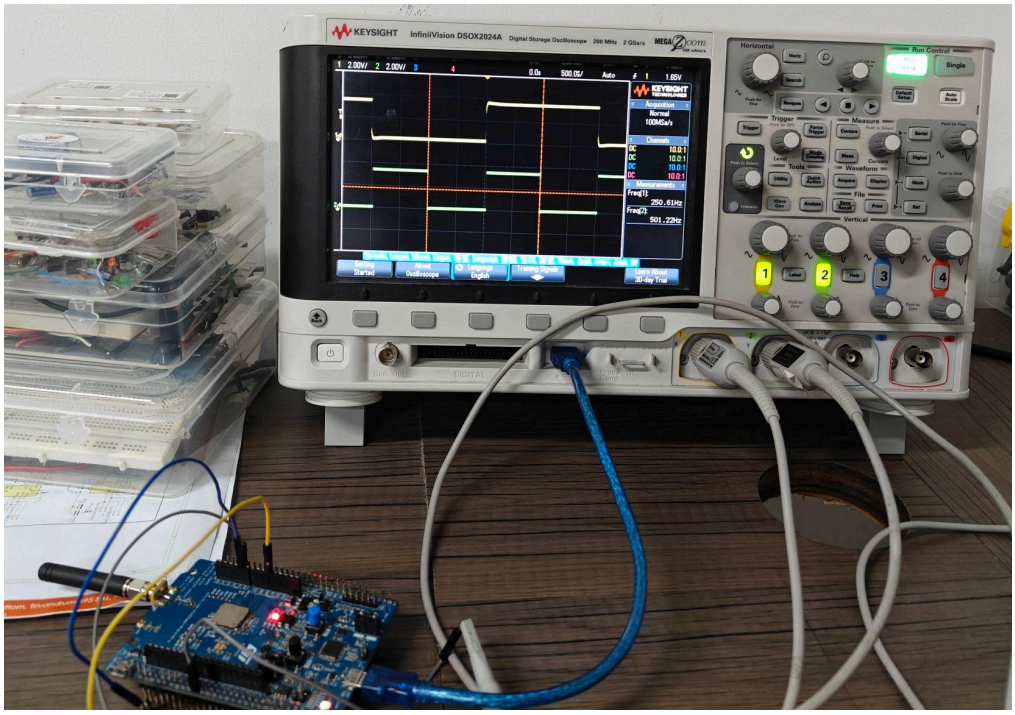
ICFOSS

RTOS IMPLEMENTATION IN STM 32

ROHITH S - Hardware IoT INTERN



Introduction

RTOS stands for Real Time Operating System. And as the name suggests, it is capable of doing tasks, as an operating system does. The main purpose of an OS is to have the functionality, where we can use multiple tasks at the same time. Which obviously isn't possible with bare metal.

This tutorial is the first in the series of many, and will cover the following:-

1.) Setting up Free RTOS using CubeMX

2.) Benefit of using a RTOS.

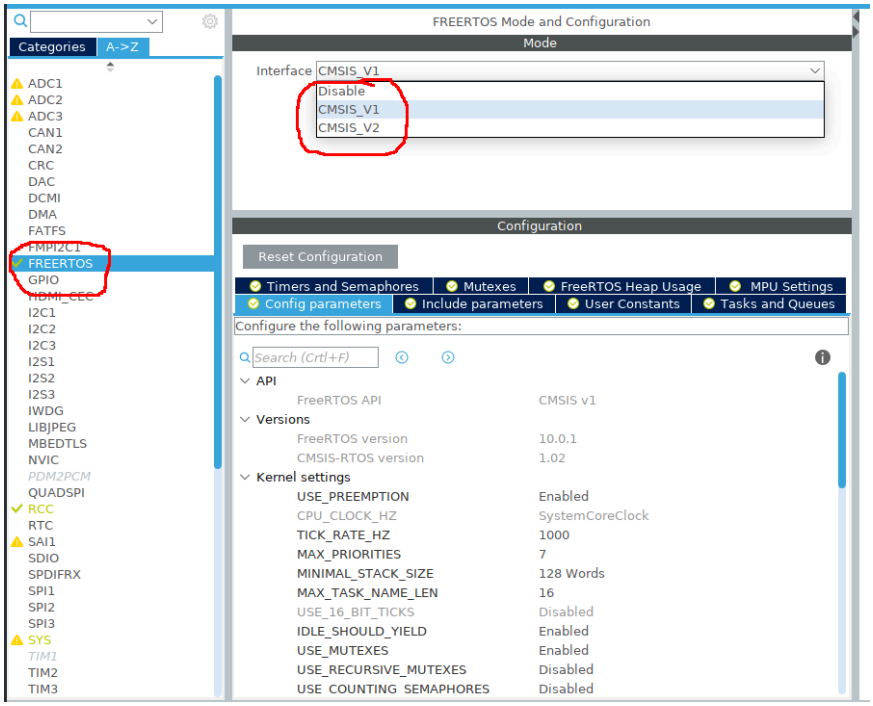
3.) Creating tasks with or without CubeMX

4.) Using priorities to sort out some common problems

Let's start by setting up the CubeMX

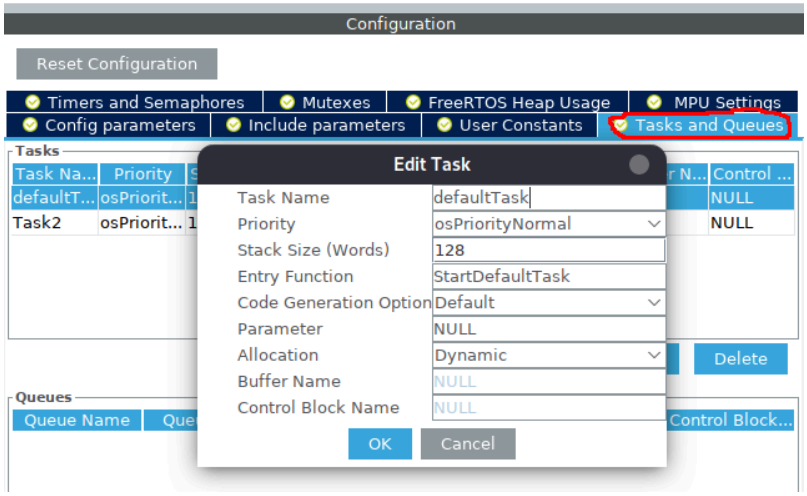
CubeMX Setup

After selecting the controller, the CubeMX will open the default page for you. Now select the **FREERTOS** and follow the screenshot below



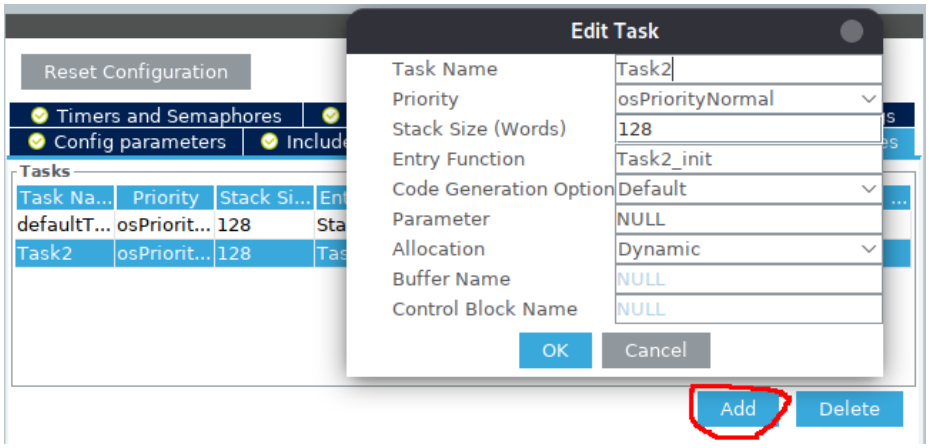
I am choosing version 1, because it is supported by majority of STM32 devices.

Next, go to the '**tasks and queues**' tab and here you will see a default task, already created for you. Double click it and you can see the following information.



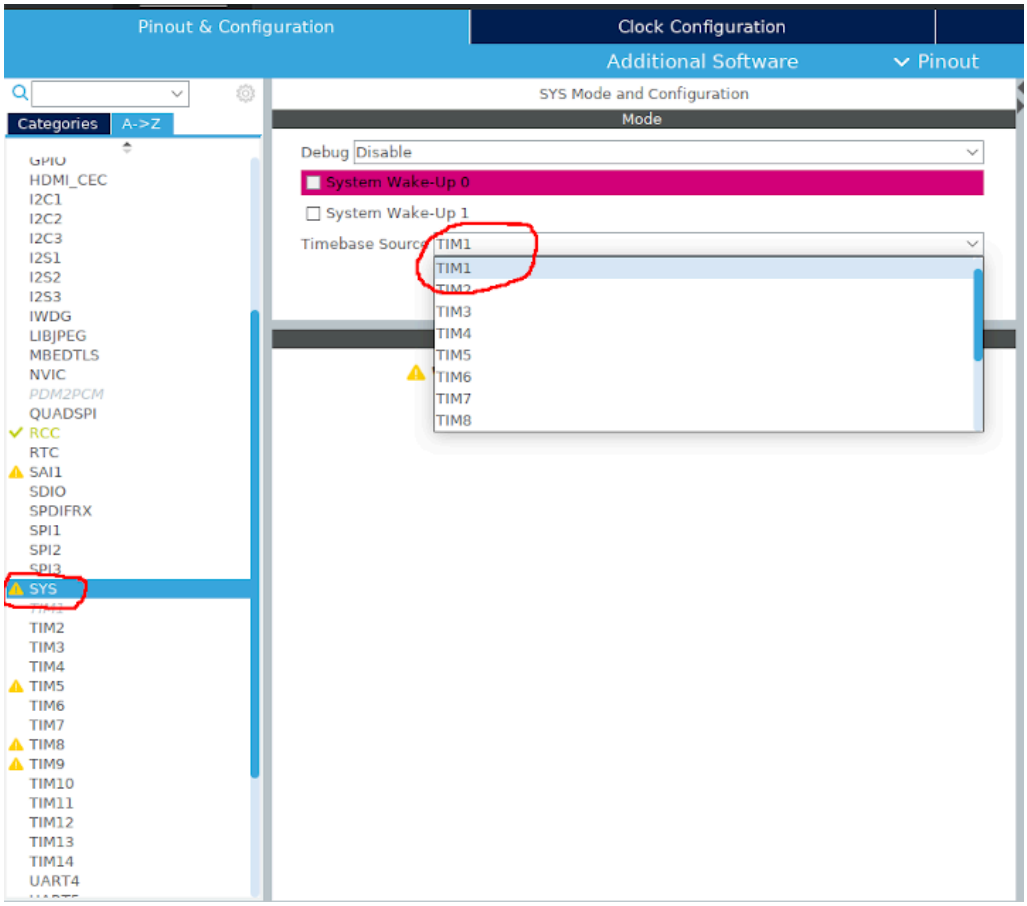
Don't worry about all this information. For now, you have to focus only on the **Task name**, **priority**, and the **entry function**.

Now, we will create one task here, and below are the properties of that task



So I am calling it **task 2**, with **normal priority**, and the entry function is **Task2_init**. You will get a better idea about these, once we write the program.

Also one important thing about using RTOS is that, we can't use systick as the time base. So go to sys, and choose some other timebase as shown below



Other than these, I am using UART 2 for transmitting data, and pins PA0, and PA1 as output.

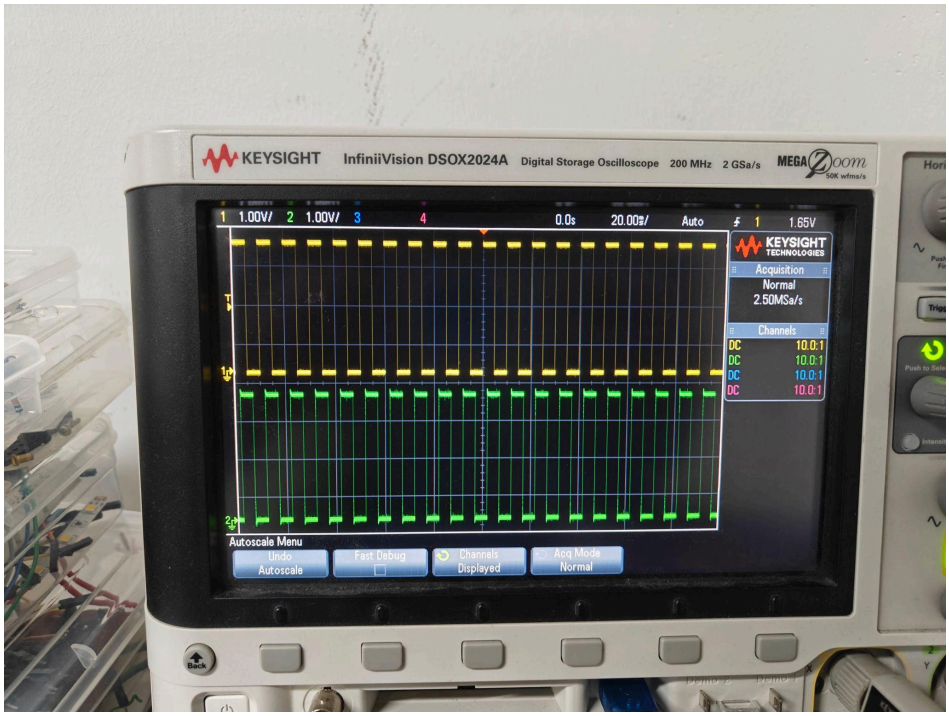
After the code is generated, open the main.c file and it's time to know the importance of using RTOS.

Benefit of using a RTOS

Let's assume a situation, where we want to toggle 2 pins, and each at some respective delays. So basically, we want both the pins to toggle at the same time. Of-course, it's not possible with simple programming, because the microcontroller will execute 1 instruction first and than second. And first one have to wait for the second execution to finish, even in a while loop.

```
while(1)

{

 HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_0);

 HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_2);

}
```

Result: It is visible that there is a delay between running both the tasks . the following is the output from each of the pin A0 and B3.

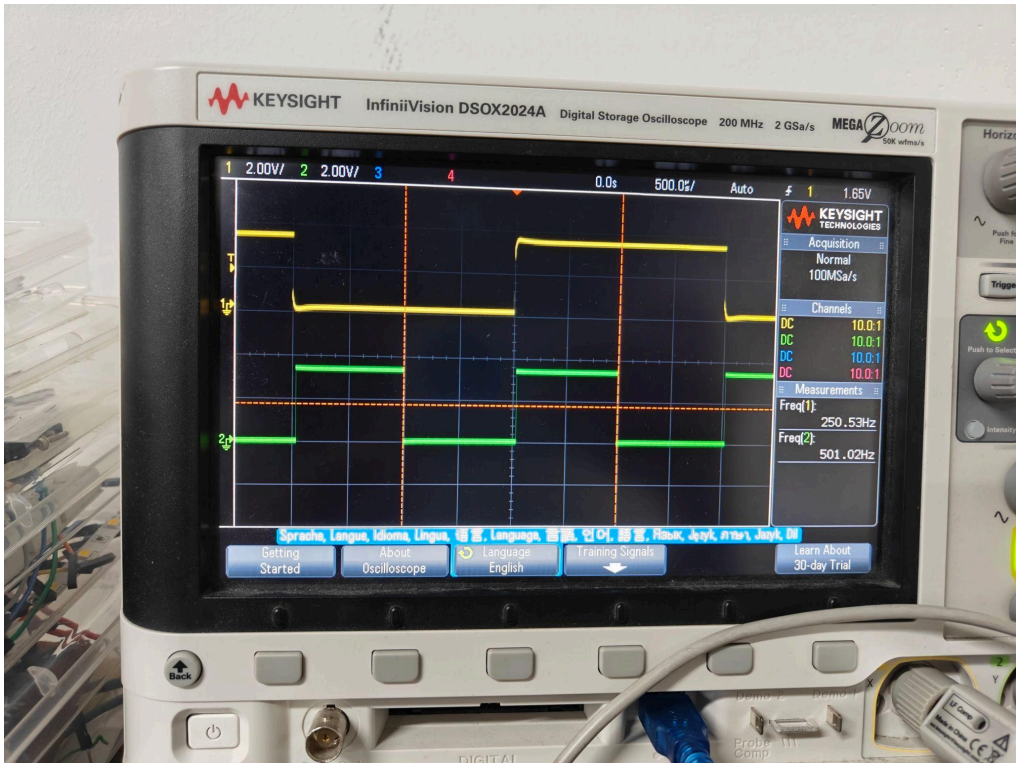


To overcome this problem, we will use the RTOS. So, we have created 2 tasks, and if you scroll down the main.c file, you will see the entry functions for the tasks are defined there.

```
void StartDefaultTask(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
     HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_0);
     osDelay(1);
  }
  /* USER CODE END 5 */
}

/* USER CODE BEGIN Header_Task2_init */
/**
* @brief Function implementing the Task2 thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_Task2_init */
void Task2_init(void const * argument)
{
  /* USER CODE BEGIN Task2_init */
  /* Infinite loop */
  for(;;)
  {
```

```
  HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_1);
  osDelay(1);
  }
  /* USER CODE END Task2_init */
}
```

RESULT:



We will toggle the pin PA0 in the default task, and pin PA1 in the Task2. This way the scheduler will schedule the time for both of these tasks, so that they get enough time to be executed.

You can check below the oscilloscope reading, when the above code was executed.

Creating a Task

In order to create a new Task, we have to follow some set of steps, and they are as follows:-

1.) Define a **ThreadID** for the task. This variable will store the unique ID of the task, once created. Later, all the operations will require this ID.

```
osThreadId Task3Handle;
```

2.) Define the **entry function** for the task. This is the main function of the task. Your program will be written inside it. Remember that the tasks in the Free RTOS, are not designed to handle any return value. So, the entry function should always have an infinite loop, inside which, your whole program should be written.

```
void Task3_init (void const * argument)
{
  while (1)
  {
    // do something
    osDelay (1000);  // 3 sec delay
  }
}
```

3.) Inside our main function, we need to **define the task** first and than **create** it.

```
// define thread
  osThreadDef(Task3, Task3_init, osPriorityBelowNormal, 0, 128);
  //create thread
  Task3Handle = osThreadCreate(osThread (Task3), NULL);
```

- **osThreadDef** takes the parameters as the name of the task, the entry function, the priority, instance, and the stack size.
- After the task is defined, we can create it using **osThreadCreate**, and assign the ID to the **Task3Handle**

Handling Priorities in Free RTOS

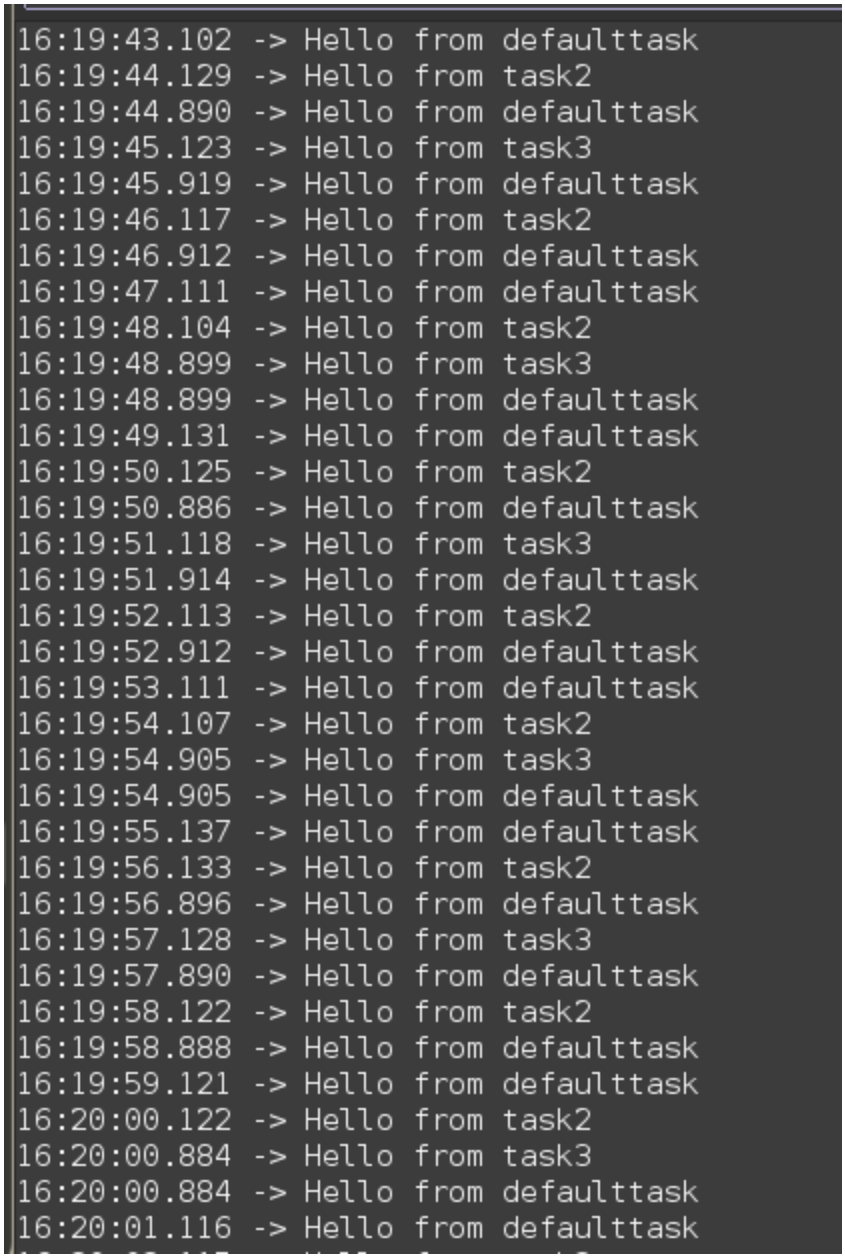
- Till now, we saw how to multitask using RTOS. But there are certain problems which comes with it.
- For instance, assume that, we want to send some data over the **UART**, via all three tasks, at the same time.
- When we write a program to do so, the result will not be the same exactly. Instead, the transmission will take place in a way that only one task will send the data in 1 second, than another task for another second and so on.
- This happens, when we try to use the shared resources among the tasks with same priorities.
- Second task have to wait for the first to complete it's execution, and than only the control comes to it.
- And similarly the third task will wait for the second one to finish. You can see this in the video below, it's better explained there.

To avoid these situations, we use different priorities for different tasks. That means we have to redefine our task priorities in the main function.

```
osThreadDef(defaultTask, StartDefaultTask, osPriorityNormal, 0, 128);

osThreadDef(Task2, Task2_init, osPriorityAboveNormal, 0, 128);

osThreadDef(Task3, Task3_init, osPriorityBelowNormal, 0, 128);
```

- Now the **Task2** have the highest priority, than **Default task**, and **Task3** with the lowest.
- When the program runs, **Task2** will execute first, than **default task** and at last the **Task3**. And all three tasks will send the data at the same time.

  Do analyse the result. below:



Extra knowledge:

The difference between `osDelay` and `HAL_Delay` lies in their implementation, purpose, and behavior in an RTOS environment (like FreeRTOS or CMSIS-RTOS). Here's a detailed comparison:

---

1. `osDelay`

- **Purpose: Suspends the current thread/task for a specified duration in milliseconds.**
- **Usage: Used in an RTOS-based system to delay task execution without blocking the entire system.**
- **Behavior:**
    - **Puts the calling task in a blocked state for the delay duration, allowing the RTOS to schedule other tasks.**
    - **The system remains responsive as other threads/tasks can execute during this time.**
- **Implementation:**
    - **Relies on the RTOS kernel tick timer for delay timing.**
    - **The actual delay may depend on the RTOS tick frequency (e.g., `configTICK_RATE_HZ` in FreeRTOS).**

Example:

```
osDelay(1000); // Suspends the current task for 1000 ms (1 second)
```

- **Precision**: Can be less precise due to the granularity of the RTOS tick.

---

2. `HAL_Delay`

- **Purpose**: Implements a **blocking delay** for a specified duration in milliseconds.
- **Usage**: Commonly used in **bare-metal programming** or **non-RTOS applications**.
- **Behavior**:
    - **Blocks the CPU** entirely for the specified duration, preventing any other code from executing.
    - The system remains unresponsive during this time.
- **Implementation**:

- Uses a busy-wait loop that relies on the **SysTick timer** (a hardware timer) to count milliseconds.
- The SysTick interrupt increments a counter, which `HAL_Delay` checks in a loop.

Example:

HAL_Delay(1000); // Blocks the CPU for 1000 ms (1 second)

Precision: Typically more precise, as it directly uses the SysTick timer.