The Programing Phase Documentation.

Using only the UART Cable we can power and program the Blue Pill.

We need two software to do so:

1. STM32CUBEPROGRAMER.
2. STM32CUBEIDE.

The STM32CUBEIDE is the actually editor.

STM32CUBEPROGRAMER is used to actually connect the STM in the blue pill to the computer.

It takes the program script in the format of (**.bin**) and downloads the script to the STM.

Keep in mind the position of the BOOT0 when downloading to the STM. In the case on downloading, it should be HIGH i.e., should be equal to 1. And already RESET.

To do subtask(1):

Use

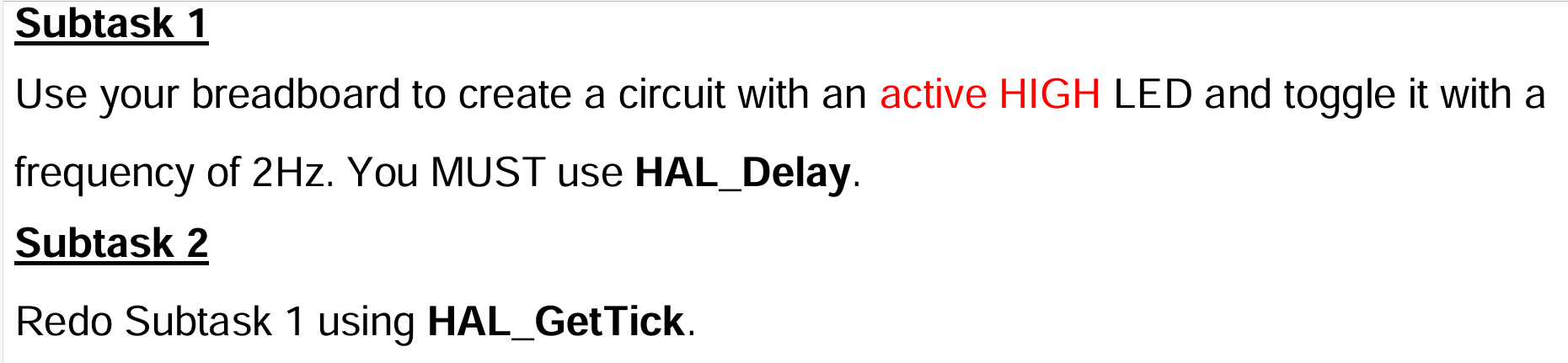
HAL\_GPIO\_TogglePin(GPIOA, GPIO\_PIN\_no);

This tells the MCU to be TOGGLE pin PA(no)

Then we use

HAL\_Delay(ms);

To do subtask (2):



HAL\_GetTick()

This function takes no input argument.

It outputs the time in ms taken since the start of the program

**The premise of usage in the task**:

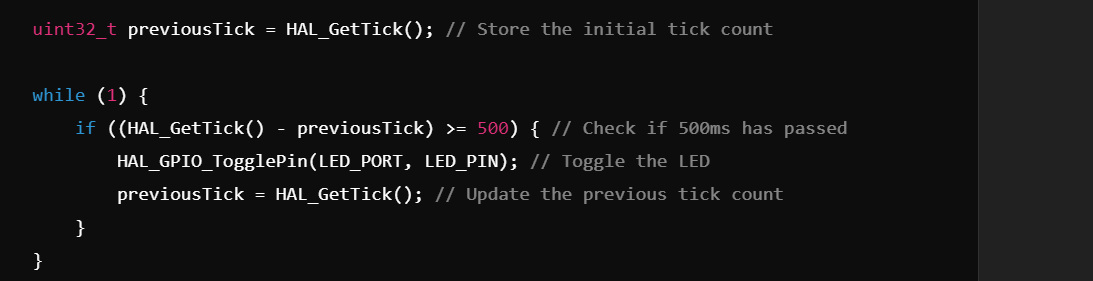
We can take a variable to store the value of the time elapsed.

Then we make a condition that as long as the current time elapsed and prerecorded time elapsed (the time value stored in the variable) have a difference less than 500ms don’t do anything:

But if the difference exceeds 500ms then:

1. Update the time value stored to the current.
2. Toggle the LED.

The code snippet:



the take away of subtask (2) is

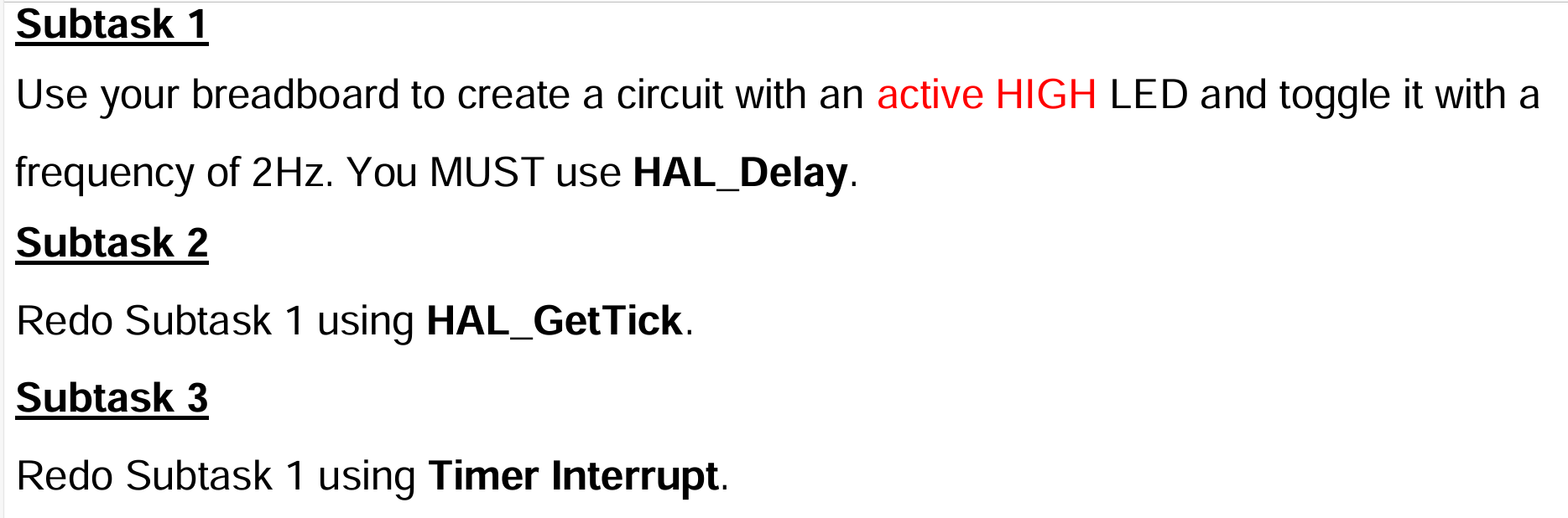
**Blocking Delay:**

* **Definition:** The processor does nothing else until the delay completes.
* **Example:** HAL\_Delay(500) halts execution for 500ms.
* **Pros:**
  + Simple to implement.
  + Easy to understand in sequential code.
* **Cons:**
  + **Wastes CPU cycles.**
  + **Prevents multitasking**—no other operations can run during the delay.

**Non-Blocking Delay:**

* **Definition:** The **delay runs concurrently with other tasks**; the CPU can execute other code while waiting.
* **Example:** Using HAL\_GetTick() to check elapsed time in the main loop without halting execution.
* **Pros:**
  + Efficient use of CPU time.
  + Enables multitasking and responsive systems.
* **Cons:**
  + More complex to implement.
  + ***Requires careful management*** (e.g., state machines, interrupts).

Subtask(3)



Premise:

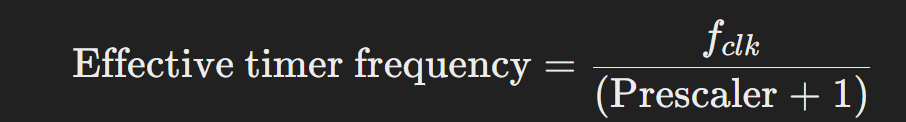
* The MC contains timers.
* We can set the CPU of the MC to receive an interruption signal.
* This interruption Signal tells the MC to stop it’s current execution and go execute the code and a prespecified ISR unit.
* ISR (Interruption Service Routine): is the code to be executed when the interruption signal is received.
* After executing the ISR, the CPU will resume execution of the program that got interrupted.

**Timer:**

the timer as a counter that counts the number of pulses/ticks, and when it reaches a defined value, it performs an action (e.g., toggling an LED) or generates an interrupt.

**Clocks and timers:**

* **System Clock:** The high-frequency clock provided to the microcontroller.
* **Prescaler:** Divides this system clock by a specified factor, yielding a lower effective clock frequency for the timer.
* **The** goal is to adjust the frequency of the clock of the timer according to our need.
* This is done using the prescalar equation.

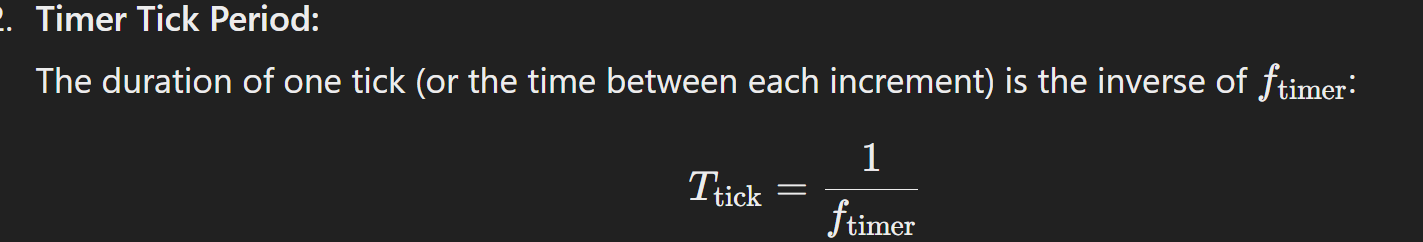


In theory, yes—if you could set the **prescaler to reduce your clock all the way down to a frequency where one tick equals your desired interval (500 ms, i.e., 2 Hz)**, you’d only need one tick (or a counter period of 0) for each interrupt. However, here are some important points:

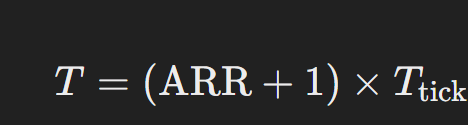
* 1. **Hardware Limitations:**  
     The prescaler value is usually limited by the hardware. Dividing a high-frequency clock (say, 72 MHz) directly down to 2 Hz would require a **very large prescalar value, which might exceed what the timer allows**.
  2. **Resolution and Flexibility:**  
     Even if you could set such a huge prescaler, using both the prescaler and **the counter period (ARR) together gives you more flexibility and finer resolution**. The prescaler first reduces the clock to a manageable rate, and then the counter period lets you adjust the overall time interval by counting a set number of these ticks.

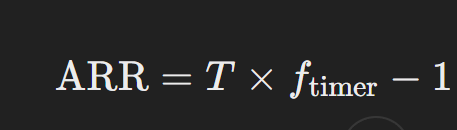
**Counter Period (Auto-Reload Register, ARR)**

* **Definition:**  
  The **counter period** is determined by the **Auto-Reload Register (ARR)**. It defines the maximum value the timer's counter will reach before it resets to zero.



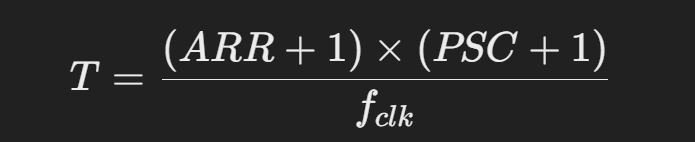
**Ftimer** is the effective frequency of the timer



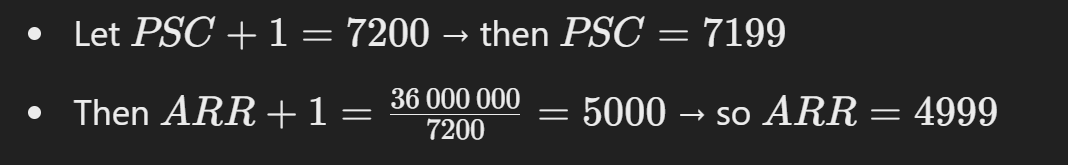
So to get the ARR value we need

In our task we have T = .5 sec and we can set appropriate values for the Prescalar and the Counter Period or the ARR.

***The Ultimate Equation***



Given T, choose an acceptable value of the (PSC) or the (ARR) then compute the other. Note that **fclk** = 72MHz.

For example:

For the coding part of the task: we have

**HAL\_TIM\_Base\_Start\_IT(&htim2);**

**Which sets the timer into the interrupt Mode.**

Meaning, this tells the CPU to execute the following function when:

**HAL\_TIM\_PeriodElapsedCallback(TIM\_HandleTypeDef \*htim)**

When the timer overflows i.e., Reaches its compared value (the ARR)

***Weak Function:***

***if you provide your own definition of the function in your project, your version will override the default one.***

The **HAL\_TIM\_PeriodElapsedCallback()** is a **Weak Function**.

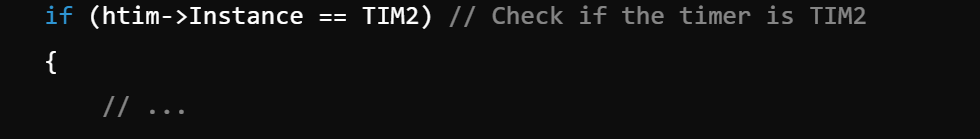
**Is It Customizable?**

* **Yes the HAL\_TIM\_PeriodElapsedCallback()** function is Customizable.

You can completely customize this function. You write your own version to perform any tasks you want when the timer interrupt occurs.

**Is True that this function is called for all timers?**

Yes, that’s why we need an if statement to specify the Timer we want to consider.



The location of function

**HAL\_TIM\_PeriodElapsedCallback(TIM\_HandleTypeDef \*htim)**

is

**User code begin 4**

**The location of the function**

**HAL\_TIM\_Base\_Start\_IT(&htim2);**

Is

User Code begin 2