# ARM Microcontroller Based Programming

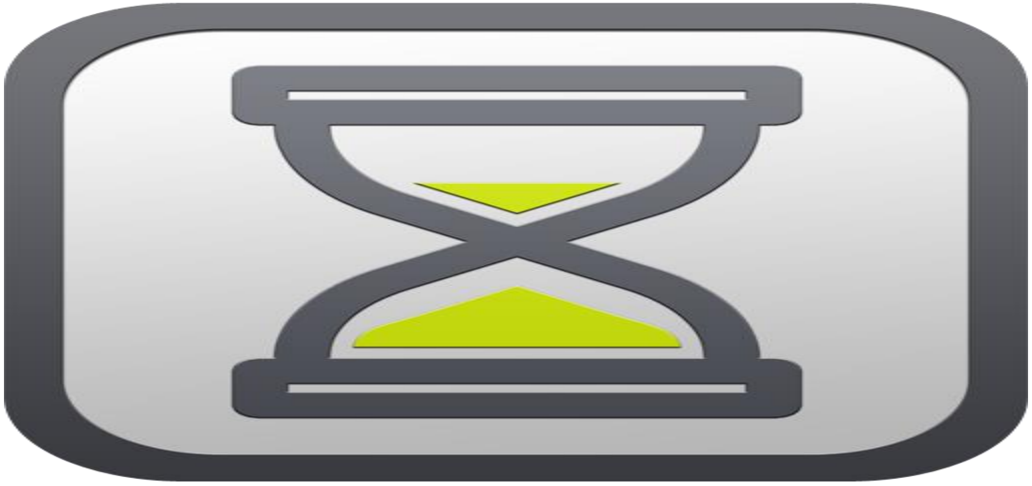
## Lecture Seven

## SysTick

# Introduction to Timers

## What is the timer?

The timer is an important peripheral in embedded systems, it maintains the timing of an operation in sync with a system clock or an external clock. The timer has so many applications such as measuring time, generating delays, they can also be used for generating baud rates.



# Timer Terminology

## Timer Resolution

Number of bits represents the timer register.

## Timer Tick Time

The time between to consecutive events, i.e. time needed to increment or decrement the timer register by 1

## Timer Count

Number of counts needed by the timer to count from 0 till it reaches 0 again. It shall be equals to $\boldsymbol{2^{resolution}}$

## Timer Overflow

The time needed by the timer to count from 0 until it reaches 0 again.
Sometimes the timer could count from its Max value until it reaches 0
It shall be equals to **Timer Count x Timer Tick Time**

## Timer Overflow

Prescaler is a division factor for the system clock (Processor Clock) before it is applied to the timer peripheral

# Timer calculations

- Timer clock = $\dfrac{System\ Clock}{Prescaler}$
- Timer clock = $\dfrac{1}{Timer\ Clock}$ = $\dfrac{Prescaler}{System\ Clock}$

Example

Assume a microcontroller working on a frequency of 8 MHz, calculate the timer tick time assuming the timer is using a prescaler of 8.

Solution

Timer Tick Time = $\dfrac{8}{8 * 10^6} = 1\ Microsecond$

i.e. this timer would be incremented or decremented every 1 micro second.

# Timer calculations

- Timer Overflow Timer = Timer Count x Timer Tick Time

$$= \quad 2^{Resolution} \quad \times \quad \frac{Prescaler}{System\ Clock}$$

Example

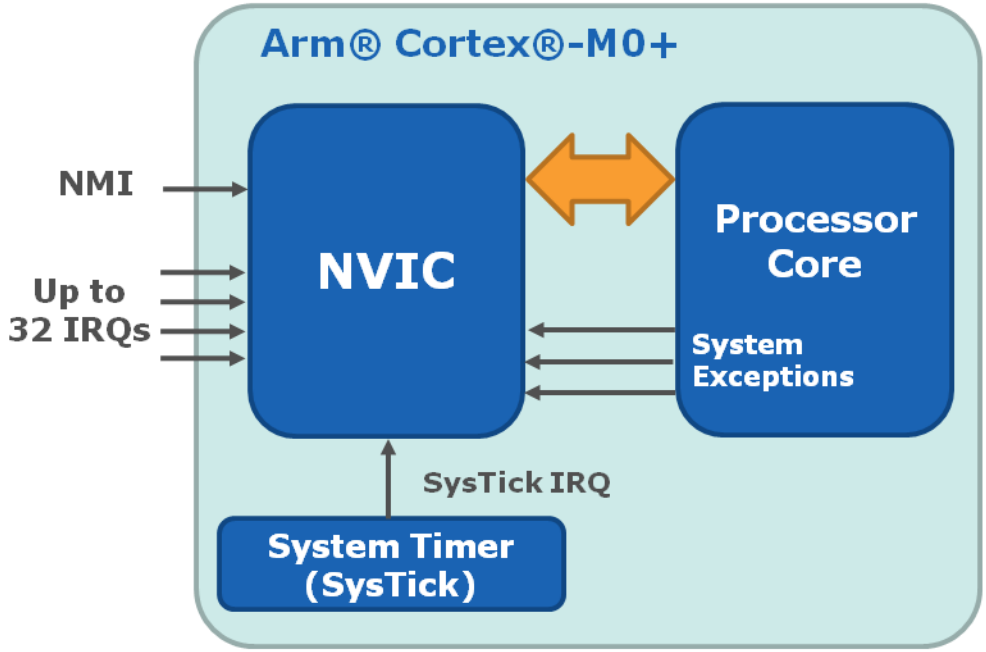
From the previous example, assuming the timer resolution is 16 bit, calculate the time of the overflow

Solution

The time of the overflow = 1 Microsecond x $2^{16}$

# SysTick Peripheral

The SysTick peripheral is one of the peripherals which are allocated inside ARM cortex-M3 processor. So SysTick is one of the core peripherals.




ARM Cortex-M0+ processor

# SysTick Peripheral

## SysTick Features

- Systick is allocated inside the processor so it will be faster in performing any operations (Core peripheral)
- Memory Mapped, means that its registers which is allocated inside the core have addresses on the memory, which means that we can access its registers using C language.
- Access level is privileged so that’s why we use Systick mainly in OS.
- SysTick is a 24-bit down counter so it means that it can count $2^{24}$ ticks.
- SysTick is a down counter timer which means that it can count from $(2^{24} - 1)$ to zero and it can generate an interrupt when it reaches zero.
- The main advantage of counting down is when calculating the preload value, you don’t  need to subtract the preload value from the maximum value of the timer counter to get the actual preload value, but we will directly assign the preload value in the timer preload register to make the timer start to count from.

# SysTick peripheral

## SysTick Features

- Working clock could be AHB or AHB/8.
- SysTick timer stops counting when the processor is halted during debugging. Depending on the design of the microcontroller, the SysTick Timer could also be stopped when the processor enters certain type of sleep modes.
- SysTick timer is portable between all cortex same family (M3).
- When the SysTick timer counter reaches zero, the counter loads the reload value from the RELOAD register. It does not stop until the enable bit in the SYSTICK Control and Status register is cleared

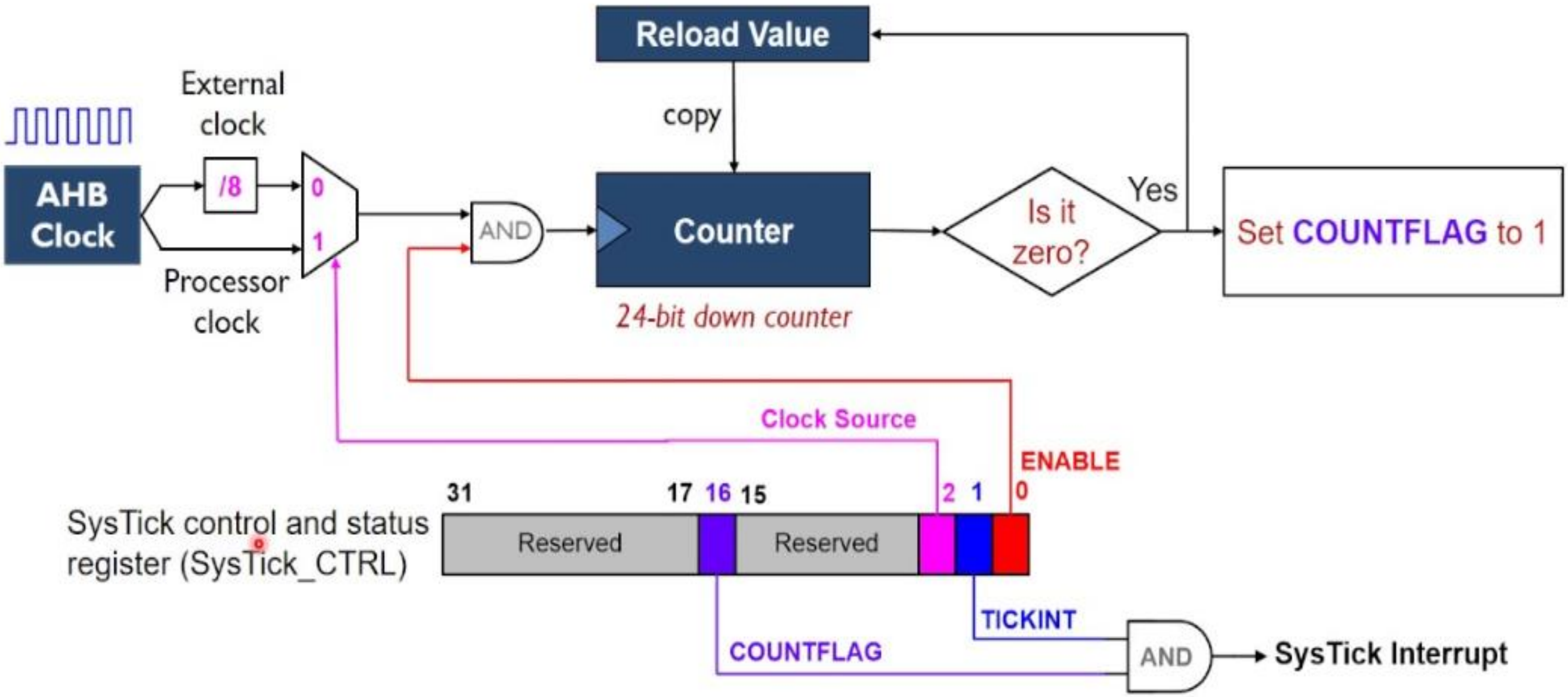
# SysTick peripheral

## How to operate SysTick

- We need to choose the Input clock source and the options are AHB or AHB/8
- We need to enable the peripheral interrupt by setting TICKINT bit in The SysTick control and status register.
- Starting Address of SysTick is at 0xE000 E010
- We mainly have two registers used to set the beginning of the counter.
- **Load Register :** This register used to set the preload value.
- **Value Register** : This register is the counter which is decremented by hardware.
- When the **Value Register** reaches zero, it will load the **Load Register** and then start counting down.
- If the **Value Register** is accessed using Software, it will be reset without setting the flag.
- If the **Load Register** is loaded with a value, the SysTick will start counting from this value, but at the next underflow.
- If we want to Set the value written in the **Load Register** to be loaded immediately to the **Value Register,** then we need to write the preload value into the **Load Register** and then put any value into the **Value Register**.
- The flag will return 1 when the timer is counted to zero.

# SysTick Peripheral

SysTick Sequence
Block diagram



# Delays

There are two methods to make a delay or to perform an action after some time.
For example, we want to toggle a LED every two seconds.
The two methods which we could perform this delay using them are called

- Busy wait delay.
- Interval delay.

## Busy wait delay

- The delay can be performed by using for loop or while loop.
- The processor will be halted inside this loop until this loop is terminated.
- Busy wait delay method is the simplest way to perform a delay, however it is not the best method to perform a delay because it causes the processor to be halted inside this loop until this loop is terminated.
- The processor will not execute any instruction and it will only execute this loop only whatever it takes.

# Delays

## Interval wait delay

- It is the second method of performing a delay.
- It is the most preferred method to perform a delay because this method is based on a timer interrupt.
- In this method we are going to use a timer peripheral to perform this delay.
- In this method the timer interrupt is configured to take place whenever the application needs to take an action.
- As the previous example where we need to toggle a LED every one second, so we will configure the interrupt timer to be fired when the timer counter counts a one second.
- Whenever the timer interrupt fires, the processor will execute the timer handler.
- In this method the processor could execute some other instructions without being halted executing inside the loop waiting this loop to be terminated
- There are three possibilities of using a timer interrupt after a desired time.

# Delays

## Interval timer

Interval timer is a normal functioning timer that counts until it overflows or downflows. The timer sets a flag and fires an interrupt if enabled.

## How to calculate a certain time based on an interval timer?

- **Step 1:** Calculate the overflow time of the used timer
- **Step 2:** Compare the desired time with the overflow time, and follow one of the following possibilities.

## Possibility 1

**Desired Time equals to overflow time**

This is the happy scenario possibility, which normally doesn't happen. If you are lucky and the desired time is equal to overflow time, then just enable the interrupt and take the desired action inside the ISR.

```
ISR (OverFlow)
{
    /* Take Desired Action */
}
```

## Possibility 2

**Desired Time than overflow time**

- **Step 1:** Calculate the number of counts needed

**Number of counts needed =** $\frac{\boldsymbol{Desired\ Time}}{\boldsymbol{Overflow\ Time}} * \boldsymbol{Overflow\ count}$

- **Step 2:** Preload the timer register with the following value:

  In case of SysTick timer which counts in a decrementing way so we will directly load the Number of counts needed into the timer register.

  **Preload Value =** Number of counts needed

- **Step 3:** Enable the timer interrupt and take the desired action

## Example on Possibility 2

Assuming 24 bit timer using a Clock of AHB/8 and the clock on AHB is 8MHz. An action is needed to be taken after 64 microseconds. Do the needed calculations and write the code.

## Solution

- **Overflow time =** $2^{24} * 1\ Microsecond$ **= 16777216 Microseconds**
- The desired time is 64 microseconds which is less than the overflow time. Then we calculate the needed number of counts
- **Number of needed counts =** (64 / 16777216) x 16777216 = 64 count
- **Preload value =** 64
- If the timer register is initialized with the preload value (64), then it needs only 64 count to overflow, i.e. the overflow interrupt would fire after only 64 count. Which is the desired time.

## Possibility 3 (Case 1)

**Desired Time more than Overflow time**

- **Step 1:** Calculate the number of overflows needed

**Number of overflows =** $\dfrac{\boldsymbol{Desired\ Time}}{\boldsymbol{Overflow\ Time}}$

- Assume that the number of the overflows needed is a ***decimal value***, then define a variable and increments it inside the overflow interrupt till we reach the desired number of overflows, then take the desired action.

## Possibility 3 (Case 2)

**Desired Time more than Overflow time**

- **Step 1:** Calculate the number of overflows needed

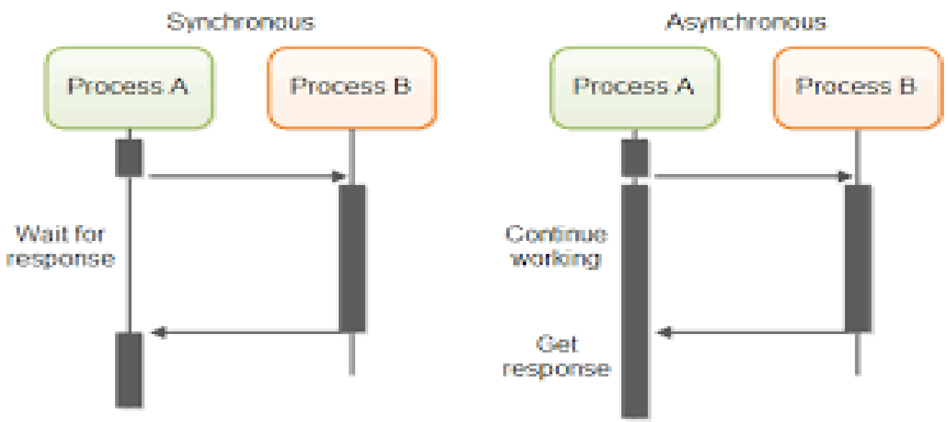
**Number of overflows =** $\frac{Desired\ Time}{Overflow\ Time}$

- Assume that the number of the overflows needed is a ***floating value***, then define a variable and increments it inside the overflow interrupt till we reach the desired number of overflows, then take the desired action.

# Asynchronous vs Synchronous Design

## Asynchronous vs Synchronous design

Synchronous basically means that you can only execute one thing at a time. Asynchronous means that you can execute multiple things at a time and you don't have to finish executing the current thing in order to move on to the next one.



# Asynchronous vs Synchronous Design

## Synchronous Design

In the case of synchronous design, the expectation is that there will be an immediate return of data. The application requests data and waits for it until a value is returned.

So in the synchronous design, function returns after all the required processing is finished, it may do some sort of busy wait for a specific operation like I/O.

## Asynchronous Design

In the case of Asynchronous design, the function returns immediately after submitting an operation request and the operation makes some sort of callback to your code or an Interrupt when it completes to give its completion status.

# Callback Function

Callback functions are an essential and often critical concept that developers need to create drivers or custom libraries.

A callback function is a reference to executable code that is passed as an argument to other code that allows a lower-level software layer to call a function defined in a higher-level layer. A callback allows a driver or library developer to specify a behavior at a lower layer but leave the implementation definition to the application layer.

A callback function is just a function pointer that is passed to another function as a parameter. In most instances, a callback will contain three pieces:

- The callback function
- A callback registration
- Callback execution

# Callback Function

Callback sequence diagram