Microprocessors

Fall 2020

7. External Interrupts and GPIO

Tentative Weekly Schedule

- Week x1 - Introduction to Course
- Week x2 - Architecture
- Week x3 - Assembly Language Introduction
- Week x4 - Assembly Language Usage, Memory and Faults
- Week x5 - Embedded C and Toolchain
- Week x6 - Exceptions and Interrupts
- **Week x7 - GPIO, External Interrupts and Timers**
- Week x8 - Timers
- Week x9 - Serial Communications I
- Week xA - Serial Communications II
- Week xB - Analog Interfacing
- Week xC - DMA
- Week xD - RTOS
- Week xE - Wireless Communications

Review: CMSIS

- **Cortex Microcontroller Software Interface Standard (CMSIS)** is a **software framework** to provide a **standardized** software interface to the processor features like interrupt and system control functions.
- Provided by ARM and targets most Cortex-M processors
- Can be used with different toolchains and IDEs
- Consist of multiple packages

Review: Interrupts

- **Hardware-triggered asynchronous software routine**
  - **Triggered** by hardware signal from peripheral or external device
  - **Asynchronous** - can happen anywhere in the program
  - **Software routine** runs in response to interrupt
- Fundamental mechanism of microcontrollers
  - Provides **efficient event-based processing** rather than polling
  - Provides **quick response to events** regardless of program state, complexity and location
  - Allows embedded systems to be **responsive** without an operating system

Review: Example C program

```
static int count = 0;

void buttonISR(void) {
    // Toggle LED
    GPIOC->ODR ^= (1 << 6);
    count++;
}

int main(void) {
    // Initialize LED and Buttons
    // Initialize interrupt

    while(1) {
        // Do nothing
    }
   return 0;
}
```

- **No calls** to the function
- Hardware **automatically jumps** there to **service** the interrupt
- If variables need to be **shared**, they should be declared **globally**
  - Also this requires careful considerations both in terms of optimization and race conditions

Review: Non-atomic shared data

- Fundamental problem is **race condition**
  Race Condition: Anomalous behavior due to unexpected critical dependence on the relative of timing of events. Result of example code depends on the relative timing of the read and write operations.
- Preemption enables ISR to interrupt other code and possibly **overwrite data**
- Must ensure **atomic (indivisible)** access to the object
- Native atomic object size depends on the **processor's instruction set** and word size
- Another way is to disable / enable interrupts within **critical code sections**
  Critical code section: A section of code which creates a possible race condition. Therefore a synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use

Q. What is happening to my SSD segments?

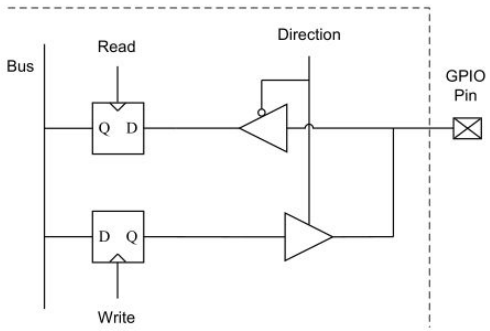
- LEDs work with current, and they require a limited amount of current
- Too low, and it will not work
- Too high, and it will wear out the LED
- Check the datasheet

General Purpose Input / Output (GPIO)

- For connecting various devices, displays, sensors and actuators, different types of interfaces exist on a microcontroller.
- To connect using these interfaces, microcontroller physical pins can be configured either as input or output.
- To support various communications and protocols, these pins can also be configured as **alternative functions** that we will discuss in the subsequent lectures.
- A general purpose input output pin can be configured either as a **digital output** or **digital input** and works by converting the associated **register value to voltage levels**. (i.e writing 1 translates to 3.3V, and 0 to 0V)

General Purpose Input / Output (GPIO)

- Below is a simple connection diagram about how a GPIO pin works. (i.e. two different level activated tri-state buffers.)



GPIO Peripheral registers: MODER

- Each GPIO pin can be configured from **MODER** as
  - **00** Digital input
  - **01** Digital output
  - **10** Alternate function
  - **11** Analog input (reset state)
- Each pin is associated with 2-bit field in **MODER**

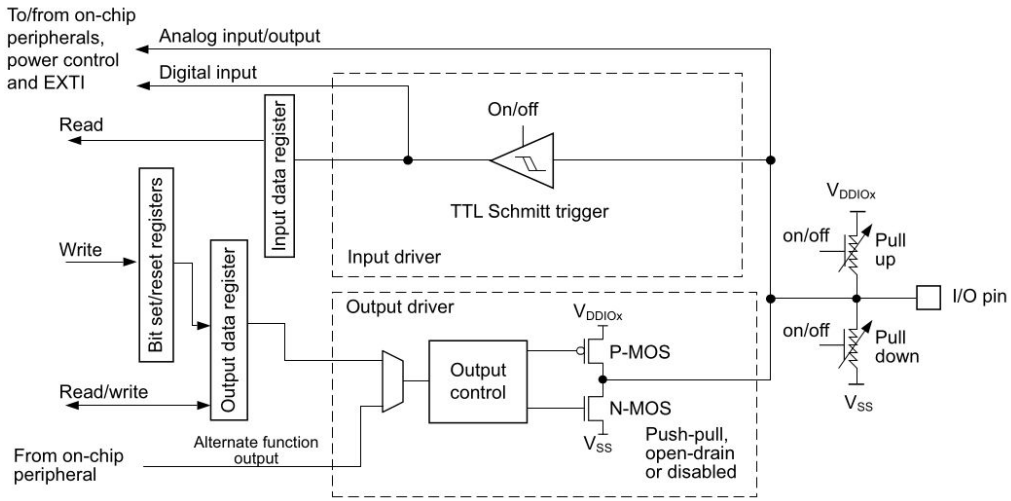
GPIO port mode register (GPIOx_MODER)
(x =A to D, F)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

GPIO Peripheral registers in STM32G0

- Each output pin can be configured to be a **push-pull** (reset state) or **open-drain** (**high-z**). (**OTYPER**)
- Each I/O output speed can be configured between low and high speeds. (**OSPEEDR**)
- Each I/O pin can be configured to have internal **pull-up** or **pull-down**. (**PUPDR**)
- Each output can be written using **ODR** and read using **IDR** registers.
- I/O pins can be chosen to work on alternate function using **AFRL** and **AFRH** registers. We will talk about and use these starting next week.

GPIO Peripheral in STM32G0

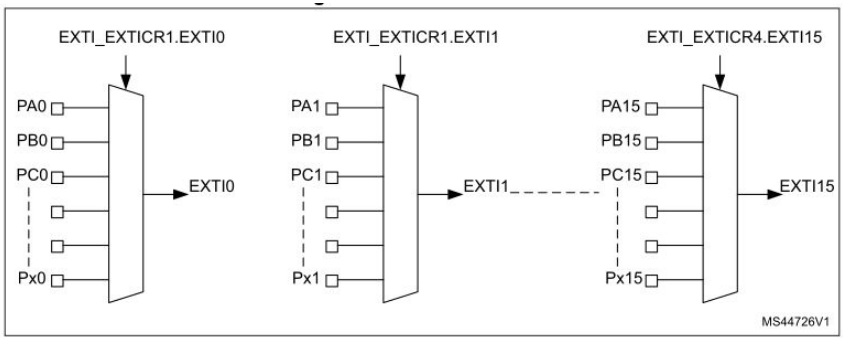


External Interrupts

- Sometimes, we want to be aware of a change in an external signal, and act on that change.
- This may be a simple button press, or a communication signal.
- It can also work as a wakeup source for the CPU from sleep modes.
- Microcontrollers usually include 1 or more interrupt capable pins for these type of situations.

EXTI GPIO Mux

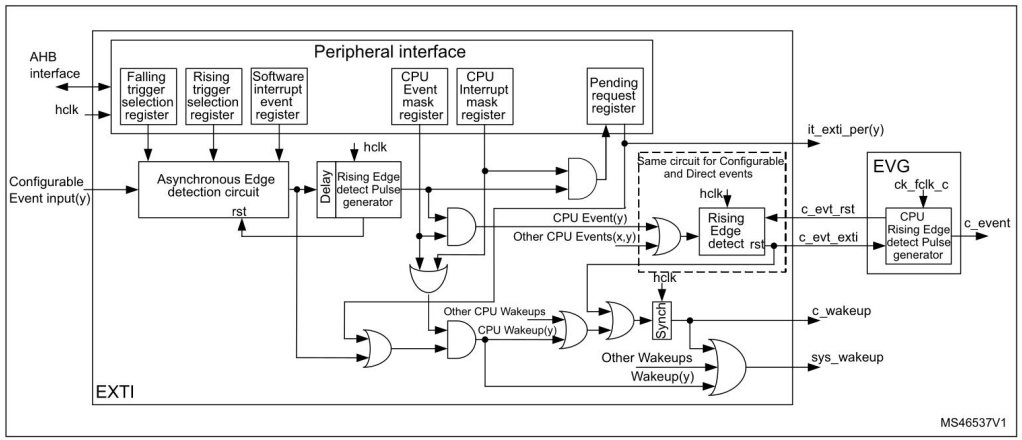
- Same pin number from different ports are MUXed to the same interrupt routine



EXTI Registers

- **RTSR1** Rising trigger selection register
- **FTSR1** Falling trigger selection register
- **SWIER1** Software interrupt event register - interrupts can be triggered by software
- **{RF}PR1** Rising / Falling pending register - triggered interrupts (hw/sw) can be cleared by writing 1
- **EXTICRx** Selection register. Each external interrupt has 8-bit fields for selection of the GPIO port (MUX control)
- **IMR** Mask register. Each external interrupt can be masked for operation

EXTI Operation



EXTI Configuration

```
EXTI->EXTICR[0] |= (2U << 8*1); // Choose PortC
MUX. 0 has 0-3, 1 has 4-7, 2 has 8-11, 3 has 12-15
EXTI->FTSR1 |= (1U << 1); // Falling edge on Px1
EXTI->IMR1  |= (1U << 1); // Mask Px1
```

Then set up NVIC for the associated interrupt request
External Interrupts are grouped into 3 (as well as the handlers)

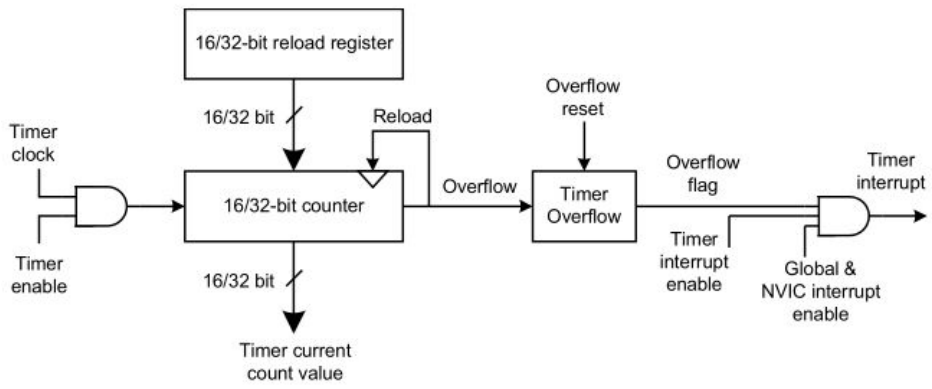
- `EXTI0_1_IRQn`
- `EXTI2_3_IRQn`
- `EXTI4_15_IRQn`

Check out startup_xxx code for exact names and organization

Timer Peripheral

- *Time keeping* is important concept for embedded systems, and usually **we want exact timing information** (especially between two events).
- Loading this task to the CPU makes it **hard to determine** the exact time, and **inefficient**.
- Microcontroller vendors include specific timer modules that are basically **dedicated counters** that will count up/down to/from given values and **indicate when the count is finished**.
- These modules can include various functionality such as count up/down, one-shot, continues count

Basic Timer block diagram w/ interrupt

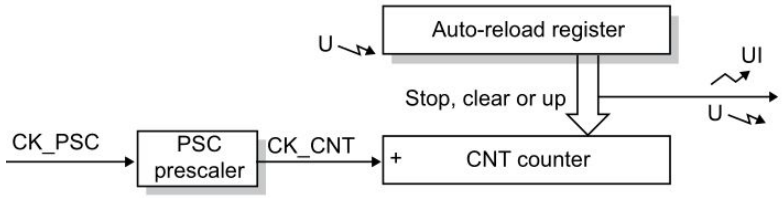


Timer / Counter

- Timer's clock source can be selected.
  - **Counter mode** - (**async**) count pulses which indicate events (e.g. number of parts passing through an assembly line)
  - **Timer mode** - (**sync**) count clock pulses
- Count **direction** can be fixed or selectable
- Count **value** can be read / written by MCU
- Count's **over/under flow action** can be selected
  - Generate an interrupt
  - Reload counter value and continue counting
  - Toggle output signal

Basic Timer operation

- Load start value from register
- Counter starts counting down with each clock pulse
  - Can also multiplied with **prescaler**
- When timer value reaches zero
  - Generate interrupt
  - Reload timer from register



Calculating MAX value

- To generate an interrupt every T seconds, we need to find the associated reload register value.
- Depends on the processor clock frequency and bus frequency that the Timer is connected to.
- For example, to create a timer with **300 ms period** (or interrupt rate) for a **processor running at 16 MHz**
  - Write `300 ms x 16 MHz = 4800000` to the reload register.
  - T * F should can be represented with timer peripheral bit count. (i.e. 32-bits)

Example: Stopwatch

- Let's say we want to implement a stopwatch.
- Measure time with `100 us` resolution
- Display elapsed time, updating screen every `10 ms`
- Increment counter every `100 us` by attaching to a timer.
  - Assuming we have `16 MHz` clock, reload value is `16M x 100us - 1 = 1599`
- Displaying every `100 us` will be a little rough and not really necessary for our eyes, so let's update the display every `10 ms`.
  - How would you implement update?

Timers in STM32G0

There are different timer related modules in G0

Inside the ARM core

- SysTick Timer

As peripherals

- General-Purpose Timers
- Advanced-control Timers - various additional functionalities such as input capture, output compare and PWM
- Watchdog Timers
  - Independent WDG
  - Window WDG

We will talk about these later in the course

The SysTick Timer

- In order to allow an Operating System to carry out periodical *context switching* to support multi-tasking, **the program execution must be interrupted** by a hardware device like a timer.
- When the timer interrupt is triggered, an exception handler that handles **OS task scheduling** is executed
  - The handler might also carry out other OS maintenance tasks.
- For Cortex-M processors, a simple timer called **SysTick** is included **inside the processor** to perform the function of generating this periodic interrupt request.
- For non-OS uses, this acts as a generic timer.

The SysTick Timer

- Integrated to NVIC and generates **SysTick exception (#15)**
- Basic **24-bit down counter**
- **Reloads automatically** after reaching zero and the reload value is programmable
- for non-OS uses, it can be used as *basic timekeeping*, *timing measurement*, or *as an interrupt source* for tasks that need to be executed periodically.
- The **exception generation is programmable**, and if disabled, it can be used with polling method, i.e. by checking the current value of the counter

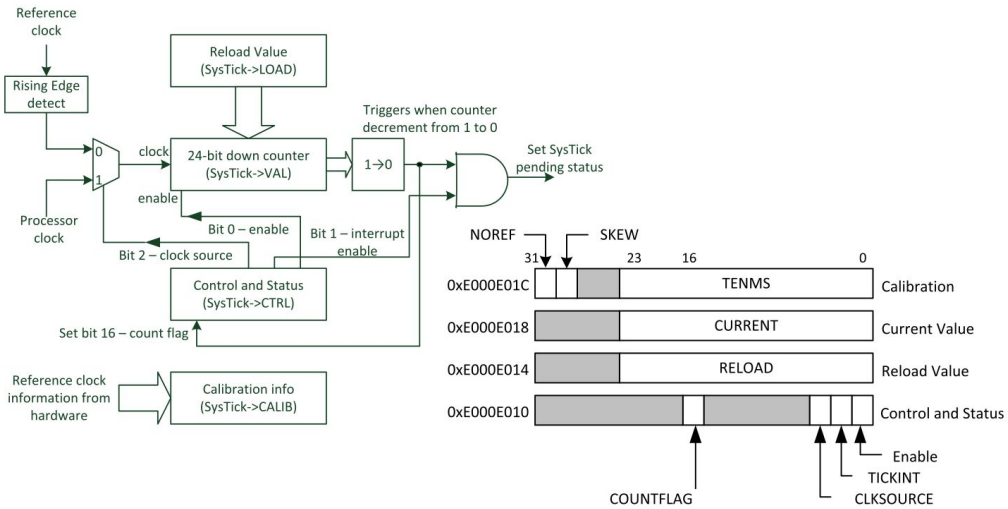
SysTick Registers

Register	CMSIS Name	Details	Address
SysTick Control and Status Register	SysTick->CTRL	Table 10.2	0xE000E010
SysTick Reload Value Register	SysTick->LOAD	Table 10.3	0xE000E014
SysTick Current Value Register	SysTick->VAL	Table 10.4	0xE000E018
SysTick Calibration Value Register	SysTick->CALIB	Table 10.5	0xE000E01C

There are 4 SysTick related registers

- **CTRL** - Holds count flag, clock source, exception enable and SysTick enable fields
- **LOAD** - Holds reload value after reaching zero
- **VAL** - Holds current counter value
- **CALIB** - Holds calibration data (skew, ref, vs)

SysTick Operation

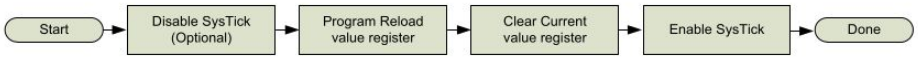


SysTick Configuration

```
SysTick->CTRL = 0;   // Disable SysTick
SysTick->LOAD = 999; // Count down from 999 to 0
SysTick->VAL  = 0;   // Clear current value
SysTick->CTRL = 0x7; // Enable Systick, exception,
                     //   and use processor clock
```

Alternatively, using CMSIS, you can invoke

```
SysTick_Config(1000);
```



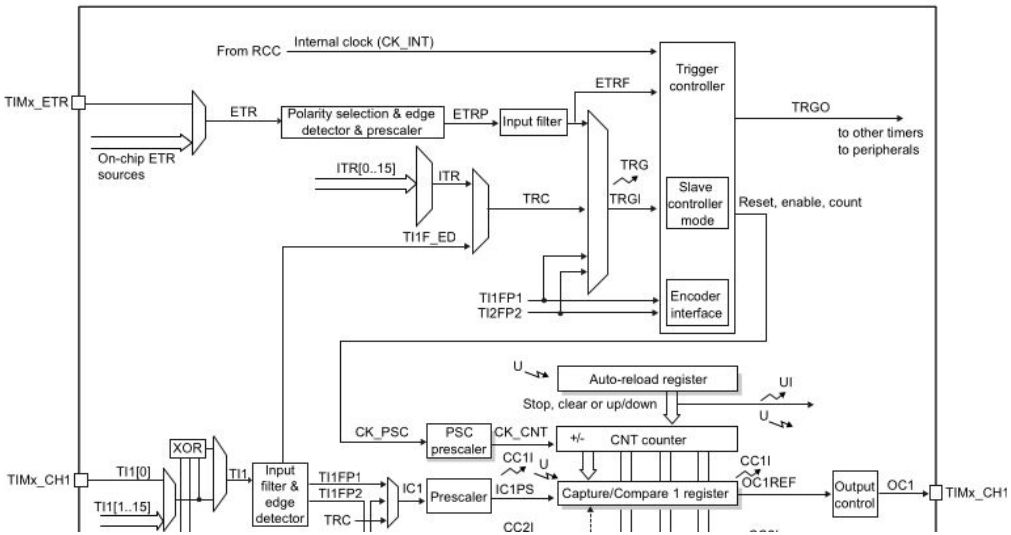
Timers in STM32G0

Timer type	Timer	Counter resolution	Counter type	Maximum operating frequency	Prescaler factor	DMA request generation	Capture/ compare channels	Comple-mentary outputs
Advanced-control	TIM1	16-bit	Up, down, up/down	128 MHz	Integer from 1 to 2^{16}	Yes	4	3
General-purpose	TIM2	32-bit	Up, down, up/down	64 MHz	Integer from 1 to 2^{16}	Yes	4	-
	TIM3	16-bit	Up, down, up/down	64 MHz	Integer from 1 to 2^{16}	Yes	4	-
	TIM14	16-bit	Up	64 MHz	Integer from 1 to 2^{16}	No	1	-
	TIM16 TIM17	16-bit	Up	64 MHz	Integer from 1 to 2^{16}	Yes	1	1
Low-power	LPTIM1 LPTIM2	16-bit	Up	64 MHz	2^n where n=0 to 7	No	N/A	-

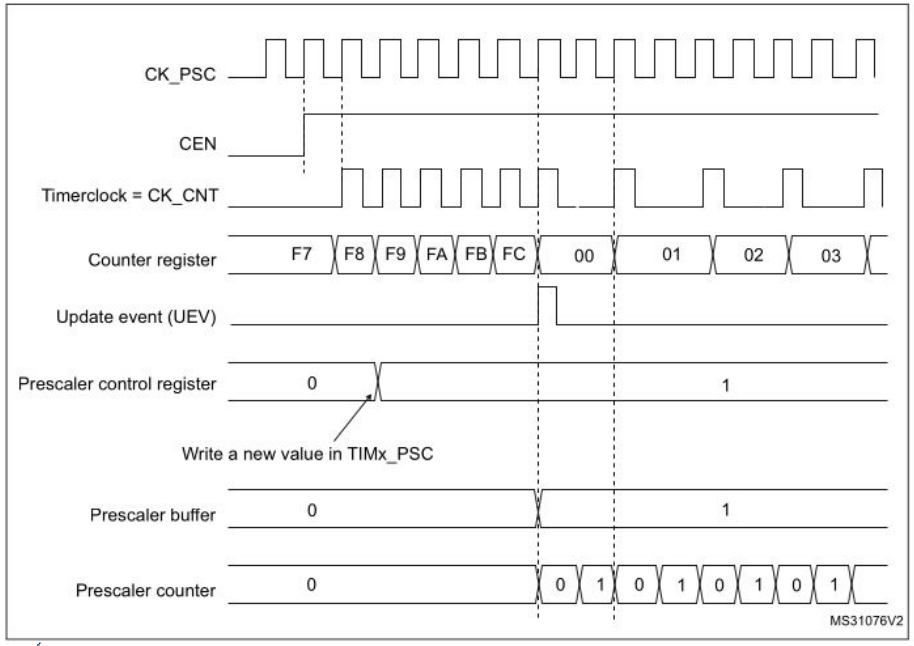
Timers in STM32G0

- These timers can be used for
  - standard up/down counter mode
  - measuring pulse lengths of input signals (input capture)
  - generating output waveforms (output compare)
  - pulse-width modulation generation
  - interrupt / DMA generation
- They can be 16 / 32 bits, so be careful
- Include synchronization circuit to control the timer with external signals and to connect several timers together

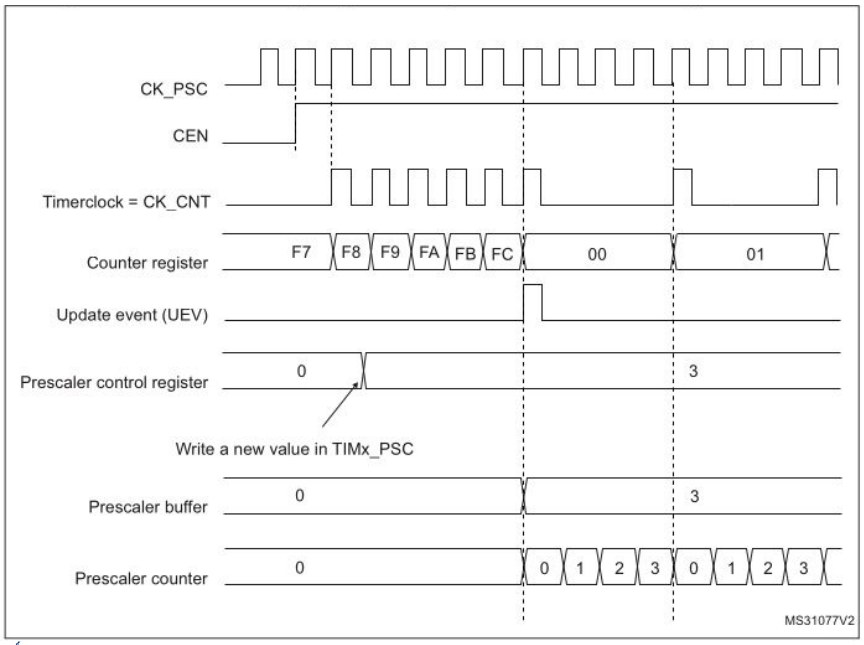
STM32G0 - General-purpose timer block



Basic counter with prescaler 1 -> 2



Basic counter with prescaler 1 -> 4



This week

- Read Chapter 10.3 from Yiu
- Read Section 6 from RM0444
- Lab3 assigned and due 30th of November
- HW4 incoming and due 9th of December
- Project 2 assigned and due on 21st December