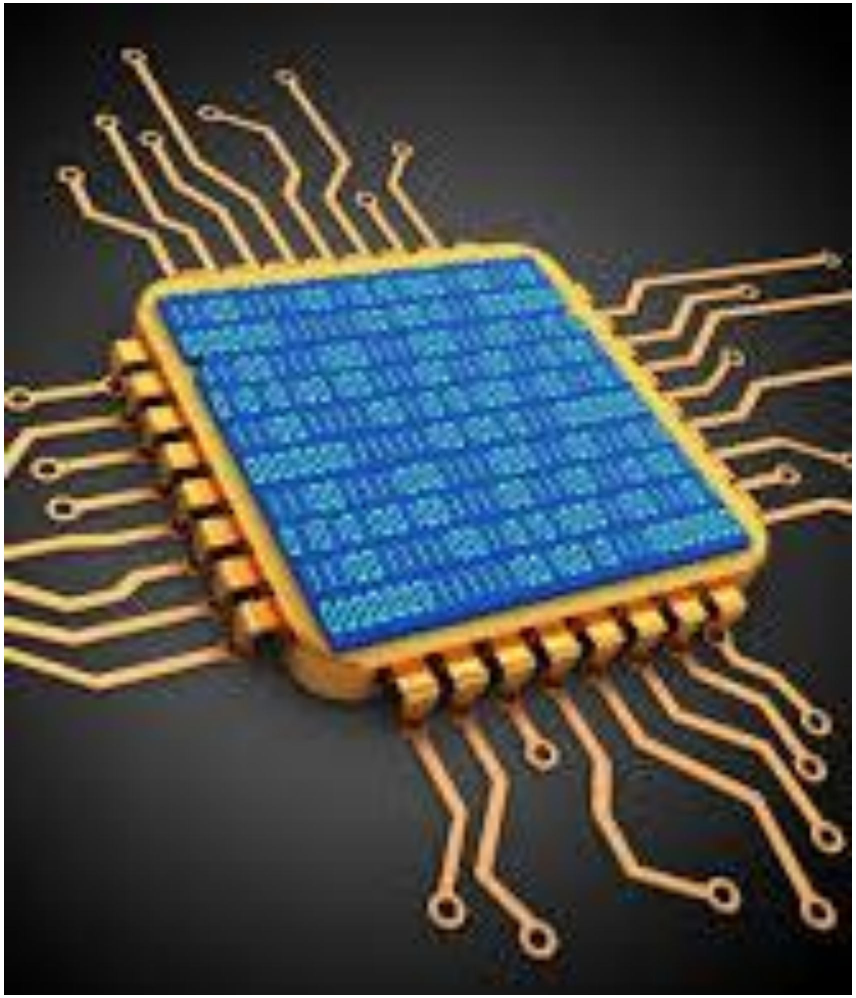
EMBEDDED PROGRAMMING

Andrea Calanca

Embedded Programming

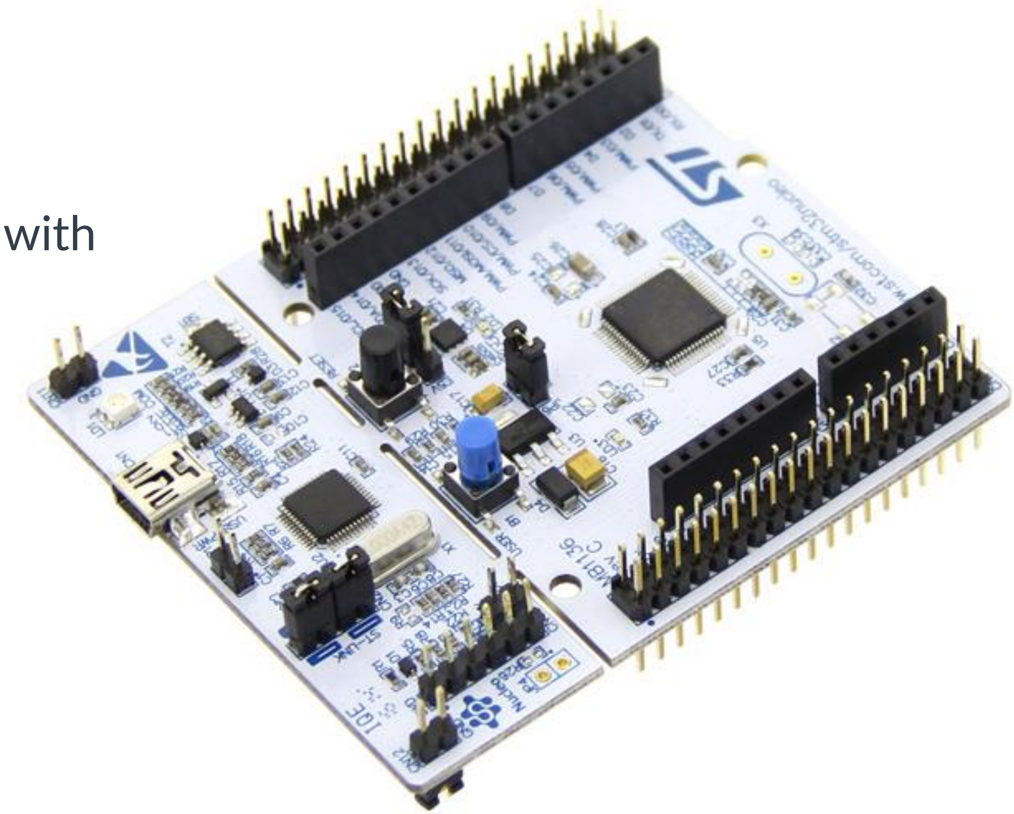
A type of programming that don't operate on traditional operating systems (the way that full-scale laptop computers and mobile devices do)

Embedded programming is also known as embedded software development or embedded systems programming.



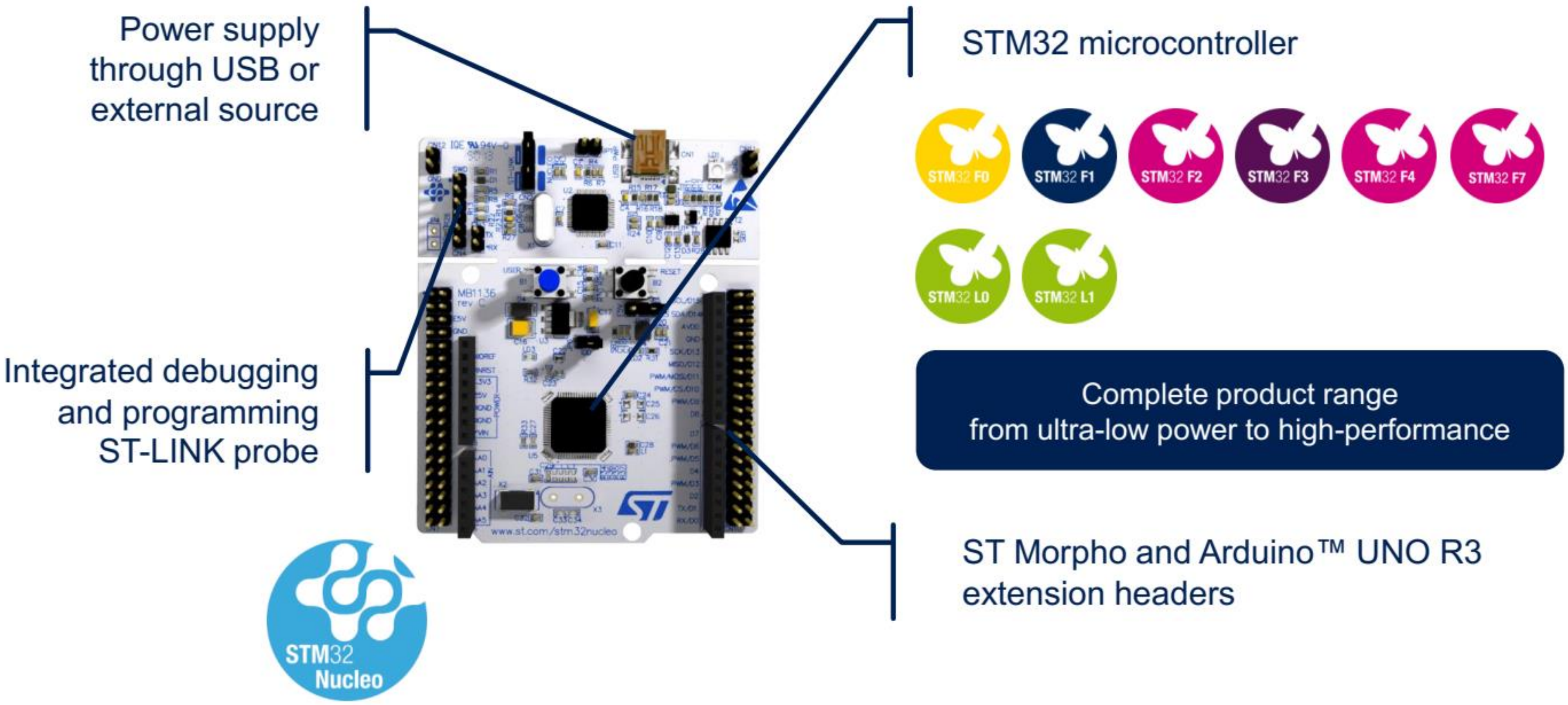
ST NUCLEO Boards

- STM32F446RET6 in LQFP64 package
- ARM®32-bit Cortex®-M4 CPU with FPU
- 180 MHz max CPU frequency
- VDD from 1.7 V to 3.6 V
- 512 KB Flash
- 128 KB SRAM
- GPIO (50)
- 12-bit ADC (3) with 16 channels
- 12-bit DAC with 2 channels
- RTC, Timers, I2C, USART, SPI, USB OTG Full Speed, …



https://os.mbed.com/platforms/ST-Nucleo-F446RE/

ST NUCLEO Boards



Datasheet

STM32 Nucleo-64 boards User manual:
https://www.st.com/content/ccc/resource/technical/document/user_manual/98/2e/fa/4b/e0/82/43/b7/DM00105823.pdf/files/DM00105823.pdf/jcr:content/translations/en.DM00105823.pdf

STM32F446 Datasheet:
https://www.st.com/resource/en/datasheet/stm32f446ze.pdf

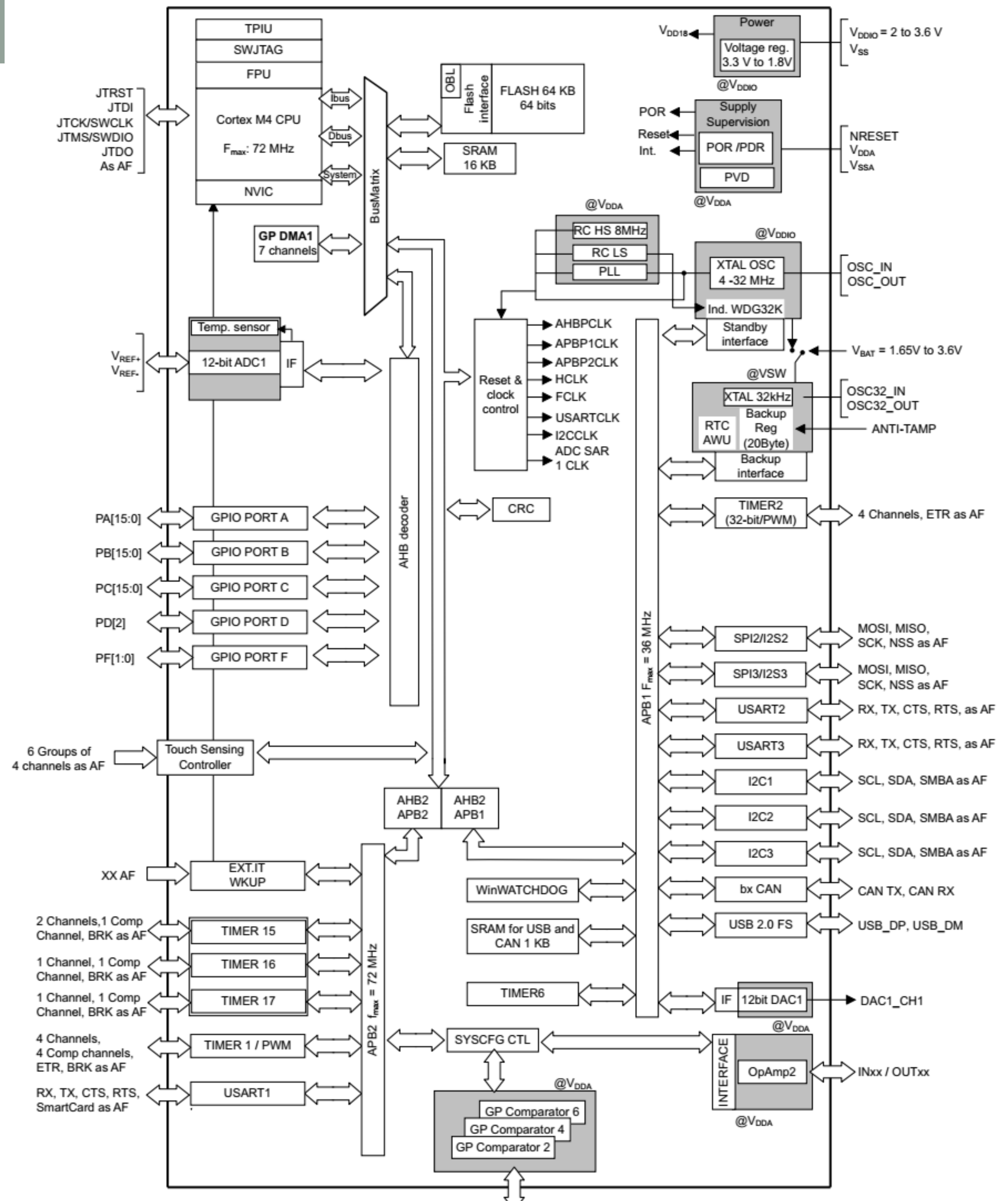
If you don’t know something, **search it** on the datasheet (or on the user manual).

µProcessor Datasheet (example)

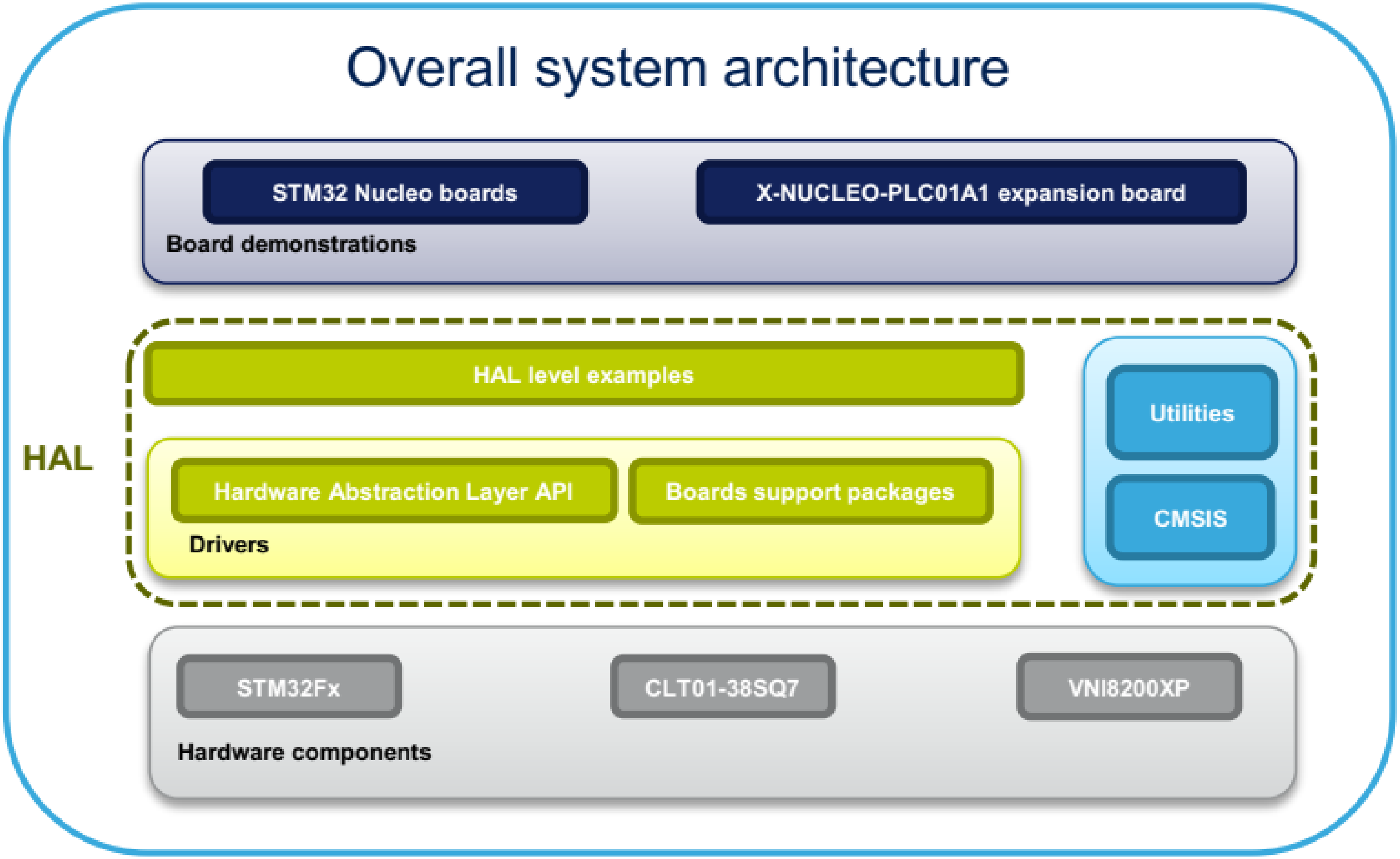
Table 2. STM32F302x6/8 device features and peripheral counts

Peripheral		STM32F302Kx		STM32F302Cx		STM32F302Rx	
Flash (Kbytes)		32	64	32	64	32	64
SRAM (Kbytes)		16					
Timers	Advanced control	1 (16-bit)					
	General purpose	3 (16-bit) 1 (32 bit)					
	Basic	1					
	SysTick timer	1					
	Watchdog timers (independent, window)	2					
	PWM channels (all)[1]	16		18			
	PWM channels (except complementary)	10		12			
Comm. interfaces	SPI/I2S	2					
	I^2C	3					
	USART	2		3			
	USB 2.0 FS	1					
	CAN 2.0B	1					
GPIOs	Normal I/Os (TC, TTa)	9		20		26	
	5-Volt tolerant I/Os (FT, FT1)	15		17		25	
DMA channels		7					
Capacitive sensing channels		13		17		18	
12-bit ADC Number of channels		1 8		1 11		1 15	
12-bit DAC channels		1					
Analog comparator		2		3		3	
Operational amplifier		1					
CPU frequency		72 MHz					
Operating voltage		2.0 to 3.6 V					

µProcessor Datasheet (example)

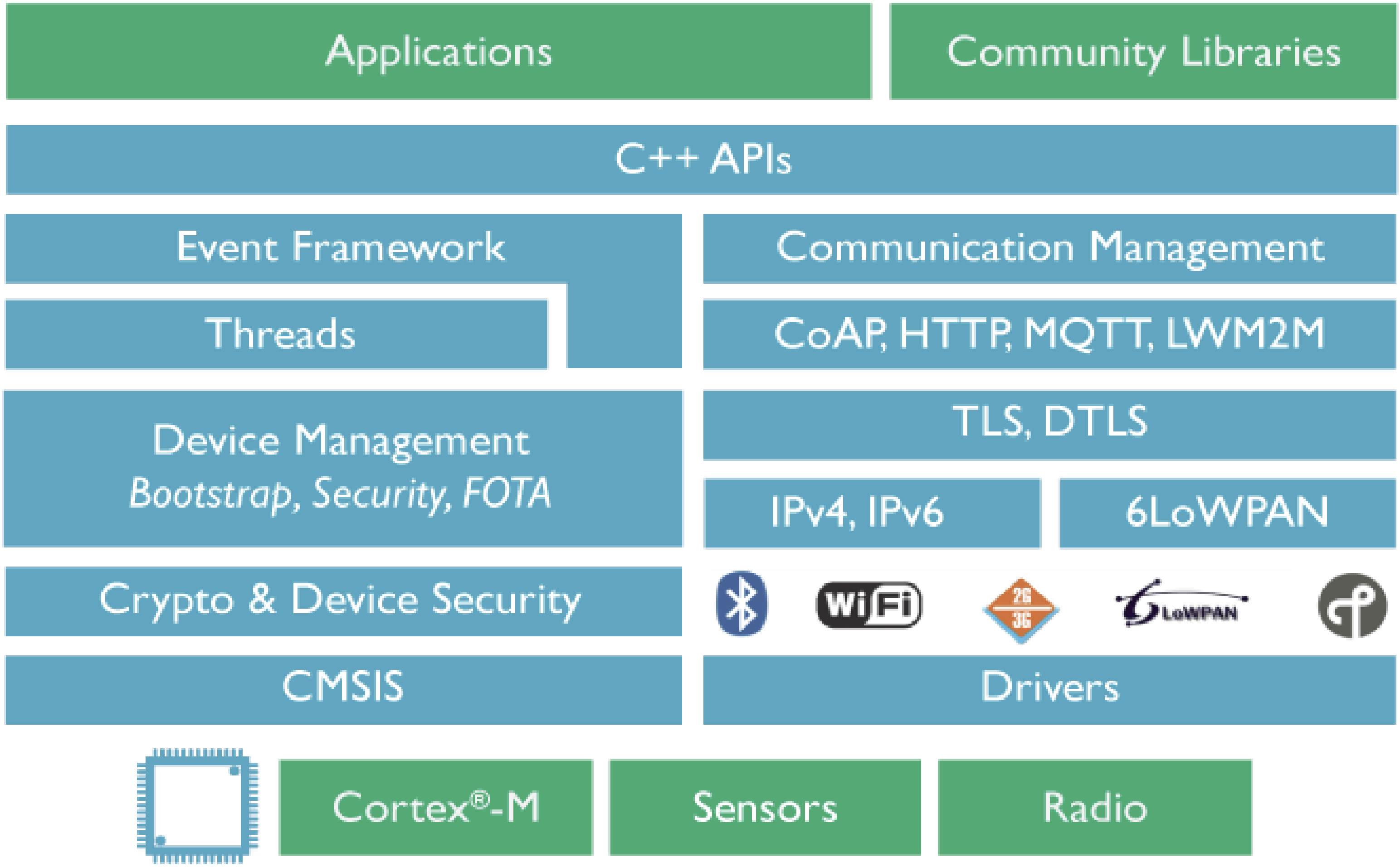


Hardware Abstraction Layer

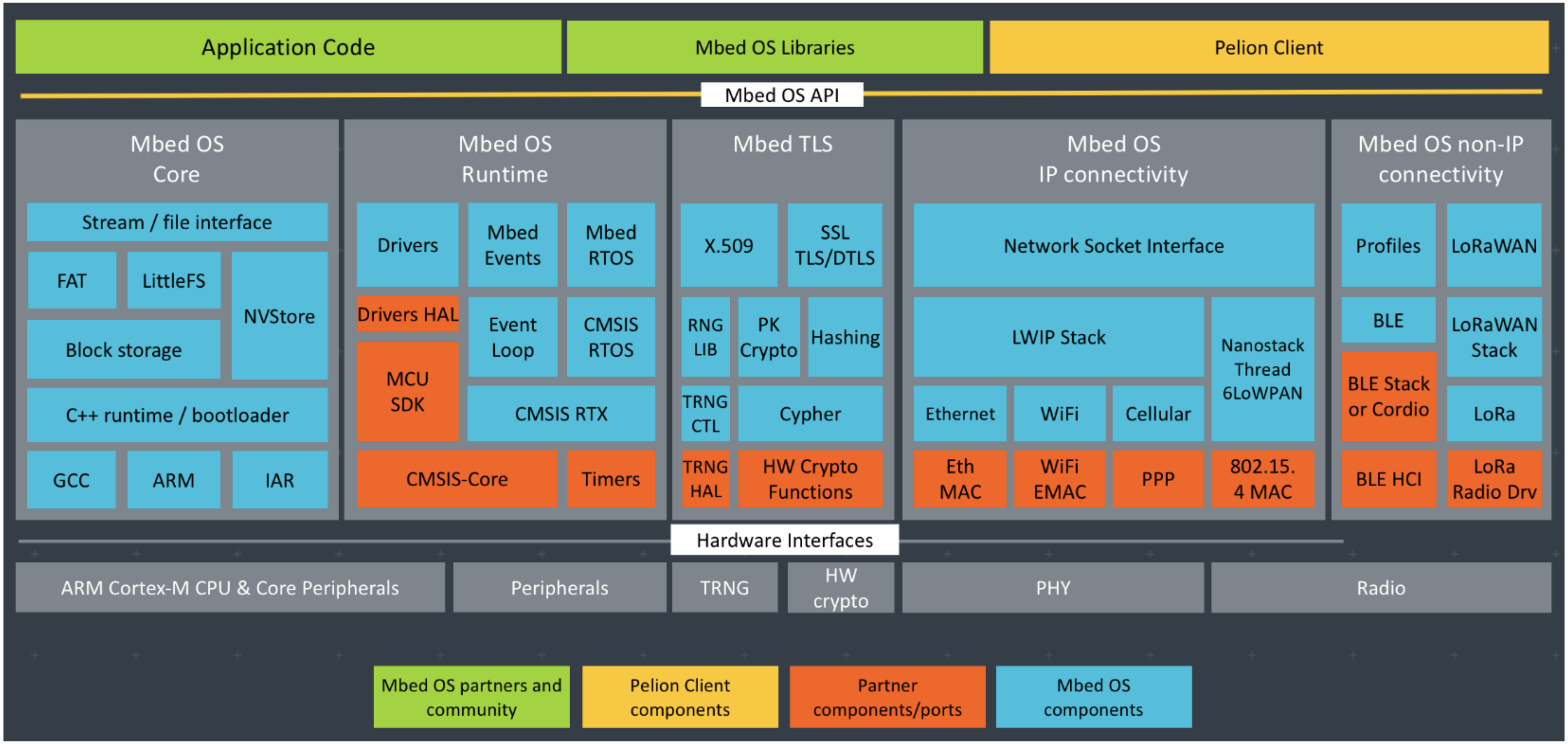


Mbed OS

https://www.mbed.com/en/



Mbed OS

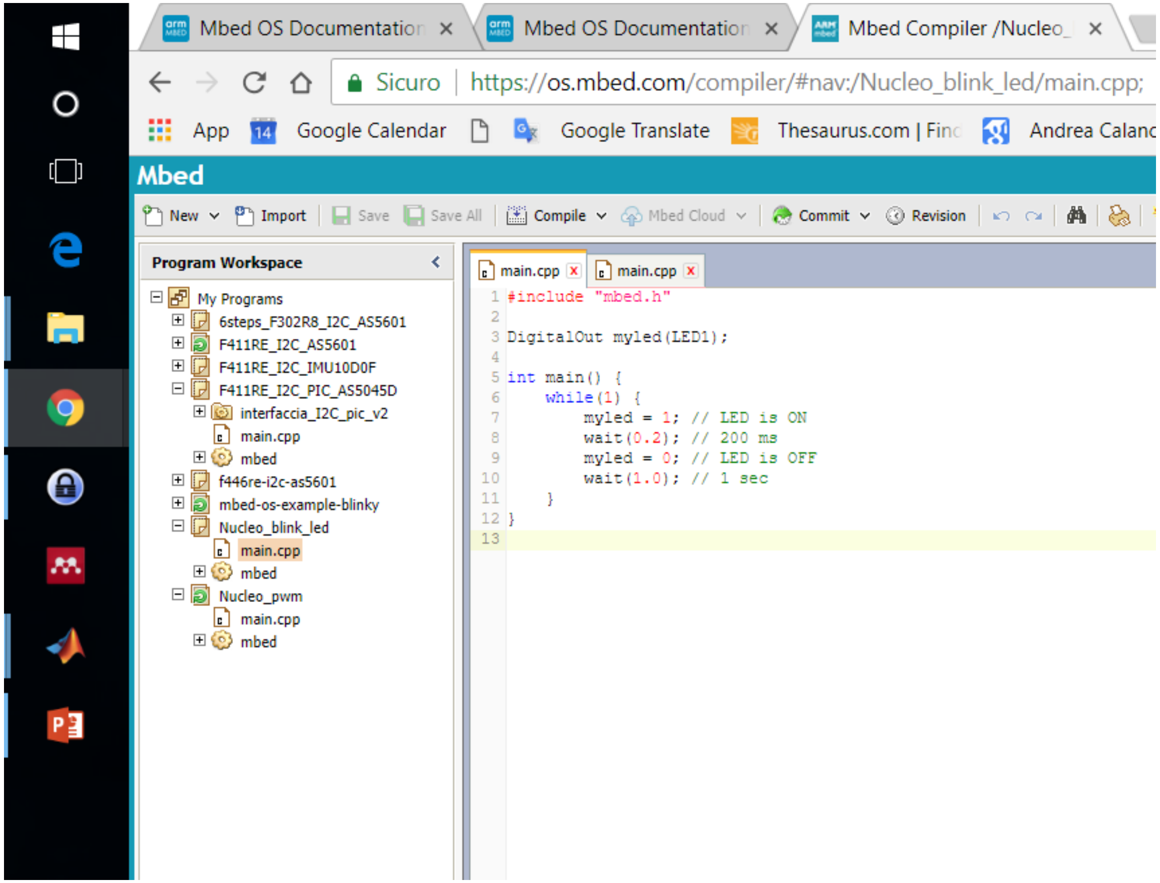

Application Code
Mbed OS Libraries
Pelion Client
Mbed OS API
Mbed OS Core
Stream / file interface
FAT
LittleFS
NVStore
Block storage
C++ runtime / bootloader
GCC
ARM
IAR
Mbed OS Runtime
Drivers
Mbed Events
Mbed RTOS
Drivers HAL
Event Loop
CMSIS RTOS
MCU SDK
CMSIS RTX
CMSIS-Core
Timers
Mbed TLS
X.509
SSL TLS/DTLS
RNG LIB
PK Crypto
Hashing
TRNG CTL
Cypher
TRNG HAL
HW Crypto Functions
Mbed OS IP connectivity
Network Socket Interface
LWIP Stack
Nanostack Thread 6LoWPAN
Ethernet
WiFi
Cellular
Eth MAC
WiFi EMAC
PPP
802.15.4 MAC
Mbed OS non-IP connectivity
Profiles
LoRaWAN
BLE
LoRaWAN Stack
BLE Stack or Cordio
LoRa
BLE HCI
LoRa Radio Drv
Hardware Interfaces
ARM Cortex-M CPU & Core Peripherals
Peripherals
TRNG
HW crypto
PHY
Radio
Mbed OS partners and community
Pelion Client components
Partner components/ports
Mbed OS components

Environment (online)

Online IDE provided by Mbed **Mbed Online Compiler**.
A lightweight online C/C++ IDE that is preconfigured to let you quickly write programs, compile and download them to run on your mbed Microcontroller.

1. Load the IDE on the browser
2. download the complier program (.bin)
3. **manually upload** on the board (like if it's a USB pendrive).

We will not use this.



https://os.mbed.com/handbook/mbed-Compiler

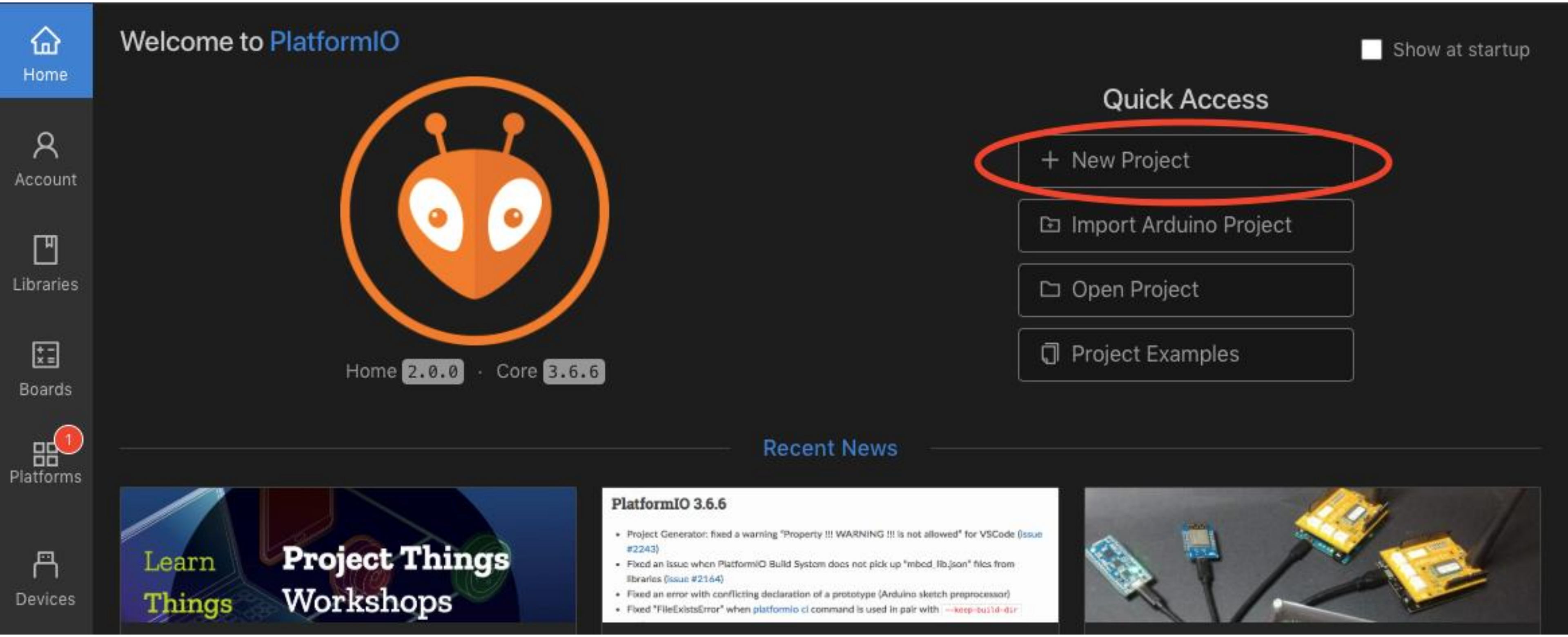
Environment

We will use **Visual Studio Code** with the **Platformio plugin**.

https://code.visualstudio.com
https://platformio.org/platformio-ide

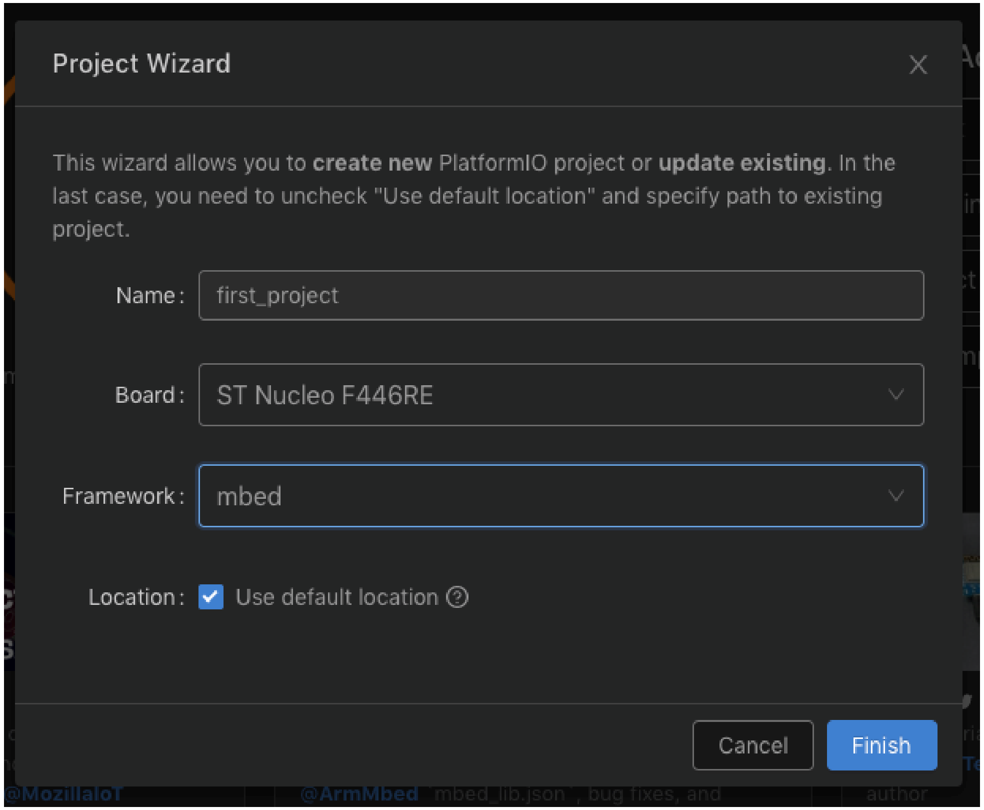
New project

Let’s start with the first project.
In the PIO home, choose **New Project**.



New project

In the Project Wizard write the name of the project and select the **board** (F446RE) and the **framework** (mbed).



New project

One important file is the *platformio.ini*.
Here you can change the target board, the framework and other stuffs.
Important: add the `upload_protocol` like in the figure if you want to upload the program on a real board.

platformio.ini ×

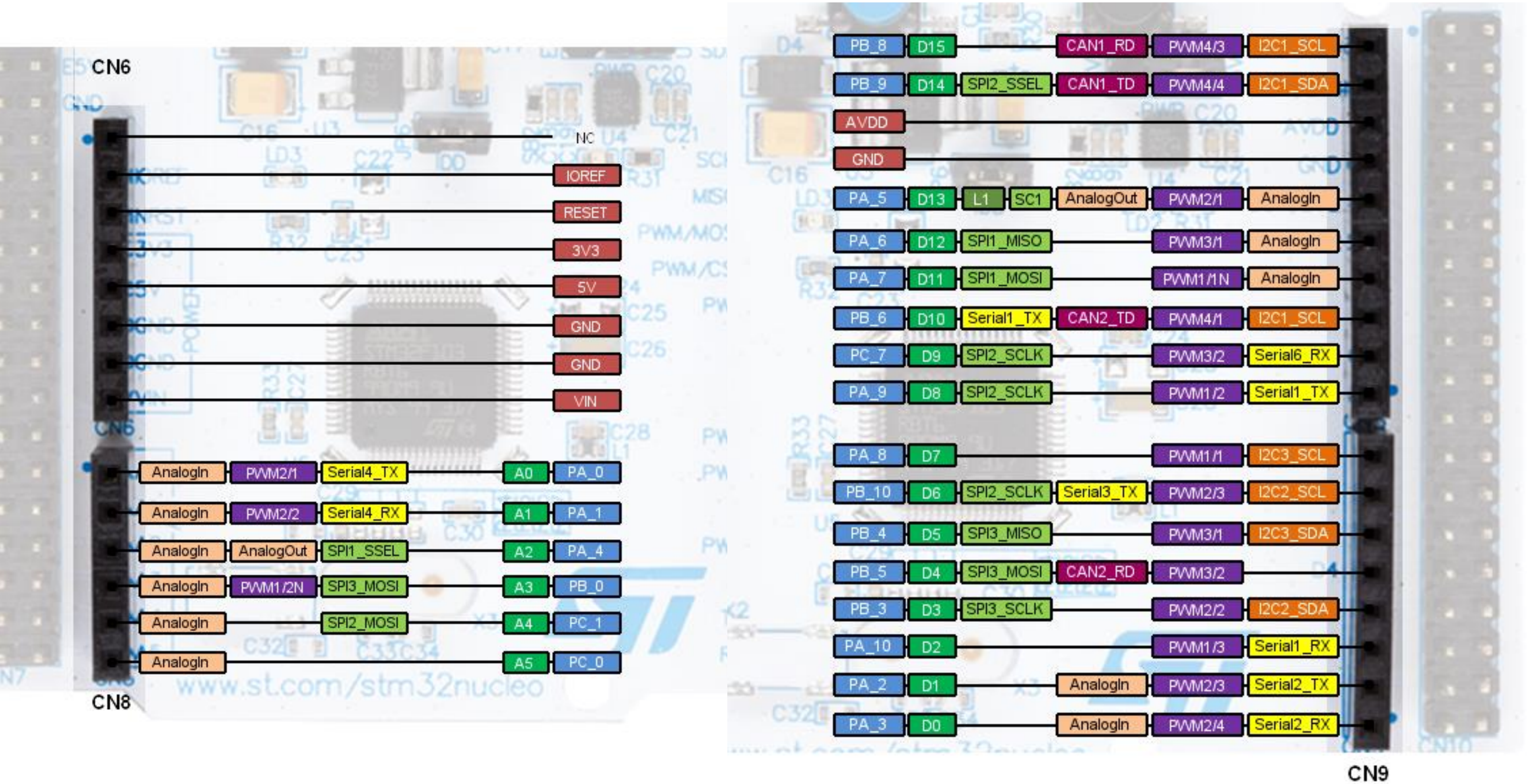
```ini
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:nucleo_f446re]
platform = ststm32
board = nucleo_f446re
framework = mbed
upload_protocol = mbed

```

Mbed API Documentation

You can find the API documentation in the link below.
The actual latest version is 6:
https://os.mbed.com/docs/mbed-os/v6.0/apis/index.html

F446RE pinout:
https://os.mbed.com/platforms/ST-Nucleo-F446RE/#nucleo-pinout

Pinout Example



DigitalIn and DigitalOut

DigitalIn

- Use the DigitalIn interface to read the value of a digital input pin. The logic level is either 1 or 0.
- You can use any of the numbered Arm Mbed pins can be used as a DigitalIn.
- https://os.mbed.com/docs/mbed-os/v5.12/apis/digitalin.html

DigitalOut

- Use the DigitalOut interface to configure and control a digital output pin by setting the pin to logic level 0 or 1.
- https://os.mbed.com/docs/mbed-os/v5.12/apis/digitalout.html

AnalogIn

- Use the AnalogIn API to read an external voltage applied to an analog input pin.

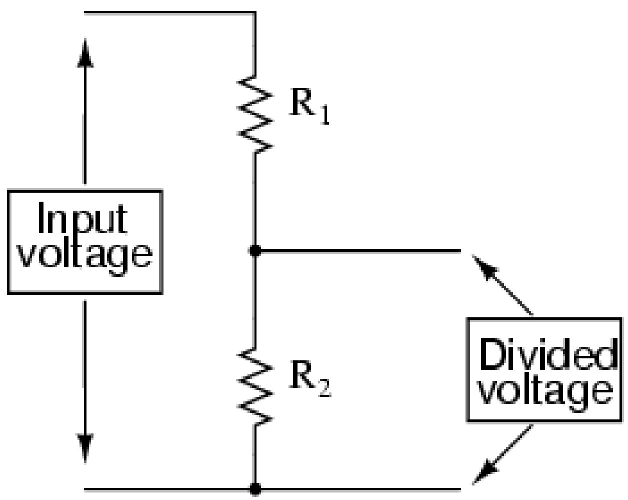
- AnalogIn() reads the voltage as a fraction of the system voltage. The value is a floating point from 0.0(VSS) to 1.0(VCC). For example, if you have a 3.3V system and the applied voltage is 1.65V, then AnalogIn() reads 0.5 as the value.

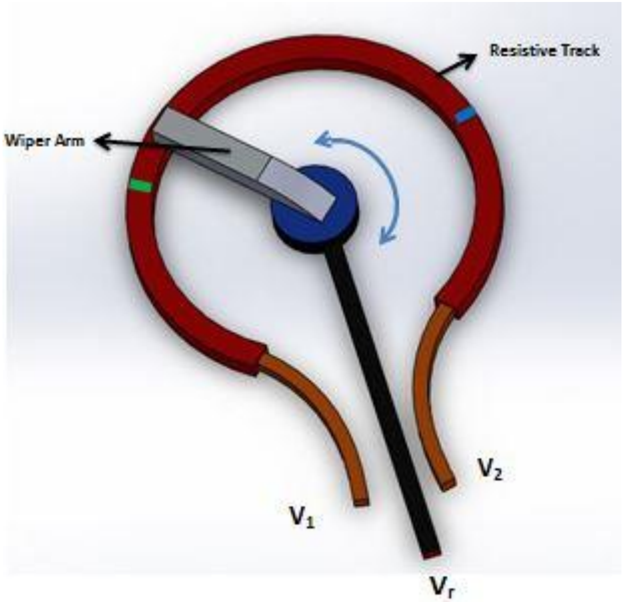
Note: Only certain pins are capable of making these measurements, so check the pinmap of your board for compatible pins.

Remember to use the API documentation:
https://os.mbed.com/docs/mbed-os/v5.12/apis/analogin.html

AnalogIn - Test

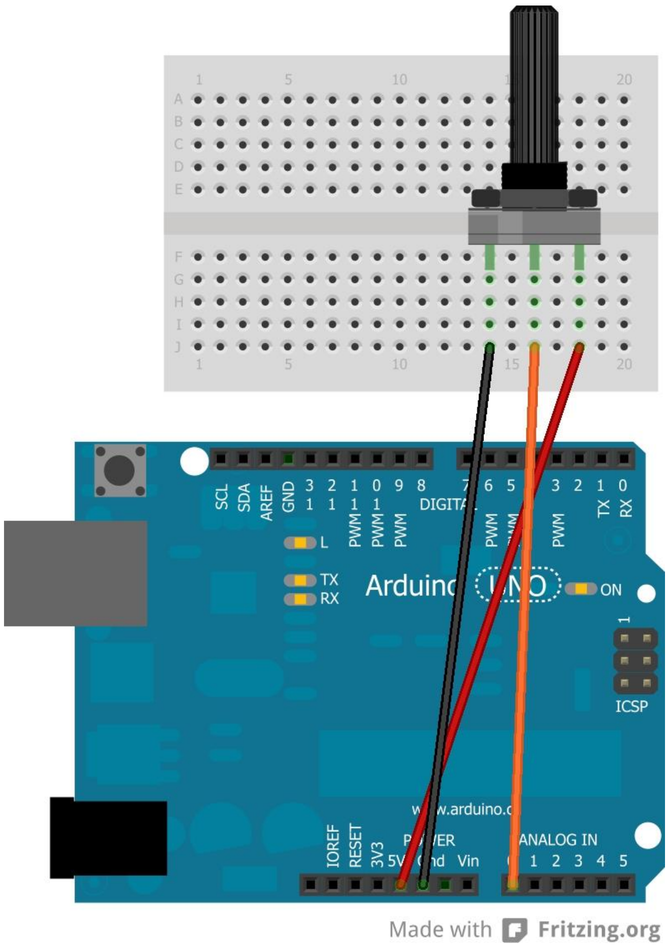
- Potentiometer
  - Simple and low cost
  - Can have noise at high frequencies
  - Dissipative (choose high R!!)
  - Deterioration because of sliding friction





$$P = \frac{V^2}{R}$$

AnalogIn - Test




```
#include "mbed.h"

AnalogIn input(A0);

int main() {
    uint16_t samples[1024];

    for(int i=0; i<1024; i++) {
        samples[i] = input.read_u16();
        wait(0.001f);
    }

    printf("Results:\n");
    for(int i=0; i<1024; i++) {
        printf("%d, 0x%04X\n", i, samples[i]);
    }
}
```

AnalogOut

- Use the AnalogOut interface to set the voltage of an analog output pin as a floating point number from 0.0 (VSS) to 1.0 (VCC).

Note: Not all pins are capable of being AnalogOut, so check the pinout for your board.

DigitalInOut

Use the DigitalInOut interface as a bidirectional digital pin:

- Read the value of a digital pin when set as an input().
- Write the value when set as an output().

You can use any of the numbered Arm Mbed pins as a DigitalInOut.

Homework #1

- Connect a potentiometer
- Make a LED blinking with a frequency set by the potentiometer from 1Hz to 10Hz

PwmOut

Use the PwmOut interface to control the frequency and duty cycle of a PWM signal.

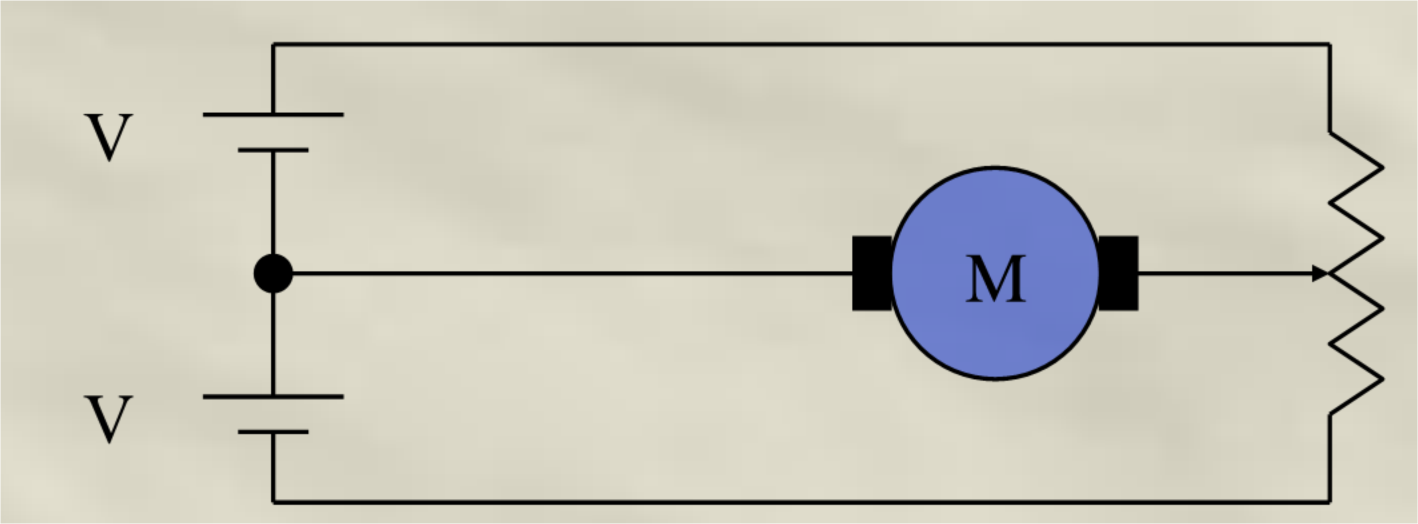
Tips:

- Set the cycle time first, and then set the duty cycle using either a relative time period via the write() function or an absolute time period using the pulsewidth() function.
- The default period is 0.020s, and the default pulse width is 0.

Power Electronics (DC Motors)

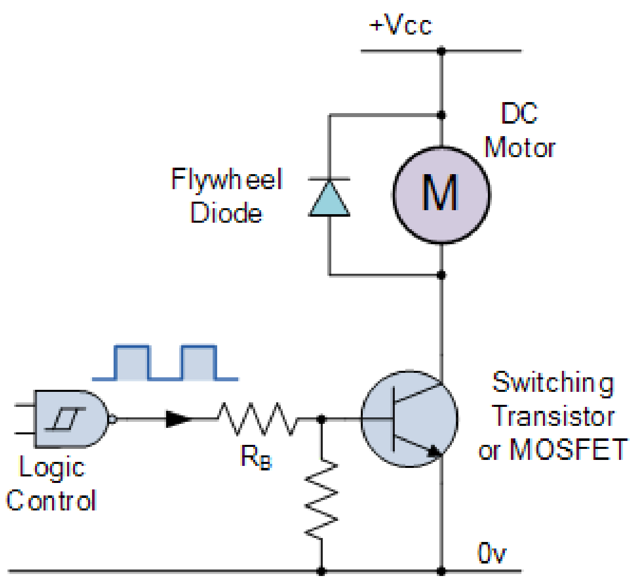
Voltage Control

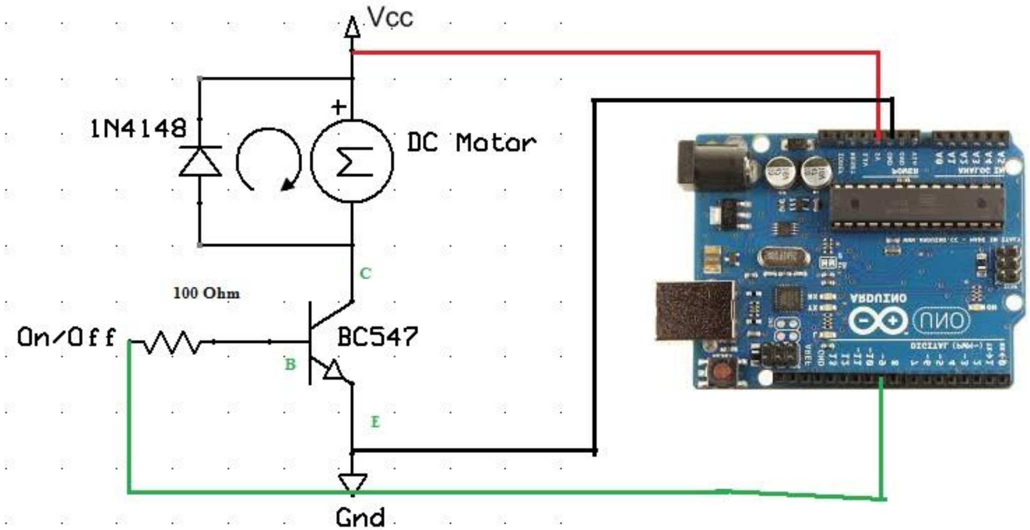
- The circuits below can apply a desired voltage but implies power dissipation



Power Electronics

- Use a switching mechanism (i.e. on-off transistor) to partialize the current.
- Use a desired logic to switch on and off the transistor

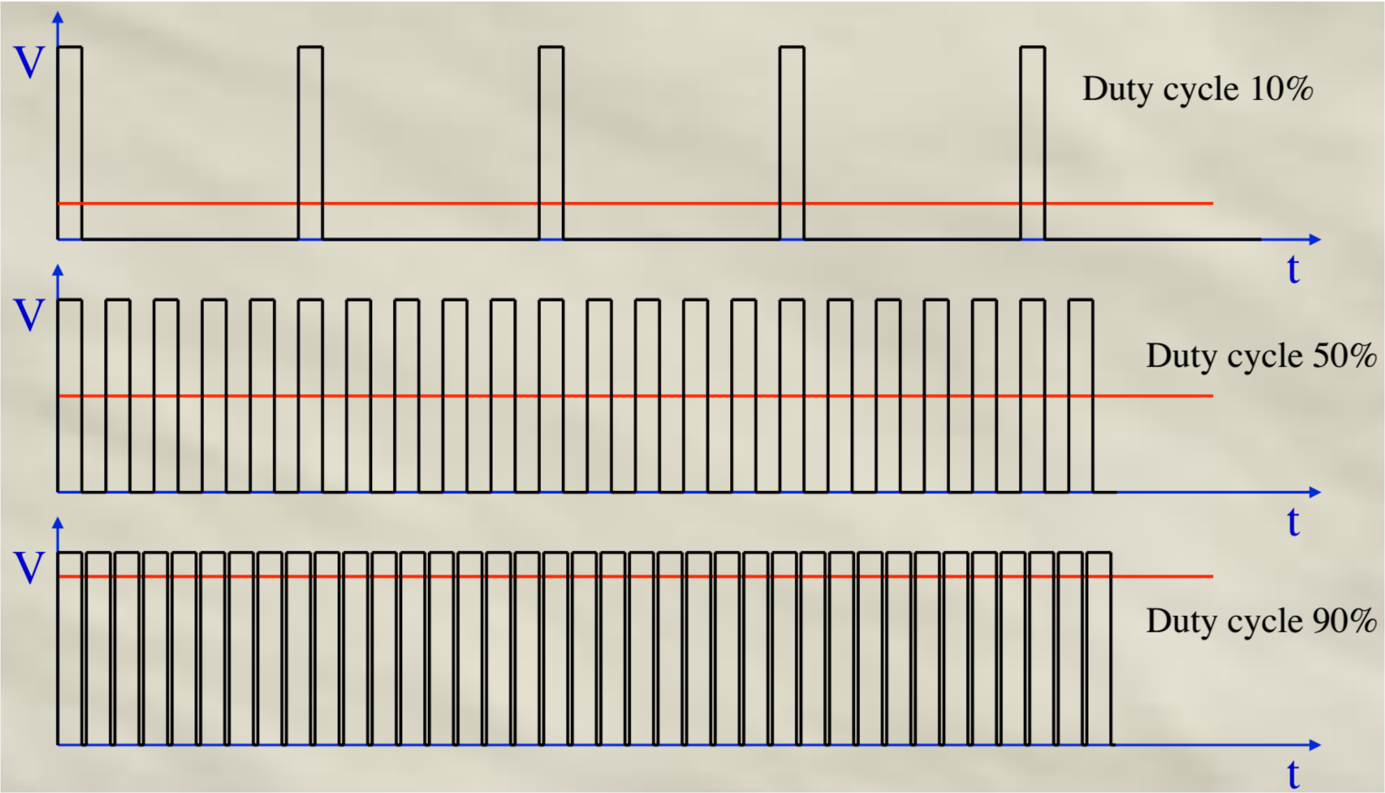




Power Electronics

(Voltage) Control

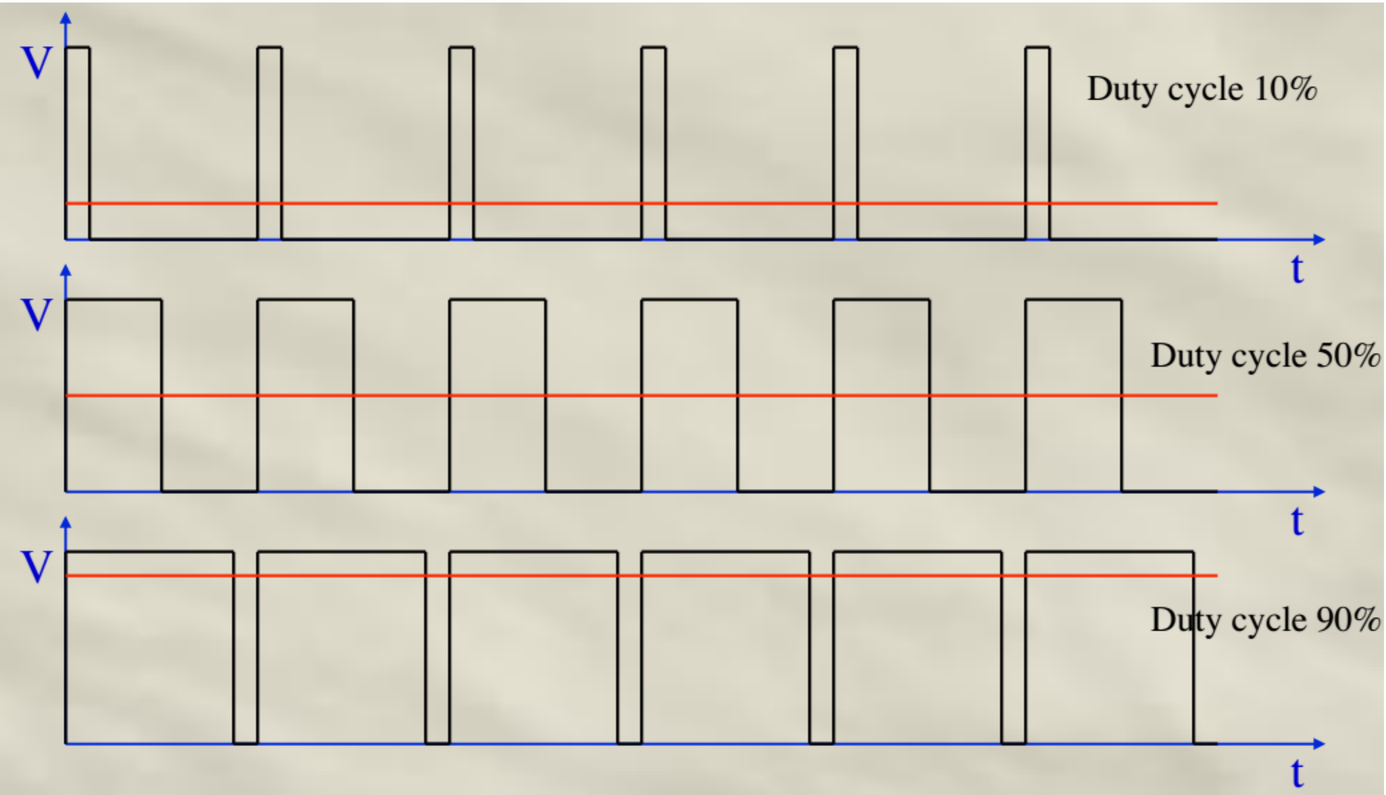
- Logic: Pulse Frequency Modulation



Power Electronics

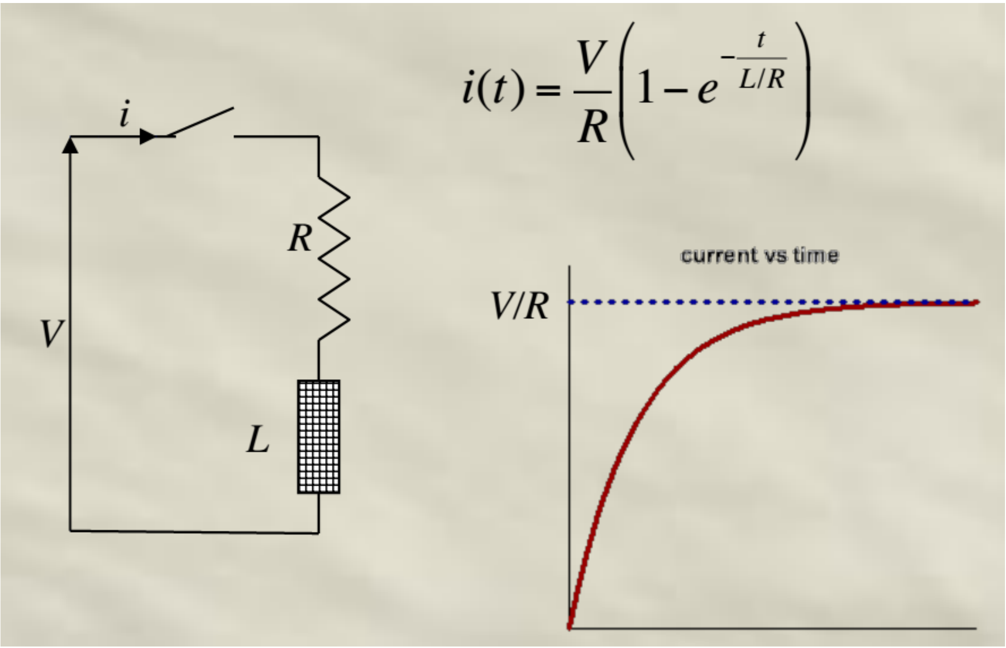
(Voltage) Control

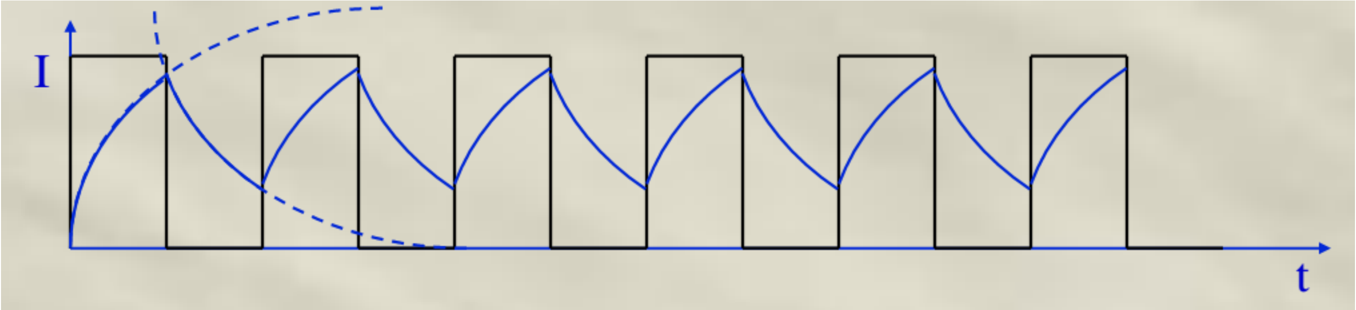
- Logic: Pulse Width Modulation



Power Electronics

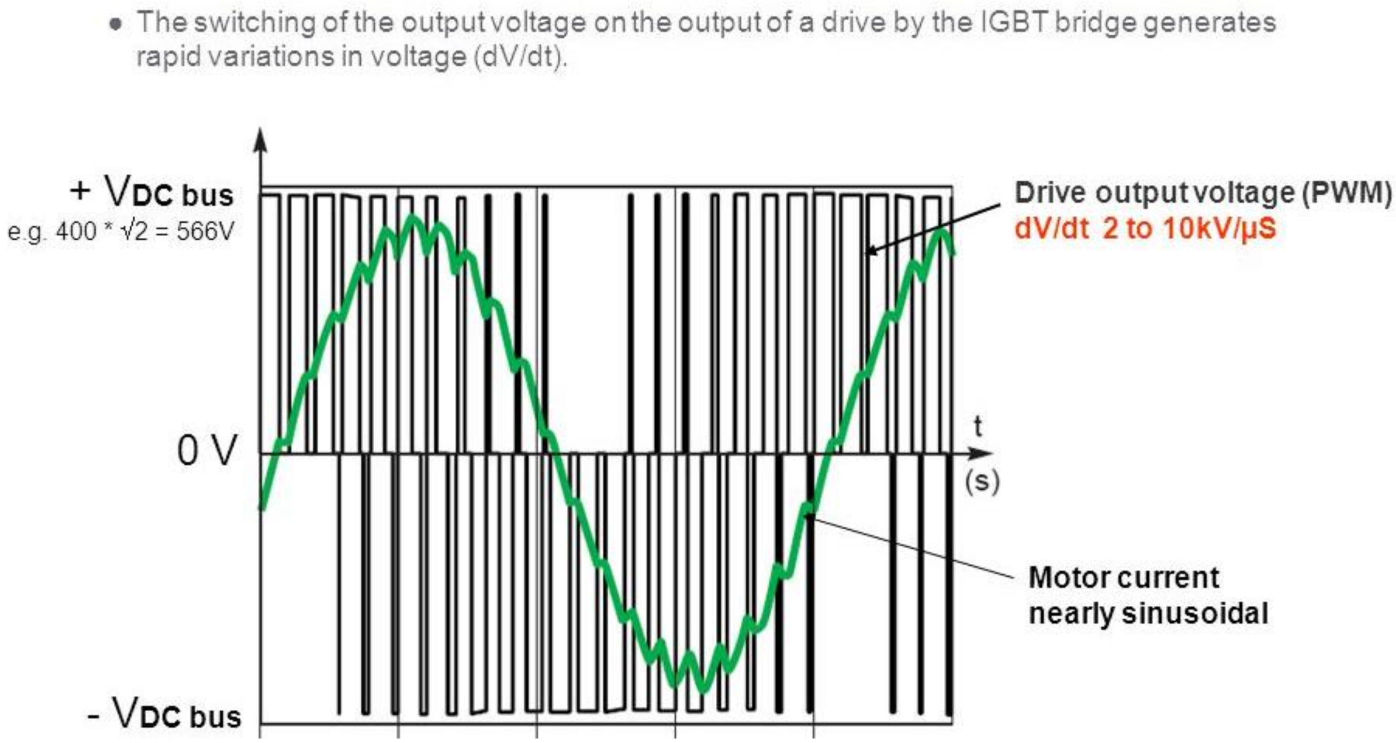
Voltage Control





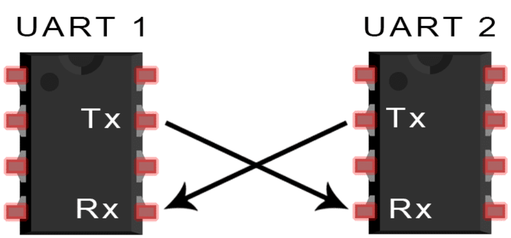
Power Electronics

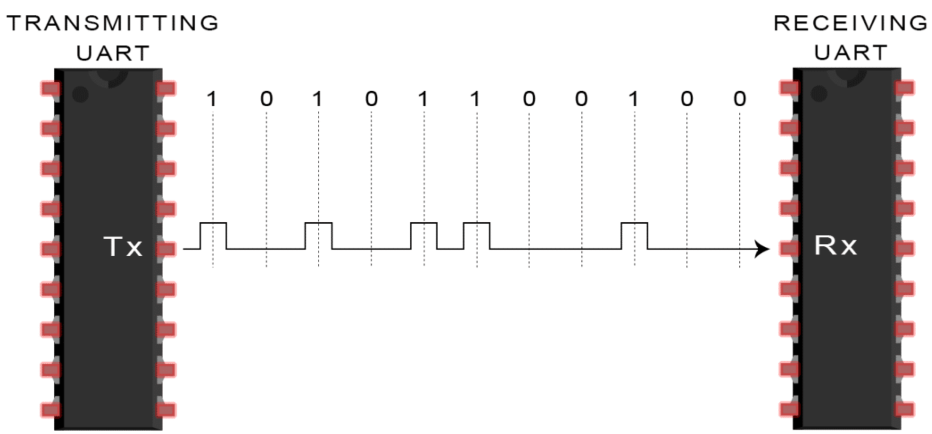
- Voltage Control



Serial

- The Serial interface provides UART functionality. The serial link has two unidirection channels, one for sending and one for receiving. The link is asynchronous, and so both ends of the serial link must be configured to use the same settings.
- One of the serial connections uses the Arm Mbed USB port, allowing you to easily communicate with your host PC.





Serial

- Introduction
- Reference
  - Overview
  - Runtime
  - Configuration
  - Technology
  - APIs
    - Platform
    - Drivers
      - AnalogIn
      - AnalogOut
      - DigitalIn
      - DigitalOut
      - DigitalInOut

Public Member Functions	
	Serial (PinName tx, PinName rx, const char *name=NULL, int baud=MBED_CONF_PLATFORM_DEFAULT_SERIAL_BAUD_RATE)
	Serial (PinName tx, PinName rx, int baud)
bool	**readable** ()
bool	**writable** ()
bool	**writeable** ()
Public Member Functions inherited from mbed::SerialBase	
void	baud (int baudrate)
void	format (int bits=8, Parity parity=SerialBase::None, int stop_bits=1)
int	readable ()
int	writeable ()

The **baudrate** is the transmission's speed and it's expressed in **bit per second**. Standards values are 9600, 115200, ..., 921600.

Serial

Public Member Functions inherited from mbed::Stream	
	Stream (const char *name=NULL)
int	**putc** (int c)
int	**puts** (const char *s)
int	**getc** ()
char *	**gets** (char *s, int size)
int	**printf** (const char *format,...)
int	**scanf** (const char *format,...)

```
#include "mbed.h"

Serial pc(USBTX, USBRX); // tx, rx

int main() {
    pc.printf("Hello World!\n\r");
    while(1) {
        pc.putc(pc.getc() + 1); // echo input back to terminal
    }
}
```

Serial - PC side

To read data from the serial with the PC you can use two different methods:

PIO Serial Monitor

The **default baudrate is 9600 bit/s**, so if you want to use an higher speed you must change it in the *platformio.ini* file adding the following line:

```
monitor_speed = 115200
```

(example with a baudrate of 115200)

serial_monitor.out
(suggested)

https://gitlab.com/altairLab/elasticteam/nucleo/ultimate-serial-monitor

Change the *config.csv* file to set the baudrate and to select the port.

Homework #2

- Connect a potentiometer
- Make a LED blinking with a frequency setted by the potentiometer [from 1Hz to 10Hz]
- In the meanwhile send the potentiometer value to the PC via serial every 0.2s

Timers

- Use the Timer interface to create, start, stop and read a timer for measuring small times (between microseconds and seconds).
- You can independently create, start and stop any number of Timer objects.

Warning: Timers are based on 32-bit int microsecond counters, so they can only time up to a maximum of 2^31-1 microseconds (30 minutes). They are designed for times between microseconds and seconds. For longer times, you should consider the time() real time clock.

Timers

```
#include "mbed.h"

Timer t;

int main() {
    t.start();
    printf("Hello World!\n");
    t.stop();
    printf("The time taken was %f seconds\n", t.read());
}
```

mbed::Timer Class Reference

Public Member Functions	
	Timer (const ticker_data_t *data)
void	start ()
void	stop ()
void	reset ()
float	read ()
int	read_ms ()
int	read_us ()
	operator float ()
us_timestamp_t	read_high_resolution_us ()

Other Timing methods

https://os.mbed.com/questions/61002/Equivalent-to-Arduino-millis/

Tickers

- Use the Ticker interface to set up a recurring interrupt; it calls a function repeatedly and at a specified rate.
- You can create any number of Ticker objects, allowing multiple outstanding interrupts at the same time. The function can be a static function, a member function of a particular object or a Callback object.

Warnings and notes

- No blocking code in ISR: avoid any call to wait, infinite while loop or blocking calls in general.
- No printf, malloc or new in ISR: avoid any call to bulky library functions. In particular, certain library functions (such as printf, malloc and new) are not re-entrant, and their behavior could be corrupted when called from an ISR.

Tickers

```
#include "mbed.h"

Ticker flipper;
DigitalOut led1(LED1);
DigitalOut led2(LED2);

void flip() {
    led2 = !led2;
}

int main() {
    led2 = 1;
    flipper.attach(&flip, 2.0); // the address of the function to be attached (flip) and the interval (2
seconds)

    // spin in a main loop. flipper will interrupt it to call flip
    while(1) {
        led1 = !led1;
        wait(0.2);
    }
}
```

Hardware Timers

- Tickers cannot precisely manage high frequency real time tasks!

- We need to use TIMERS and within underlying HAL interface.
- HAL User Manual

http://www.st.com/content/ccc/resource/technical/document/user_manual/2f/71/ba/b8/75/54/47/cf/DM00105879.pdf/files/DM00105879.pdf/jcr:content/translations/en.DM00105879.pdf

Hardware Timers

User Manual pag 984

- Prescaler: specifica il valore prescaler usato per dividere il TIM clock
- CounterMode: specifica la modalità di count (up, down)
- Period: specifica il valore del periodo da essere caricato dentro al registro Auto-Reload al prossimo evento di update
- ClockDivision: specifica gli eventi (rising, falling, both)

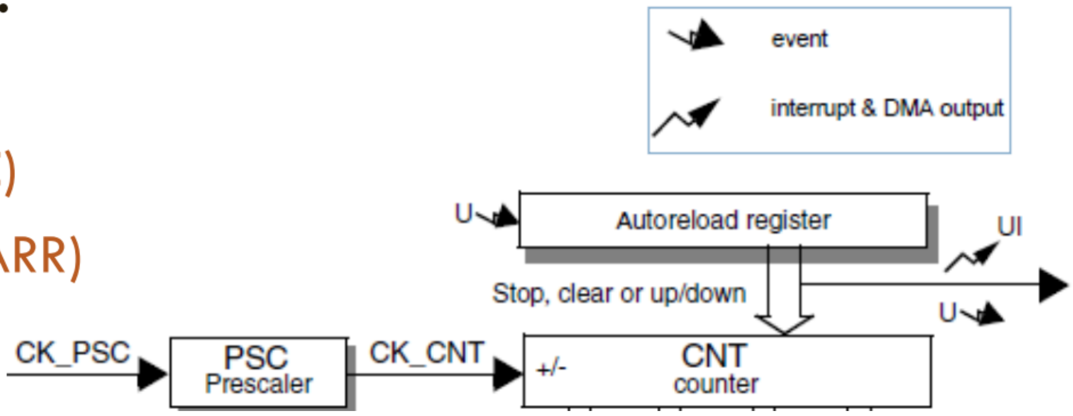
Hardware Timers

Field Documentation

- ***uint32_t TIM_Base_InitTypeDef::Prescaler***
  Specifies the prescaler value used to divide the TIM clock. This parameter can be a number between Min_Data = 0x0000U and Max_Data = 0xFFFFU
- ***uint32_t TIM_Base_InitTypeDef::CounterMode***
  Specifies the counter mode. This parameter can be a value of ***TIM_Counter_Mode***
- ***uint32_t TIM_Base_InitTypeDef::Period***
  Specifies the period value to be loaded into the active Auto-Reload Register at the next update event. This parameter can be a number between Min_Data = 0x0000U and Max_Data = 0xFFFF.
- ***uint32_t TIM_Base_InitTypeDef::ClockDivision***
  Specifies the clock division. This parameter can be a value of ***TIM_ClockDivision***
- ***uint32_t TIM_Base_InitTypeDef::RepetitionCounter***
  Specifies the repetition counter value. Each time the RCR downcounter reaches zero, an update event is generated and counting restarts from the RCR value (N). This means in PWM mode that (N+1) corresponds to:the number of PWM periods in edge-aligned modethe number of half PWM period in center-aligned mode This parameter must be a number between Min_Data = 0x00 and Max_Data = 0xFF.
  Note: This parameter is valid only for TIM1 and TIM8.

Hardware Timers

- The main block of the programmable timer is a 16-bit counter with its related auto-reload register
  - The counter can count up, down or both up and down
  - The counter clock can be divided by a prescaler.
- The counter, the auto-reload register and the prescaler register can be written or read by software
  - This is true even when the counter is <u>running</u>
- The time-base unit includes:
  - Counter register (TIMx_CNT)
  - Prescaler register (TIMx_PSC)
  - Auto-reload register (TIMx_ARR)



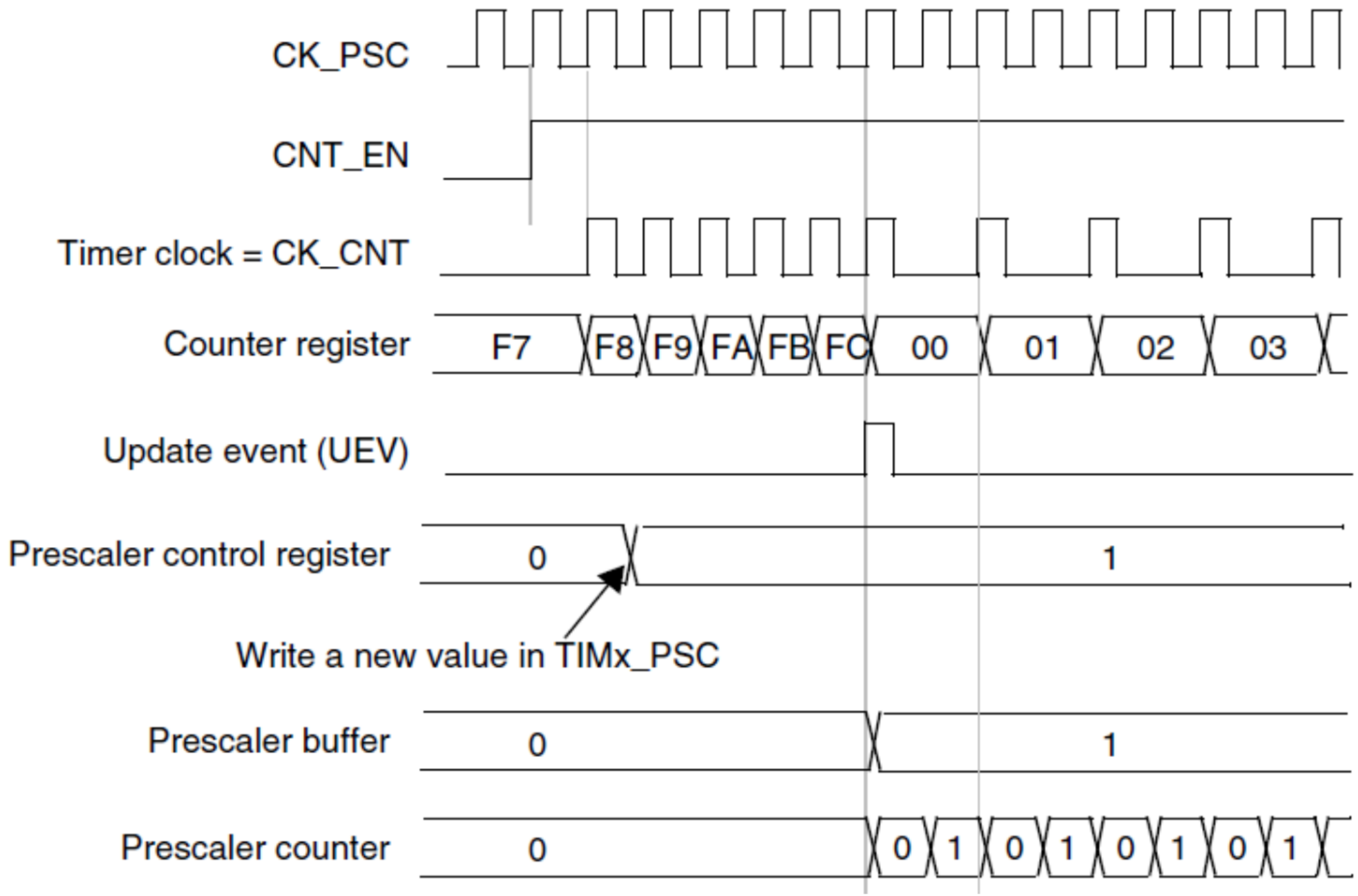
Hardware Timers

- Auto-reload register is preloaded
- Writing to or reading from the auto-reload register accesses the preload register
- Contents of preload register are transferred into the shadow register permanently or at each update event (UEV), depending on the auto-reload preload enable bit (ARPE) in TIMx_CR1 register
- Update event is sent when counter reaches overflow or underflow and if the UDIS bit equals 0 in TIMx_CR1 register
- Update event can also be generated by software
- Counter is clocked by prescaler output CK_CNT, which is enabled only when counter enable bit (CEN) in TIMx_CR1 register is set
  - actual counter enable signal CNT_EN is set 1 clock cycle after CEN

Hardware Timers

- Prescaler can divide the counter clock frequency by any factor between 1 and 65536
- Based on a 16-bit counter controlled through a 16-bit register (in the TIMx_PSC register)
- It can be changed on the fly as this control register is buffered
  - New prescaler ratio is taken into account at the next update event

Hardware Timers



Hardware Timers

- Counter counts from 0 to the auto-reload value (content of the TIMx_ARR register), then restarts from 0 and generates a counter overflow event
- An Update event can be generated at each counter overflow or by setting the UG bit in the TIMx_EGR register
- When an update event occurs, all the registers are updated and the update flag (UIF bit in TIMx_SR register) is set (depending on the URS bit):
  - The buffer of the prescaler is reloaded with the preload value (content of the TIMx_PSC register)
  - The auto-reload shadow register is updated with the preload value (TIMx_ARR)

Hardware Timers

How to use TIMs

$$\mathbf{FREQ} = \frac{\textbf{CLK\_FREQ}}{\textbf{(PRESCALER+1)} * \textbf{(PERIOD+1)}}$$

Frequency of what?

ClockDivision = TIM_CLOCKDIVISION_DIV1;

PRESCALER: integer between 1 and 65536

PERIOD: integer between 1 and 65536

Hardware Timers - examples

F446RE

Max clock freq = 90MHz

TIM7: Resolution = 16bit

TIM3: Resolution = 16bit

F401RE

Max clock freq = 84MHz

TIM3: Resolution = 16bit

Example

1KHz = 84MHz / ((999+1) * (83+1))

Hardware Timers

```
/* Here we define the HW TIM we're going to use */
/* In this case we choose TIM15 from the datasheet */
#define TIM_USR       TIM15
#define TIM_USR_IRQ   TIM15_IRQn
```

```
/* We use this variable to detect whether the interrupt has incurred */
volatile char flag_time = 0;

// Timer handler structure
TIM_HandleTypeDef mTimUserHandle;

/* This function handle timer 15 interrupt, it simply set flag_time to 1 to save execution time */
extern "C"
void M_TIM_USR_Handler(void) {
  if (__HAL_TIM_GET_FLAG(&mTimUserHandle, TIM_FLAG_UPDATE) == SET) {
    /* We clear the flag so we're able to receive another interrupt */
    __HAL_TIM_CLEAR_FLAG(&mTimUserHandle, TIM_FLAG_UPDATE);
    /* We set our flag_time variable so we can temporize the while(true) cycle in the main() */
      flag_time = 1;
  }
}
```

Hardware Timers

```
// Enable the clock related to TIM15
__HAL_RCC_TIM15_CLK_ENABLE();

/* Here we configure the mTimUserHandle structure to be able to use TIM15 */
/* Based on the clock frequency, we set the parameters in order to get a ~1KHz loop frequency */
mTimUserHandle.Instance               = TIM_USR;
mTimUserHandle.Init.Prescaler         = 4799;
mTimUserHandle.Init.CounterMode       = TIM_COUNTERMODE_UP;
mTimUserHandle.Init.Period            = 2;
mTimUserHandle.Init.ClockDivision     = TIM_CLOCKDIVISION_DIV1;

/* We init the previous defined structure*/
HAL_TIM_Base_Init(&mTimUserHandle);
/* And we start the Timer */
HAL_TIM_Base_Start_IT(&mTimUserHandle);

/* We need also to enable ther interrupt service related to TIM15 */
NVIC_SetVector(TIM_USR_IRQ, (uint32_t)M_TIM_USR_Handler);
NVIC_EnableIRQ(TIM_USR_IRQ);
```

Homework #3

- Connect a potentiometer
- Make a LED blinking by using a hardware timer
- In the meanwhile, send the potentiometer value to the PC via serial at the maximum frequency.
- **How much is such maximum frequency? Aru you sure that such maximum frequency is guaranteed?**