POLITECNICO
MILANO 1863

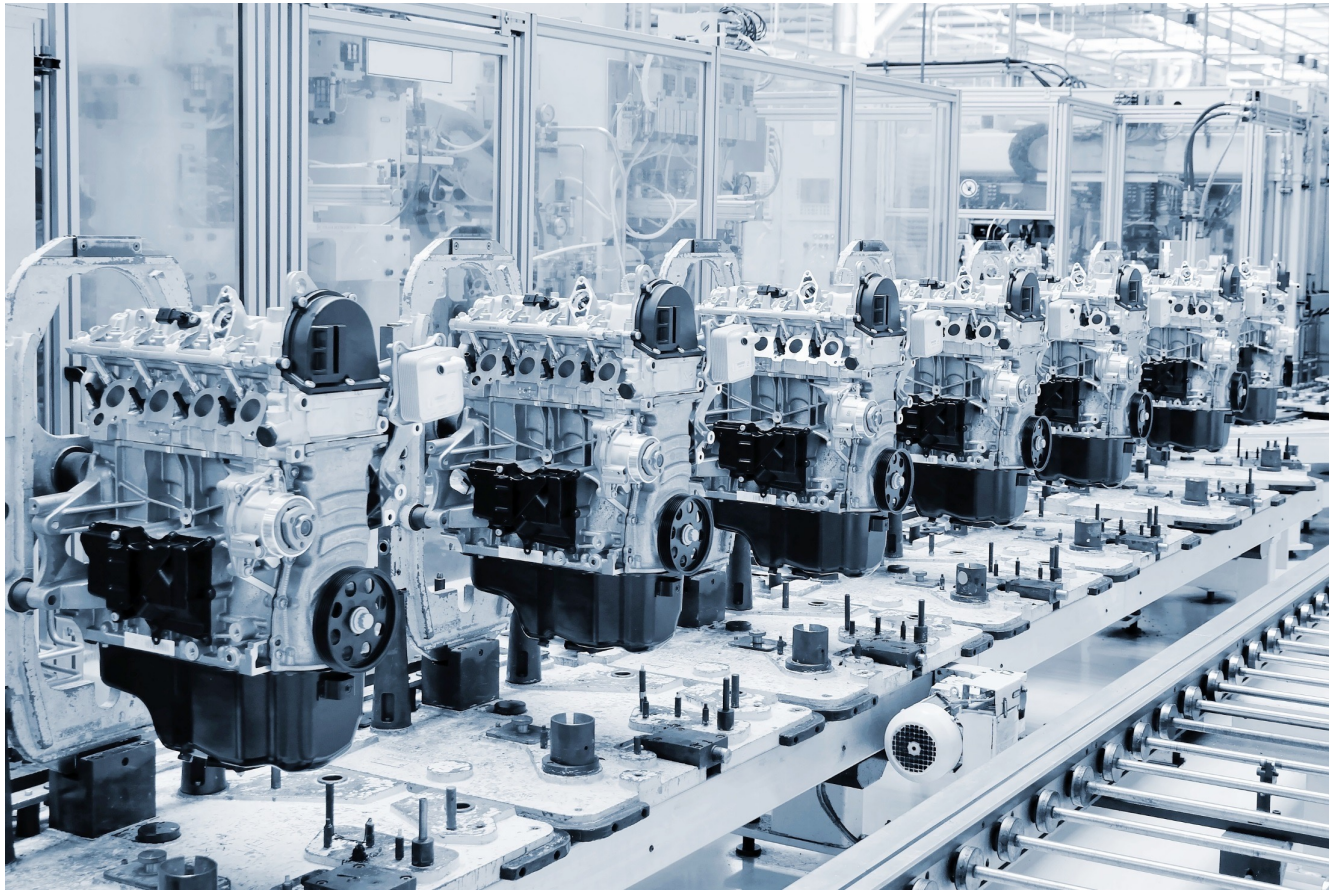
From Arduino to STM32

Ver 2.0 - 2020
Ver 1.0 - 2012 Oct

Fabio Salice

Arduino to ST NUCLEO

MIGRATION NOTES – Max Cavazzana (August 2017)



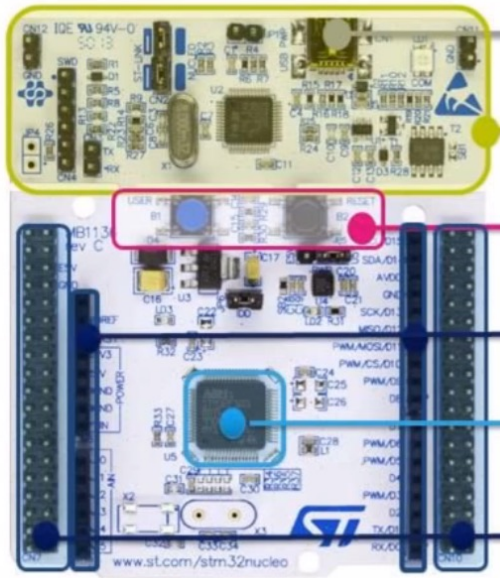
ARDUINO UNO vs. ST NUCLEO 64 – main differences

	ARDUINO UNO	ST NUCLEO 64
Microcontroller core	ATMEL AtMega	STM32 ARM Cortex
	8 bit	32 bit
Core voltage	5V	3.3V
Core speed	16MHz	32 – 180Mhz
Flash size	32KB	64K-1MB
RAM size	2KB	8-128 KB
Digital I/O	14	50
Analog inputs	6	16

- **Some version of the STM32 core also integrates the FPU (Floating Point Unit) to perform faster floating point calculations**
- **Some versions of the STM32 core has integrated DAC for real analog output**

ST NUCLEO form factor

STM32 Nucleo features



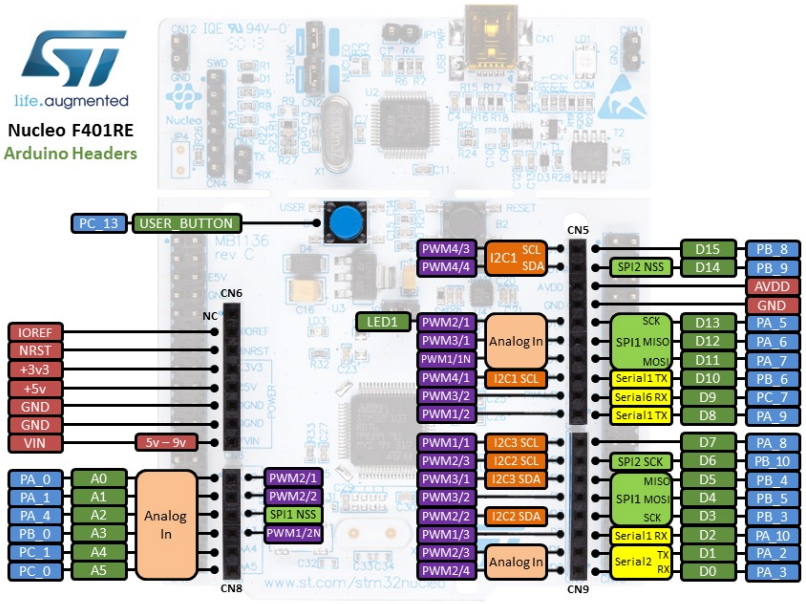
- The board is supplied through USB or external source
- Integrated ST-LINK/V2-1 for debug and programming
- 1 User and 1 Reset push buttons and 1 User LED
- Arduino™ extension connectors allowing add-ons compatibility
- One STM32 MCU flavor with 64 pins
- ST Morpho extension headers to direct access to all MCU I/Os

ALTRI VIDEO

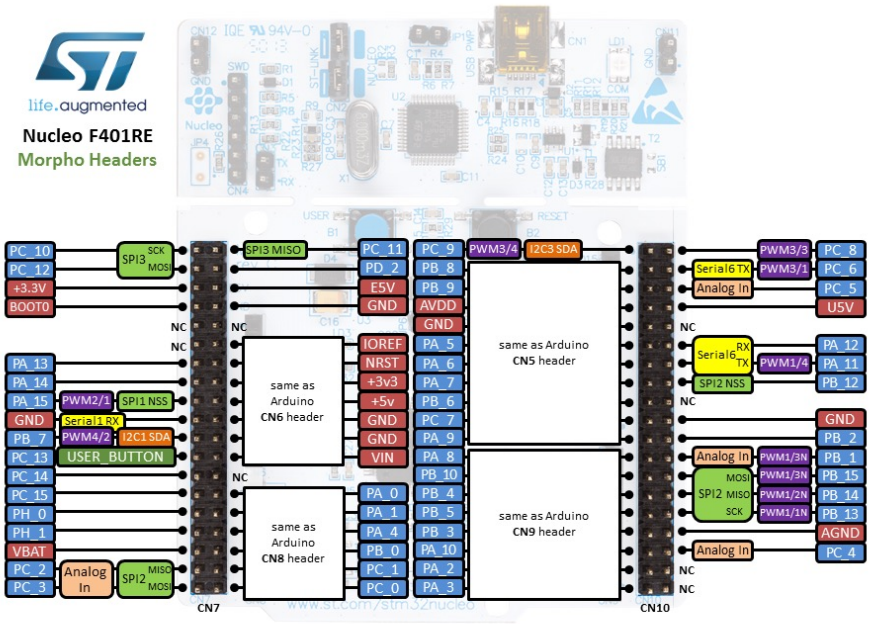
STM32 Nucleo offers an integrated ST-LINK/V2-1 for direct programming under mbed environment

ST NUCLEO connectors

Arduino-compatible headers



ST-MORPHO headers



- **Easy programming through USB:**
  - ST Nucleo board connected via USB to the PC: seen as a USB drive
  - File *.bin generated by the compiler can be dragged to the USB drive: **MCU IS NOW PROGRAMMED!**
  - Using online ARM mBed as development tool, DEBUG is done via USB serial UART using any hyper terminal software (TeraTerm, Zterm…)

C Data types - INTEGER

- Better using C99 standard int types
- Advantages:
  - code portability: c type names may change in different compiler versions
  - data types contains memory size: you always know what you're declaring

INTEGER TYPES

C type	stdint.h type	Bits	Sign	Range
char	uint8_t	8	Unsigned	0 .. 255
signed char	int8_t	8	Signed	-128 .. 127
unsigned short	uint16_t	16	Unsigned	0 .. 65,535
short	int16_t	16	Signed	-32,768 .. 32,767
unsigned int	uint32_t	32	Unsigned	0 .. 4,294,967,295
int	int32_t	32	Signed	-2,147,483,648 .. 2,147,483,647
unsigned long long	uint64_t	64	Unsigned	0 .. 18,446,744,073,709,551,615
long long	int64_t	64	Signed	-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807

- **uint8_t** and **int8_t** are only intended to be used instead of char as integer 8 bit variables. For text character and strings always declare **char** types.
- Because the natural data-size for an ARM processor is 32-bits, it is much more preferable to use **(u)int32_t** as a variable instead **(u)int16_t**; the processor may actually have to use more instructions to do a calculation on a 16 bit than a 32 bit!

C Data types – FLOATING POINT and POINTERS

- Arduino only manage 32 bit floating point (float)
- In Arduino, 64 bit floating point (double) may be declared but are stored and computed as 32 bit.
- ST Nucleo and ARM are really managing 64 bit floating points

FLOATING POINT TYPES

C type	IEE754 Name	Bits	Range
float	Single Precision	32	-3.4E38 .. 3.4E38
double	Double Precision	64	-1.7E308 .. 1.7E308

POINTERS

- The ARMv7-M architecture used in mbed microcontrollers is a 32-bit architecture, so standard C pointers are 32-bits.
- ie: a pointer to a 8 bit variable declared as follows takes 32 bits:

```
foo *int8_t;
```

C reference – Digital I/O ➔ standard functions to C++ classes

ARDUINO IDE (uses standard functions)

PinMode (pin, mode)
– defines the digital pin direction and mode

```
..
pinMode (13, OUTPUT); //Set the digital pin 13 as output
pinMode (7, INPUT); //Set the digital pin 7 as input
..
```

void ***digitalWrite(pin, value)***
– write a digital ***value*** on the ***pin*** (HIGH or LOW)

```
..
digitalWrite (13, HIGH); //Set the high output on digital pin 13
..
```

int ***digitalRead(pin)***
– read the digital value from the pin

```
..
Int value;
value = digitalRead (7); //read the digital input 7 into variable "value"
..
```

STM32 NUCLEO on MBED (uses C++ classes)

class ***DigitalIn(PinName pin)***
– defines a digital input class and mode (optional)

```
..
DigitalIn button_1(D0) ; //create a class button_1 connected to pin "D0"
..
```

int ***DigitalIn.read***()
– read the digital value from the pin

```
..
int value;
value = button_1.read() ; //read the digital input button_1 into variable "value"
..
```

class ***DigitalOut(PinName pin)***
– defines a digital output class

```
..
DigitalOut led(LED1) ; //create a class led connected to pin "LED1" (D13)
..
```

void ***DigitalOut.write****(int value)*
– write the int value on the digital output pin

```
..
led.write(1) ; //write a digital 1 (high) to the "led" pin
led = 1; //a shorthand to write 1 (high) to the "led" pin
led = !led; //a shorthand to toggle the digital output
..
```

C reference – Analog I/O ➔ classes again!

ARDUINO IDE (uses standard functions)

int ***analogRead(pin)***
– read an analog ***value*** from the ***pin***

```
..
int value
value = analogRead(A0); //read the analog value of pin "A0" into variable "value"
..
```

void ***AnalogWrite(pin, int value) ** only generates PWM***
– write the analog value as PWM on a digital output pin

```
..
pinMode (DO, OUTPUT)
analogWrite (D0, 512); // generates a 50% duty cycle PWM on the digital output pin
..
```

ARDUINO does not integrate the DAC so ***AnalogOutput*** only generates a PWM.
Value is int type in the range of 0-1023:
0 = 0 duty cycle (pin always off)
512 = 50% duty cycle
1023 = 100% duty cycle (pin always on)

STM32 NUCLEO on MBED (uses C++ classes)

class ***AnalogIn(PinName pin)***
– defines an analog input class

```
..
AnalogIn pot(A0) ; //create a class "pot" connected to analog input pin "A0"
..
```

float ***AnalogIn.read***()
– read the analog value from the pin

```
..
float value;
value = pot.read() ; //read the analog input "pot" into variable "value"
```

class ***AnalogOut(PinName pin) ** only works for DAC outputs***
– defines an analog output class

```
..
AnalogOut dac(PA_4) ; //create a class "dac" connected to analog output pin "PA_4"
..
```

void ***AnalogOut.write****(float value)*
– write the float value on the analog output pin

```
..
dac.write(0.5) ; //write a digital 1 (high) to the "led" pin
dac = 0.5; //a shorthand to write 0.5 to the "dac" pin
..
```

AnalogIn and ***AnalogOut*** classes are using float values as a percentage: 0.1 = 10%; 0.5 = 50%; 0.625 = 62.5%.....

Usage of classes in STM32 Nucleo with MBED is much linear and easy than using standard functions with ARDUINO

- write and read can be done using shorthand operators for both digital and analog classes
- Analog write and read uses float so the precision of the input/output is limited only by the microcontroller peripherals and not by the code (float values can carry up to 7 decimals)
- Reading or writing analog values as percentage makes the code simpler and calculations easier
- Creating classes is helping the clarity of the code as the input or output classes can be defined only once at the beginning of the code: uses of #define to refers to pin is not necessary
- No pin direction definition is needed: you don't need to remember how and if the pin direction has been defined
- The sintax and construction of read and write operations is the same for both digital and analog classes
- Both Analog and Digital classes are integrating some advanced member functions for additional features like reading the status of an output pin or checking if the pin is actually connected or not.

mbed interrupt handling – classes… again!

- mbed uses C++ classes to define interrupts
- almost every pin of the microcontroller can be configured as an external interrupt input (ARDUINO UNO only has 2….)
- the interrupt can be defined for RISE and FALL events separately and different functions can be called for the same interrupt depending if a rise or fall event occurs

```
1  // Flash an LED while waiting for events
2
3  #include "mbed.h"
4
5  InterruptIn event(p16);          //define interrupt "event" connected to pin "p16"
6  DigitalOut led(LED1);            //define digital output "led" connected to pin "LED1"
7
8  void trigger_rise() {                    //void function to be executed when the interrupt is triggered
9      printf("rising edge triggered!\n");  //print a text on the standard output to show that
10                                          //the interrupt routine has been executed
11 }
12
13 void trigger_fall() {
14     printf("falling edge triggered!\n");
15 }
16
17 int main() {
18     event.rise(&trigger_rise);   //define the "event" interrupt for RISE event and attach the
19                                  //void function "trigger_rise"
20     event.fall(&trigger_fall);   //define the "event" interrupt for FALL event and attach the
21                                  //void function "trigger_fall"
22
23     while(1) {
24         led = !led;              //toggle the LED output
25         wait(0.25);              //wait 250 milliseconds
26     }
27 }
```

interrupt defined as a class

the functions we want to assign to an interrupt MUST be «void» with no parameters passed

The member functions «rise» and «fall» are attaching the functions to the interrupt respectively on rising and falling events
The argument passed to these functions are the pointers to the interrupt routine functions

- "rise" and "fall" member functions are acting separately. If we want to define the same interrupt routine for both the events we need to attach the same function:

```
29     event.rise(&trigger_rise);
30     event.fall(&trigger_rise);
31
```

mbed PWM generator

– class PwmOut allow the generation of PWM on almost all the microcontroller pins

```
// Fade a led on
#include "mbed.h"

PwmOut led(LED1);           //define a PWM "led" connected to pin "LED1"

int main() {
    while(1) {
        led = led + 0.01;   //increase PWM duty cycle by 1% (float used as percentage)
        wait(0.2);          //wait 200 milliseconds
        if(led == 1.0) {    //if PWM duty cycle of "led" reach 100%..
            led = 0;        //set PWM duty cycle to 0
        }
    }
}
```

PWM defined as a class

The write operation can be performed using simple float operations. Duty Cycle is expressed as float percentage

The PWM duty cycle value can be read using a float operator

– PWM period length can be defined within the class….

```
// high speed PWM example
#include "mbed.h"

PwmOut pulse(D0);           //define a PWM "pulse" connected to pin "D0"

int main()
    pulse.period_us(10);    //set "pulse" period lenght to 10 microseconds (100Khz)
    pulse.write(0.45);      //set duty cycle of "pulse" to 45% - this turn PWM on
    ...                     //this is equal to "pulse = 0.45"
    ...
    pulse.write(0.0);       //set duty cycle of pulse to 0 - this turn PWM off
}
```

PWM period lenght can be specified using those member functions:
«period(float)» - specify the period in seconds using decimals
«period_ms(int)» – specify the period in milliseconds using integer
«period_us(int)» – specify the period in microseconds using integer
Minimum period lenght is 1 microsecond means 1Mhz PWM

mbed PWM generator

– class PwmOut allow the generation of PWM on almost all the microcontroller pins

```
1  #include "mbed.h"
2
3  PwmOut led(LED1);
4
5  int main() {
6      // specify period first
7      led.period(4.0f);      // 4 second period
8      led.write(0.50f);      // 50% duty cycle, relative to period
9      //led = 0.5f;          // shorthand for led.write()
10     //led.pulsewidth(2);   // alternative to led.write, set duty cycle time in seconds
11     while(1);
12 }
```

PWM defined as a class

Define the period of the PWM

Define the Duty-Cycle (value from 0 to 1)

mbed Ticker

- class Ticker allow the repeatedly call of functions at a recurring interval

```
// Toggle the blinking led after 5 seconds

#include "mbed.h"

Ticker swap;            //define a ticker with name "swap"
DigitalOut led1(LED1);  //define digital output for LED1
DigitalOut led2(LED2);  //define digital output for LED1

uint8_t flip = 0;       //define a 8 bit int variable used as a boolean

void attime() {         //this function will be attached to the ticker and called
    flip = !flip;       //invert the status of the "flip" variable (as a boolean)
}

int main() {
    swap.attach(&attime, 5); //attach the "attime" function and call it every 5 seconds
    while(1) {               //infinite looping
        if(flip == 0)        //check the bollean value of "flip"
            led1 = !led1;    //blink LED1 if flip is 0
        } else {
            led2 = !led2;    //blink LED2 if flip is 1
        }
        wait(0.2);           //LED blink delay is 200ms
    }
    swap.detach();
}
```

Ticker defined as a class

Define a function to be called by the ticker

Attach the function to the ticker and specify the time interval for calling the function. In this case the function «attime» will be called every 5 seconds

The function «attime» invert the boolean value of the variable «flip» so this if conditions return a different value every 5 seconds, changing the LED which is currently bliking

The ticker function can be detached if needed. This line is used only as an example: in this case this instruction will never be reached as the while loop above is infinite

- Interval can be defined also in microseconds by using the function "ticker.attach_us"

```
ticker.attach(Callback<void()>func, float t);
//time in seconds float

ticker.attach_us((Callback<void()>func, us_timestamp_t t);
//time in microseconds int

ticker.detach();
```

- Ther is no limit at the number of Ticker that you can define and use simultaneously
- You can detach the function when needed and the Ticker will stop.
- Ticker is a simple way of using interrupts generated by the timer

Thank you!