ST NUCLEO BOARDS

Andrea Calanca
ST NUCLEO Boards

• STM32F411RET6 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 with 16 channels
• RTC, Timers (8), I2C (3), USART (3), SPI (5), USB OTG Full Speed, SDIO

• https://os.mbed.com/platforms/ST-Nucleo-F302R8/
• https://os.mbed.com/platforms/ST-Nucleo-F411RE/
ST NUCLEO Boards

- Power supply through USB or external source
- Integrated debugging and programming ST-LINK probe
- STM32 microcontroller
- Complete product range from ultra-low power to high-performance
- ST Morpho and Arduino™ UNO R3 extension headers
Processor Datasheet

**STM32F302x6 STM32F302x8**

ARM® Cortex®-M4 32-bit MCU+FPU, up to 64 KB Flash, 16 KB SRAM, ADC, DAC, USB, CAN, COMP, Op-Amp, 2.0 - 3.6 V

**Features**

- **Core**: ARM® 32-bit Cortex®-M4 CPU with FPU (72 MHz max.), single-cycle multiplication and HW division, DSP instruction
- **Memories**
  - 32 to 64 Kbytes of Flash memory
  - 16 Kbytes of SRAM on data bus
- **CRC calculation unit**
- **Reset and power management**
  - \( V_{DD}, V_{DDA} \) voltage range: 2.0 to 3.6 V
  - Power-on/Power down reset (POR/PDR)
  - Programmable voltage detector (PVD)
  - Low-power: Sleep, Stop, and Standby
  - \( V_{BAT} \) supply for RTC and backup registers
- **Clock management**
  - 4 to 32 MHz crystal oscillator
  - 32 kHz oscillator for RTC with calibration
  - Internal 8 MHz RC with x 16 PLL option
  - Internal 40 kHz oscillator
- **Up to 51 fast I/O ports, all mappable on**

- **Up to 18 capacitive sensing channels**
  - Supporting touchkey, linear and rotary sensors
- **Up to 9 timers**
  - One 32-bit timer with up to 4 IC/OC/PWM or pulse counter and quadrature (incremental) encoder input
  - One 16-bit 6-channel advanced-control timer, with up to 6 PWM channels, deadtime generation and emergency stop
  - Three 16-bit timers with IC/OC/OCN or PWM, deadtime gen. and emergency stop
  - One 16-bit basic timer to drive the DAC
  - Two watchdog timers (independent, window)
  - SysTick timer: 24-bit downcounter
- **Calendar RTC with alarm, periodic wakeup from Stop/Standby**
### Table 2. STM32F302x6/8 device features and peripheral counts

<table>
<thead>
<tr>
<th>Peripheral</th>
<th>STM32F302Kx</th>
<th>STM32F302Cx</th>
<th>STM32F302Rx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash (Kbytes)</td>
<td>32</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>SRAM (Kbytes)</td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td><strong>Timers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Advanced control</td>
<td>1 (16-bit)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General purpose</td>
<td>3 (16-bit)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SysTick timer</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Watchdog timers (independent, window)</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PWM channels (all)(^{(1)})</td>
<td>16</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>PWM channels (except complementary)</td>
<td>10</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td><strong>Comm. interfaces</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPI/I2S</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I²C</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USART</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>USB 2.0 FS</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAN 2.0B</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>GPIOs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Normal I/Os (TC, TTa)</td>
<td>9</td>
<td>20</td>
<td>26</td>
</tr>
<tr>
<td>5-Volt tolerant I/Os (FT, FT1)</td>
<td>15</td>
<td>17</td>
<td>25</td>
</tr>
<tr>
<td>DMA channels</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capacitive sensing channels</td>
<td>13</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>12-bit ADC</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Number of channels</td>
<td>8</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>12-bit DAC channels</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Analog comparator</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Operational amplifier</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>CPU frequency</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating voltage</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hardware Abstraction Layer

Overall system architecture

- STM32 Nucleo boards
- X-NUCLEO-PLC01A1 expansion board

HAL level examples
- Hardware Abstraction Layer API
- Boards support packages

Utilities
- CMSIS

Hardware components
- STM32Fx
- CLT01-38SQ7
- VNI8200XP
Mbed OS

Mbed OS

https://os.mbed.com/docs/v5.8/reference/index.html
Environment

```cpp
#include <embed.h>

DigitalOut myled(LED1);

int main()
{
    // put your setup code here, to run once:
    while(1) {
        // uint16_t ui_data = pot.read_u16();
        // float f_data = pot.read();
        // myled = f_data;
        myled = 1; // LED is ON
        wait(1.0); // 200 ms
        myled = 0; // LED is OFF
        wait(2.0); // 1 sec
        // pc.printf("%f \n", f_data);
    }
}
```
Environment
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.

https://os.mbed.com/docs/v5.8/reference/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%4X\n", i, samples[i]);
    }
}
```
Analog Out

- 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 pinmap for your board.

https://os.mbed.com/docs/v5.8/reference/analogout.html
Digital In & out

- **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.

- **DigitalOut**
  - Use the DigitalOut interface to configure and control a digital output pin by setting the pin to logic level 0 or 1.
Digital In & out

- **DigitallInOut**
- Use the DigitallInOut 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 DigitallInOut.

https://os.mbed.com/docs/v5.8/reference/digitalinout.html
PWM Out

- **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.

https://os.mbed.com/docs/v5.8/reference/pwmout.html
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.

https://os.mbed.com/docs/v5.8/reference/serial.html
## Serial

### 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 ()`

- `void attach (Callback< void() > func, IrqType type=RxIrq)`

- `template<typename T > void attach (T *obj, void(T::*method)(), IrqType type=RxIrq)`

- `template<typename T >`
Public Member Functions inherited from `mbed::Stream`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Stream</code></td>
<td>(const char *name=NULL)</td>
</tr>
<tr>
<td><code>putc</code></td>
<td>(int c)</td>
</tr>
<tr>
<td><code>puts</code></td>
<td>(const char *s)</td>
</tr>
<tr>
<td><code>getc</code></td>
<td>()</td>
</tr>
<tr>
<td><code>gets</code></td>
<td>(char *, int size)</td>
</tr>
<tr>
<td><code>printf</code></td>
<td>(const char *format,...)</td>
</tr>
<tr>
<td><code>scanf</code></td>
<td>(const char *format,...)</td>
</tr>
</tbody>
</table>

```c
#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

#include "mbed.h"

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

Serial pc(USBTX, USBRX);

void callback_ex() {
    // Note: you need to actually read from the serial to clear the RX interrupt
    pc.putc(pc.getc());
    led2 = !led2;
}

int main() {
    pc.attach(&callback_ex);

    while (1) {
        led1 = !led1;
        wait(0.5);
    }
}

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

```c
#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

<table>
<thead>
<tr>
<th>Public Member Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Timer</strong> *(const ticker_data_t <em>data)</em></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td><strong>start</strong> ()</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td><strong>stop</strong> ()</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td><strong>reset</strong> ()</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td><strong>read</strong> ()</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td><strong>read_ms</strong> ()</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td><strong>read_us</strong> ()</td>
</tr>
<tr>
<td>us_timestamp_t</td>
</tr>
<tr>
<td><strong>read_high_resolution_us</strong> ()</td>
</tr>
</tbody>
</table>
Other Timing methods

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

https://docs.mbed.com/docs/mbed-os-api/en/latest/api/group__hal__UsTicker.html
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.
#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);
    }
}

https://os.mbed.com/docs/v5.8/reference/ticker.html
Hardware Timers

• Tickers cannot precisely manage high frequency real time tasks (is it true?)

• We need to use the underlying HAL interface.

• HAL User Manual

Hardware Timers

User Manual pag 868

- 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)

User Manual pag 873

Funzionamento TIM API
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 running
- 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

CK_PSC

CNT_EN

Timer clock = CK_CNT

Counter register

F7 F8 F9 FA FB FC 00 01 02 03

Update event (UEV)

Prescaler control register

Write a new value in TIMx_PSC

Prescaler buffer

0 1

Prescaler counter

0 01 01 01 01 01
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

\[ FREQ = \frac{CLK\_FREQ}{(PRESCALER+1) \times (PERIOD+1)} \]

ClockDivision = TIM_CLOCKDIVISION_DIV1;
PRESCALER: integer between 1 and 65536
PERIOD: integer between 1 and 65536
Hardware Timers

F446RE
Max clock freq = 90MHz
TIM7: Resolution = 16bit
TIM3: Resolution = 16bit

F401RE
Max clock freq = 84MHz
TIM3: Resolution = 16bit
Examples:
1KHz = 1'000Hz = 84MHz / 4799*5 / (86+1)
10KHz = 84MHz / 100 / (83+1)
Hardware Timers

F411RE: Max timer clock = 100MHz
TIM4: Resolution = 16bit
Examples:
1KHz = 1'000Hz = 100MHz / 4799*5 / (3+1)
10Hz = 100’000'000Hz / 4799*5 / (414+1)

F302R8: Max timer clock = 72MHz
TIM15: Resolution = 16bit
TIM16: Resolution = 16bit
Examples:
200Hz = 72MHz / 4799 / (74+1)
1KHz = 1'000Hz = 72MHz / 4799*5 / (2+1)
10Hz = 72'000'000Hz / 4799*5 / (299+1)
Hardware Timers

```cpp
#include "mbed.h"

// Internal Timer
#define TIM_USR TIM4
#define TIM_USR_IRQ TIM4_IRQHandler

// Serial communication
Serial pc(USBTX, USBRX, 57600); // tx, rx, speed
Timer tim;

// Flag for interrupt timer
volatile char flag_time = 0;

// Timer handler
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) {
        __HAL_TIM_CLEAR_FLAG(&mTimUserHandle, TIM_FLAG_UPDATE);
        flag_time = 1;
    }
}
```
Hardware Timers

```c
// Enable timer 4
__HAL_RCC_TIM4_CLK_ENABLE();
// 100 000 000 clock frequency
// Set timer 4 values to work at 1KHz
mTimUserHandle.Instance = TIM_USR;
mTimUserHandle.Init.Prescaler = 1000-1;
mTimUserHandle.Init.CounterMode = TIM_COUNTERMODE_UP;
mTimUserHandle.Init.Period = 100-1;
mTimUserHandle.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
HAL_TIM_Base_Init(&mTimUserHandle);
HAL_TIM_Base_Start_IT(&mTimUserHandle);

// Enable timer 15 interrupt
NVIC_SetVector(TIM_USR_IRQ, (uint32_t)M_TIM_USR_Handler);
NVIC_EnableIRQ(TIM_USR_IRQ);
```
Encoder Interfacing

- [https://gitlab.com/calanca/CyberPhysicalCode](https://gitlab.com/calanca/CyberPhysicalCode)

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DigitalEncoder</td>
</tr>
<tr>
<td>DigitalEncoderAB</td>
</tr>
<tr>
<td>DigitalEncoderAS5601</td>
</tr>
<tr>
<td>DigitalEncoderI2C</td>
</tr>
<tr>
<td>DigitalEncoderPIC</td>
</tr>
<tr>
<td>I2CRW</td>
</tr>
</tbody>
</table>
**Encoder Interfacing**

```cpp
/**
 * Abstract class used to implement different types of encoder
 */

enum AngleFormat {angleContinuous = 0, angleMod = 1};

class DigitalEncoder {
  public:
    virtual int getAngle() = 0;
    virtual int getMaxAngle() = 0;
    double getAngleDeg();
    double getAngleRad();
    AngleFormat getAngleFormat();
    void setAngleFormat(AngleFormat angleFormat);
  private:
    AngleFormat angleFormat = angleContinuous;
    int rev_count;
};
```
Encoder AB

class DigitalEncoderAB : public DigitalEncoder {
    public:
        DigitalEncoderAB(float cpr, float gearRatio = 1.0f);
        void reset();
        int getAngle();
        int getMaxAngle();
        void setCountPerRevolution(float cpr);
        float getCountPerRevolution();
        void setGearRatio(float gr);
        float getGearRatio();
    private:
        float cpr;
        float gearRatio;
        void EncoderInitialise_TIM2();
};
Encoder AB

```c
void DigitalEncoderAB::EncoderInitialise_TIM2() // 32 bit
{
    // configure GPIO PA0 & PA1 as inputs for Encoder
    RCC->AHB1ENR |= 0x00000001; // Enable clock for GPIOA

    GPIOA->MODER |= GPIO_MODER_MODER0_1 | GPIO_MODER_MODER1_1; // PA0 & PA1 as Alternate Function /*!< GPIO
    GPIOA->OTYPER |= GPIO_OTYPER_OT_0 | GPIO_OTYPER_OT_1; // PA0 & PA1 as Inputs /*!< GPIO
    GPIOA->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR0 | GPIO_OSPEEDER_OSPEEDR1; // Low speed /*!< GPIO
    GPIOA->PUPDR |= GPIO_PUPDR_PUPDR0_1 | GPIO_PUPDR_PUPDR1_1; // Pull Down /*!< GPIO
    GPIOA->AFR[0] |= 0x00000011; // AF01 for PA0 & PA1 /*!< GPIO
    GPIOA->AFR[1] |= 0x00000000;

    /*!< GPIO alternate function registers, Address offset: 0x20-0x24 */
    // configure GPIO PA5 & PB3 as inputs for Encoder
    __GPIOA_CLK_ENABLE();
    //__GPIOB_CLK_ENABLE();
    // configure TIM2 as Encoder input
    RCC->APB1ENR |= 0x00000001; // Enable clock for TIM2

    TIM2->CR1  = 0x0001; // CEN(Counter ENable)=’1’ ◄ TIM control register 1
    TIM2->SMCR  = 0x0003; // SMS=’011’ (Encoder mode 3) ◄ TIM slave mode control register
    TIM2->CCMR1 = 0xF1F1; // CC1S=’01’ CC2S=’01’ ◄ TIM capture/compare mode register 1
    TIM2->CCMR2 = 0x0000; // ◄ TIM capture/compare mode register 2
    TIM2->CCER  = 0x00011; // CC1P CC2P ◄ TIM capture/compare enable register
    TIM2->PSC   = 0x0000; // Prescaler = (0+1) ◄ TIM prescaler
    TIM2->ARR   = 0xffffffff; // reload at 0xffffffff ◄ TIM auto-reload register

    TIM2->CNT  = 0x0000; // reset the counter before we use it
}```
Brushless Motor Control – 6 step

```cpp
// Switch through step values to activate correct phases
// For each step we have one phase with positive voltage PWM,
switch(step_number)
{
    case 0:
        uh_1.write(pwm_positive);   //PWM
        vh_2.write(0.0f);           //DISCONNECTED
        wh_3.write(gnd_negative);   //GND
        en_1 = 1;                   //PWM
        en_2 = 0;                   //DISCONNECTED
        en_3 = 1;                   //GND
        break;

    case 1:
        uh_1.write(0.0f);          //DISCONNECTED
        vh_2.write(pwm_positive);   //PWM
        wh_3.write(gnd_negative);   //GND
        en_1 = 0;                   //DISCONNECTED
        en_2 = 1;                   //PWM
        en_3 = 1;                   //GND
        break;

    case 2:
        uh_1.write(gnd_negative);   //GND
        vh_2.write(pwm_positive);   //PWM
```
Brushless Motor Control – 6 step

```c
#define PWM_PERIOD 0.00001f
// Internal Timer 7
#define TIM_USR TIM7
#define TIM_USR_IRQ TIM7_IRQn

// Serial communication
Serial pc(USBTX, USBRX, 115200); // tx, rx, speed

// AS5601
AS5601 encoder(I2C_SDA, I2C_SCL);

// variable to keep angle value from hall sensor
float angle = 0;
// Analog input to read potentiometer
AnalogIn ain(PB_1);
// Analog input to read motor current shunt resistor
AnalogIn phase_B_curr(PC_1);
```
Brushless Motor Control – 6 step

```c
// PWM duty cycle for motor phase
float duty_cycle = 0.0f;
float pwm_positive = duty_cycle;
float gnd_negative = 0.0f;
// ON board LED
DigitalOut led1(LED1);
// Phase 1 PWM OUT
PwmOut uh_1(PA_8);
// Phase 2 PWM OUT
PwmOut vh_2(PA_9);
// Phase 3 PWM OUT
PwmOut wh_3(PA_10);
// Phase 1 ENABLE PIN
DigitalOut en_1(PC_10);
// Phase 2 ENABLE PIN
DigitalOut en_2(PC_11);
// Phase 3 ENABLE PIN
DigitalOut en_3(PC_12);
// MOTOR DRIVER CHIP ENABLE PIN
DigitalOut en_chip(PA_6);
```
Brushless Motor Control – 6 step

```c
// This function converts hall sensor's angle into 6 electrical positions of motor
void stepRead()
{
    // Check in which of 6 position the motor is
    if(position<=8.97f || position>50.5f)
    {
        step_number=5;
    }
    if(position>8.35f && position<=17.49f)
    {
        step_number=4;
    }
    if(position>16.87f && position<=26.11f)
    {
        step_number=3;
    }
    if(position>25.49f && position<=34.81f)
    {
        step_number=2;
    }
    if(position>33.84f && position<=43.2f)
    {
        step_number=1;
    }
}```
Position Control

\[ r(t) \quad e(t) \quad u(t) \quad y(t) \]

\[ \sum \quad P \quad I \quad D \]

\[ K_p e(t) \quad K_i \int e(\tau) d\tau \quad K_d \frac{de(t)}{dt} \]

**Diagram:**
- **P (Proportional):** \( K_p e(t) \)
- **I (Integral):** \( K_i \int e(\tau) d\tau \)
- **D (Derivative):** \( K_d \frac{de(t)}{dt} \)

**System Diagram:**
- \( r(t) \) to \( e(t) \) through \( \sum \)
- \( e(t) \) to \( u(t) \) through \( P \)
- \( u(t) \) to \( y(t) \) through \( \text{Plant / Process} \)

**Equations:**
- Proportional control: \( u(t) = K_p e(t) \)
- Integral control: \( u(t) = K_i \int e(\tau) d\tau \)
- Derivative control: \( u(t) = K_d \frac{de(t)}{dt} \)
PID Position Control

```c
while(1){

    position = ...;

    err = ref - position;
    ierr += err * dt;
    derr = (err - prev_err);

    output = Kp * err + Ki * ierr + Kd * derr;

    prev_err = err;
}
```