Brushless Direct Current Motor Control for Inspired Flight

Author: Aidan McCoy
Advisor: Dr. Franz J. Kurfess

June 6, 2018
Abstract

Brushless direct current motors have become a very common part of many modern electronics. Although they are more expensive and less robust than conventional brushed direct current motors, they provide large advantages in control, power output, longevity as well as efficiency. To make these motors run, they need more than just power. Each motor needs a separate motor controller that is responsible for taking in a signal for what speed/torque to run the motor at, and then managing the current flow to the motor to keep it spinning while fitting within these boundaries. Brushless direct current motors have a set of pairs of poles that act as electromagnets that need to be turned on and off at very certain times to start and then spin the motor. There are a variety of control strategies, each with their benefits and shortcomings, to facilitate startup and operation. This paper will examine fixed as well as dynamic startup strategies, as well as six step and field oriented motor control strategies with code samples to show implementations. Fixed startup is easier to implement as it just runs a startup script over and over again until there is readable feedback from the motor, but dynamic can allow for a more controlled and faster startup that rarely has to reset entirely. Once spinning, field oriented control provides the most control but requires extra hardware and/or more processing power than six step alternatives but can be challenging to set up on a new motor. For drone applications, a combination of fixed startup scripts with field oriented control makes for the best control strategy by balancing complexity with improved control when you really need it.
Contents

1. Title Page .............................................................................................................. 0
2. Abstract .............................................................................................................. 1
3. Contents ............................................................................................................. 2
4. Table of Abbreviations ....................................................................................... 3
5. Brushed Direct Current Motors ......................................................................... 4
6. Brushless Direct Current Motors ....................................................................... 5
7. Generating Control Signals .................................................................................. 5
   a. Why Use PWMs............................................................................................. 5
   b. Interpreting Control Signals .......................................................................... 6
   c. PWM Control Signal Code Example ............................................................. 7
   d. MOSFET Control ......................................................................................... 9
8. Motor Commutation Diagrams ........................................................................... 10
9. Motor Startup ..................................................................................................... 11
   a. Dynamic Startup .......................................................................................... 11
   b. Fixed Startup .............................................................................................. 11
   c. Fixed Startup Code Example ...................................................................... 12
10. Field Oriented Control ...................................................................................... 13
11. Field Oriented Control Code Example ............................................................. 14
12. Field Oriented Control Main.c ......................................................................... 18
13. References ....................................................................................................... 29
# Table of Abbreviations

1. **BLDC**  
   Brushless Direct Current Motor

2. **FOC**  
   Field Oriented Control

3. **MOSFET**  
   Metal Oxide Semiconductor Field Effect Transistor

4. **PWM**  
   Pulse Width Modulation

5. **ESC**  
   Electronic Speed Controller

6. **V**  
   Volts

7. **EMF**  
   Electromagnetic Field
Brushed Direct Current Motors

For years brushed direct current motors were the most common type of motor used for all direct current applications. They are ridiculously simple as many of us have made a basic one in a physics class at one time or another. They work by creating a magnetic fixed magnetic field with permanent magnets around coils of wire in the middle. When current is applied, a magnetic field is created by the current in the coil and the coil spins to align with the permanent magnetic field. Once the coil is about to align with the permanent field, the current in the coil switches, causing it to want to flip again and again, flipping the current each time it rotates. The current is switched mechanically by having brushes that touch the rotor only on one half as displayed in figure 1. This concept works well but has physical limitations. With only a two wire control, they have little to no speed control, with a physical rubbing between brushes and commutator, the brushes must be cleaned or replaced on a regular basis. Also as speed and torque increases, they have a hard time cooling as friction becomes a big issue. As technology progressed and microcontrollers became smaller with more processing power, a new type of motor emerged. One that gets rid of the physical connection for switching polarities by instead calculating the optimal magnetic field and creating it by turning on and off pairs of electromagnetic poles with a series of MOSFETS. Although this new technology can outperform brushed dc motors in almost every application, brushed motors are still used because they are cheap to produce, and are simple to operate even in harsh conditions.

![DC Motor Conceptual Diagram](image)

Figure 1
**Brushless Direct Current Motor (BLDC)**

BLDC motors work on the concept of almost turning brushed motors inside out. Instead of having the windings spinning with brushes to connect to them, BLDC motors place the permanent magnets on the rotors with the electromagnetic windings controlled by MOSFETs to align the magnetic field. Because there is a lot more processing that goes into BLDC motor control, the entire system becomes exponentially more complicated and can be broken into 4 parts:

1. Generating the Control Signal
2. Interpreting the signal
3. MOSFET control by rotating commutations
4. Motor response

**Generating the Control Signal**

Since this project is geared towards motor control for drones, we will focus on systems that use PWM to communicate power and speed. A PWM is a signal that is switched on and off quickly to make a square wave that is sent from the flight controller to the ESC generally across 2 wires. The ESC takes in the signal and translates it into an average voltage to use when powering the motor. This is accomplished in the following steps.

1. Read in the signal and establish what is considered “on” and “off” these will represent 1’s and 0’s. Normally this is pretty straightforward but since there is sometimes noise included in the wires, it can be a bit tougher to read.
2. Using a certain interval, determine the duty cycle. The duty cycle is the percentage of time in a given interval that the signal is “on.”
3. Next take this duty cycle percentage and multiply it by the input voltage from the power source (Normally a battery in drone applications. For this application 12V will be used) to get the average voltage to send to the motor

Example of the above steps on the next page

**Why use PWMs?**

The problem that PWMs address here is transferring a constant stream of the percentage of available voltage to use to drive the motor. PWMs with a very high frequency can provide an incredibly accurate representation of this data. An analog signal would be inaccurate as there would be noise in the line from spinning magnets as well as nearby flowing current. Also, most controllers run on either 3.3 or 5v which leaves very little room for errors. By keeping it digital until the last step, we preserve the accuracy needed while keeping the data flow constantly updating which is necessary as drone motors are constantly changing speeds to adjust for inputs and changing environments.
Interpreting the Control Signal

For the sake of this example, let’s assume the narrow pulses have a pulse length of .25 seconds, the wide pulses have a pulse length of .75 seconds and the time interval is 1 second. These numbers are enormous compared to the intervals actually used but it makes for easy to understand ratios. This figure would represent an engine throttle position being moved from slow to faster halfway through the time shown.

1. Since this square wave is clean we can assume when the signal is high it is “on” and when it is on the axis it is “off”
2. To determine the duty cycle we look at the portion of time each signal is on. For the first half with a pulse width of .25 we have a duty cycle of 25% since $0.25 \times 1 = 25$. For the second half we have a duty cycle of 75% following the same math.
3. To determine average voltage to send to the motor, we multiply these numbers by the input voltage from the battery, in this case and for the motors used in the rest of the report 12V. Therefore, for the narrow pulses (low throttle position) the motor would receive $0.25 \times 12 = 3V$ and when for the wide pulses (high throttle position) the motor would receive $0.75 \times 12 = 8V$.

Figure 3 shows a graphical representation of this relationship with average voltage sent to the motor on the top and the varying PWM signal on the bottom. This would show what the signals would look like for a throttle position that went from ~50% up to 100% back down to 0% and then back up to ~40%.
PWM Interpretation Code Example

```c
static void PWMControl(void)
{
    unsigned long duty;
    icp_sample_t input_pulse;
    input_pulse = last_icp_result;
    // For input values below the threshold set the PWM duty cycle to 0
    if (input_pulse < ICP_MIN_PULSE)
        duty = 0;
    // For input values above the minimum threshold
    else
    {
        // Limit input to maximum threshold
        if (input_pulse > ICP_MAX_PULSE)
            input_pulse = ICP_MAX_PULSE;
        // Map the input capture value to the allowable range of PWM output values from
        // MIN_PWM_COMPARE_VALUE to MAX_PWM_COMPARE_VALUE
        duty = (input_pulse - ICP_MIN_PULSE);
        duty *= MAX_PWM_COMPARE_VALUE - MIN_PWM_COMPARE_VALUE;
        duty /= ICP_MAX_PULSE - ICP_MIN_PULSE;
        duty += MIN_PWM_COMPARE_VALUE;
    }
    // Adjust the actual PWM value sent to the motors based on the calculated set point
    // TODO: switch to timer interrupt driven
    // TODO: global definition for propeller size, test different propellers to determine
    desired acceleration characteristics
    throttleFreqDivisionCounter++;
    if (throttleFreqDivisionCounter == THROTTLE_PRESCALAR)
    {
        throttleFreqDivisionCounter = 0;
        if (current_throttle < duty)
            current_throttle += 1;
        else if (current_throttle > duty)
            current_throttle -= 1;
    }
```
// When the motor is stopped
if (motorStopped)
{
    // If the throttle is above the threshold restart the motor
    if (current_throttle > MIN_PWM_COMPARE_VALUE )
    {
        motorStopped = FALSE;
        StartMotor();
        WatchdogTimerEnable();
    }
}

// While the motor is running
else
{
    // If the throttle drops below the threshold stop the motor
    if (current_throttle < MIN_PWMCOMPARE_VALUE )
    {
        motorStopped = TRUE;
        wdt_disable();
        DisableMotor();
    }
    // Otherwise the motor is running normally
    else
    {
        // Update the compare values for each PWM output based on the throttle
        OCR0B = (unsigned char)current_throttle;
        OCR2A = (unsigned char)current_throttle;
        OCR2B = (unsigned char)current_throttle;
    }
}
}
Controlling MOSFETS for 6 Step Control

Now that the correct voltage has been calculated, it now needs to be sent to the motor. To control this sequence MOSFETs are used to turn each pole on and off in a certain sequence. This sequence is broken down into 6 different combinations referred to as a commutations. For each commutation there needs to be one pole with + Vcc, one with -Vcc and one off. This allows for a creation of a magnetic field in one direction to pull the motor around. In order to tell when to switch commutations, there are multiple methods, but two common ones are the use of Hall sensors, or measuring the back EMF. Hall sensors tend to be placed every 120 degrees on the motor and are the easiest and most accurate way to tell the position of the motor. Measuring the back EMF looks at the one pole that is currently off to gauge location. As the motor spins, the magnets moving across the coils creates a voltage that can be measured. Depending on the direction of the motor this starts as a positive or negative voltage, but as it passes right over the center point, it goes to 0 and switches polarity. This is the point halfway between commutations, or 30 degrees before the next commutation. By measuring the time it took to get to the zero crossing point, it can be estimated that the next commutation should occur the same amount of time after the zero crossing point. This is the theory used in 6 step BLDC motor control.

Figure 4

Figure 4 shows a graphical representation of these 6 commutations with the back EMF shown by the solid lines and the current voltage (+, -, or off) shown by the dashed lines. These 6 commutations keep repeating as the motor spins changing the time between commutations when accelerating or decelerating by measuring the change in time it takes to reach the zero crossing point.
Figure 5 shows the physical representation of each commutation. One difference to note here is the increased number of pole pairs. This allows the motor to keep the generated magnetic field at 90 degrees creating the most amount of torque possible. Torque is created by the poles of the permanent magnet wanting to align with the opposite pole of the electromagnets. If the switching process is off by milliseconds while changing speeds or under a larger load, it can really throw off the commutation sequence impacting torque as well as efficiency, two things that are key in drone applications.
**Motor Startup**

Running the motors once they are spinning is one thing, there is feedback from sensors to time commutation switches and to tell the location of the motor, but when it is stopped, getting it started is a whole different challenge. With no rotation, the motor has no idea what position the motor is in, therefore it doesn’t know what commutation to start with. The simplest way to start a motor is a fixed start sequence that does not rely on any feedback from the motor to start. It is a carefully calibrated sequence of commutations starting slowly and working up to a speed that starts to give feedback to the controller to transition into normal operation as described above. If the start sequence is unsuccessful, it just starts over, trying again and again until a usable speed is reached. As BLDC motors become more commonplace in a variety of fields, more dynamic startup strategies have become a necessity. One of them being cars and transportation. For example, say you needed to get the motor to 100 rpm before there was noticeable feedback for Hall sensors or back EMF measurements. If every time you touched the accelerator pedal in an electric car you had to speed up to 100 rpm before anything else could happen, traffic would be a nightmare with unintentional crashes all over the place. The difference is most of these vehicles can fulfil the processing power and energy requirements of more complex strategies that the weight and size restrictions on drones cannot support.

**Dynamic Startup**

The key to dynamic start is figuring out the exact location of the motor at start time. Whatever degree it is at, the controller can then pick the corresponding commutation that corresponds with that location to get it moving and start from there. To determine location, developers have used low voltage pulses across a combination of pole pairs and measure the resulting flux to determine roughly where the rotator is located. Once it starts to move it gets easier but still challenging. At low speeds, the time between the back EMF zero crossing point and the next commutation can vary on a very large scale. This can cause the motor to shutter and even spin in the wrong direction if far enough off. Therefore at low speeds 6 step control does not work. For low speed applications, FOC becomes very important as it allows for real time monitoring of location for very accurate commutations switches rather than estimating off of zero crossing points.

**Fixed Startup**

Fixed startup is much simpler than dynamic startup. By disregarding the current location of the rotator and just starting commutation switching at a low speed, it assumes that at some point it will grab the motor and start spinning. This tends to result in shuttering, sometimes spinning the wrong direction, and sometimes not spinning at all but for some applications, including drones, this is an acceptable result. Once drone rotors start spinning, they are rarely stopped. Even when descending they are never turned completely off. Also this process only takes a quarter of a second or so depending of load, so if it fails the first and second times, there is still time to catch up before liftoff.
Fixed Startup Code Example

static void StartMotor(void)
{
    unsigned char i;
    ...
    // Set PWM outputs to the startup value
    OCR0B = STARTUP_PWM_COMPARE_VALUE;
    OCR2A = STARTUP_PWM_COMPARE_VALUE;
    OCR2B = STARTUP_PWM_COMPARE_VALUE;
    nextCommutationStep = 0;
    ...

    // Iterate through table of startup commutations
    for (i = 0; i < STARTUP_NUM_COMMUTATIONS; i++)
    {
        // Perform commutation
        PORTD = nextDrivePatternPortD;
        (nextDrivePatternPortB) ? set_bit(PORTB, CL) : clear_bit(PORTB, CL);
        TCCR0A = nextTimer0Config;
        TCCR2A = nextTimer2Config;
        StartupDelay(startupDelays[i]);
        ADMUX = ADMUXTable[nextCommutationStep];

        // Use LSB of nextCommutationStep to determine zero crossing polarity.
        zcPolarity = nextCommutationStep & 0x01;

        // Prepare for next commutation
        nextCommutationStep++;
        if (nextCommutationStep >= 6)
        {
            nextCommutationStep = 0;
        }
        nextDrivePatternPortD = driveTablePortD[nextCommutationStep];
        nextDrivePatternPortB = driveTablePortB[nextCommutationStep];
        nextTimer0Config = timer0ConfigTable[nextCommutationStep];
        nextTimer2Config = timer2ConfigTable[nextCommutationStep];
    }

    // Disable ADC so Analog Comparator can read from ADC mux
    ADCSRA &= ~(1 << ADEN);

    // Set filteredTimeSinceCommutation to the time to the next commutation.
    filteredTimeSinceCommutation = startupDelays[STARTUP_NUM_COMMUTATIONS - 1] * (STARTUP_DELAY_MULTIPLIER / 2);

    // Switch to sensorless commutation.
    TCNT1 = 0;
    SET_TIMER1_INT_COMmutation;
}
**Code Explanation**

The code above shows an example of a fixed startup algorithm. The first block sets a series of bits corresponding to the first motor commutation. As previously described, this is not necessarily the best one to start with but should be able to get the motor spinning after a few rotations. The next loop is the exact processes of going through a fixed sequence of steps. The key as it completes the first commutation, is to look at the call the checks for the zero crossing point, here labeled zcPolarity. This is where it is checking to see if there is a readable back EMF in the off pole to end the startup sequence. If it is not found, the loop continues for a designated number of attempts before considering it a failed startup. Once the loop is escaped when the back EMF is found, it switches to sensorless communication which in this case goes on a six step method of control.

**Field Oriented Control**

The FOC control strategy is one of, if not the most powerful, accurate and efficient control strategies for BLDC motors. Instead of having 6 preset steps that the motor cycles through, it constantly adjusts the power at each of the 3 phases to create a flux field that creates the most amount of power at that rotator location. As mentioned before, an electrical motor creates its maximum torque when the magnetic field created by the stator coils is orthogonal to that of the rotor. In a brushed DC motor, this only occurs twice per rotation. With the six step method, this happens 6 times each rotation. With FOC this is constant. This raises the question why don’t all motors use this if it is so efficient? Well it takes a massive amount of calculation and monitoring compared to 6 step methods. Sensors on the motor can measure the flux of any phase by looking at the current flowing through the pole. By combing the flux of all of the phases, you can determine the angle of the magnetic field created by the stator coils. As the motor spins, the controller constantly moves the generated magnetic field to be ¼ turn ahead of the rotor. This allows for smooth control when speeds are very slow as well as when they are accelerating rapidly since there is no prediction gap between cycles.

In an effort to make this process simpler, developers have come up with a solution that minimizes the calculations while the motor is running. Instead of constantly calculating flux, the controller can just track speed of the motor and lookup the phase control values corresponding with that rpm. This is not perfect, as then there are gaps between each speed lookup to determine where to set the stator field, but it utilizes the concept of FOC and allows for hundreds of commutations per rotation rather than just 6. That being said there is a great amount of tuning and calculation that goes into making one of these tabulated FOC systems work. Since the phase control values have to be precalculated, almost every possible scenario has to be planned out and have a value for the corresponding speed, load, voltage and other factors. Also storing this table takes up a decent amount of space on a controller that is already very limited on resources.
FOC Code Sample

/**
 * @brief It executes the core of FOC drive that is the controllers for Iqd
currents regulation. Reference frame transformations are carried out
accordingly to the active speed sensor. It must be called periodically
when new motor currents have been converted
 * @param this related object of class CFOC.
 * @retval int16_t It returns MC_NO_FAULTS if the FOC has been ended before
next PWM Update event, MC_FOC_DURATION otherwise
 */
#pragma inline
uint16_t FOC_CurrController(uint8_t bMotor)
{
    Curr_Components Iab, Ialphabeta, Iqd;
    Volt_Components Valphabeta, Vqd;
    int16_t hElAngledpp;
    uint16_t hCodeError;

    hElAngledpp = SPD_GetElAngle(STC_GetSpeedSensor(pSTC[bMotor]));
    PWMC_GetPhaseCurrents(pwmcHandle[bMotor], &Iab);
    Ialphabeta = MCM_Clarke(Iab);
    Iqd = MCM_Park(Ialphabeta, hElAngledpp);
    Vqd.qV_Component1 = PI_Controller(pPIDIq[bMotor],
        (int32_t)(FOCVars[bMotor].Iqdref.qI_Component1) - Iqd.qI_Component1);
    Vqd.qV_Component2 = PI_Controller(pPIDId[bMotor],
        (int32_t)(FOCVars[bMotor].Iqdref.qI_Component2) - Iqd.qI_Component2);
    FOCVars[bMotor].Vqd = Vqd;
    Vqd = Circle_Limitation(pCLM[bMotor], Vqd);
    Valphabeta = MCM_Rev_Park(Vqd, hElAngledpp);
    hCodeError = PWMC_SetPhaseVoltage(pwmcHandle[bMotor], Valphabeta);
    FOCVars[bMotor].Iab = Iab;
    FOCVars[bMotor].Ialphabeta = Ialphabeta;
    FOCVars[bMotor].Iqd = Iqd;
    FOCVars[bMotor].Valphabeta = Valphabeta;
    FOCVars[bMotor].hElAngle = hElAngledpp;
    return(hCodeError);
}
Even though FOC can handle low speeds with great control, it is still easiest to follow a relatively fixed startup sequence. The main difference is the startup sequence can transfer from six step like sequencing to start the first few rotations then transition into FOC control to get it up to a safe operating speed and then take over full FOC control afterwards. This sequence can be seen here with a case statement breaking up the sections.

(...)  
    case START:
    {
    / only for sensor-less control */
        int16_t hForcedMecSpeed01Hz;
        Curr_Components IqdRef;
        bool StartUpTransitionEnded;
        bool StartUpDoTransition;
        if(!RUC_Exec(&RevUpControlM1))
        {
            STM_FaultProcessing(&STM[M1], MC_START_UP, 0);
            /*Time allowed for startup has ended*/
        }
        else
        {
            if (SWO_transitionStartM1 == false)
            {
                IqdRef.qI_Component1 = STC_CalcTorqueReference(pSTC[M1]);
                IqdRef.qI_Component2 = FOCVars[M1].UserIdref;
                FOCVars[M1].Iqdref = IqdRef;
            }
            StartUpTransitionEnded = VSS_CalcAvrgMecSpeed01Hz(&VirtualSpeedSensorM1,&hForcedMecSpeed01Hz);
            StartUpDoTransition = VSS_SetStartTransition(&VirtualSpeedSensorM1,STO_PLL_IsObserverConverged(&STO_PLL_M1,hForcedMecSpeed01Hz));
            if (VSS_IsTransitionOngoing(&VirtualSpeedSensorM1))
            {
                if (SWO_transitionStartM1 == false)
                {
                    int16_t Iq = 0;
                    Curr_Components StatorCurrent = MCM_Park(FOCVars[M1].Ialphabeta,
                    SPD_GetElAngle(&STO_PLL_M1._Super));
                    Iq = StatorCurrent.qI_Component1;
                    REMNG_Init(pREMNG[M1]);
                    REMNG_ExecRamp(pREMNG[M1], FOVCVars[M1].Iqdref.qI_Component1, 0);
                    REMNG_ExecRamp(pREMNG[M1], Iq, TRANSITION_DURATION);
                    SWO_transitionStartM1 = true;
                }
            }
        }
    }
else
{
    if (SWO_transitionStartM1 == true)
    {
        SWO_transitionStartM1 = false;
    }
}

if (StartUpDoTransition == false)
{
    StartUpTransitionEnded = true;
}

if (StartUpTransitionEnded == true)
{
    if (PID_SPEED_INTEGRAL_INIT_DIV == 0)
    {
        PID_SetIntegralTerm(pPIDSpeed[M1], 0);
    }
    else
    {
        PID_SetIntegralTerm(pPIDSpeed[M1], (int32_t)(FOCVars[M1].Iqref.qI_Component1*PID_GetKIDvisor(pPIDSpeed[M1])/PID_SPEED_INTEGRAL_INIT_DIV));
    }
    STM_NextState(&STM[M1], START_RUN);
}
break;

case START_RUN:
  /* only for sensor-less control */
  STC_SetSpeedSensor(pSTC[M1], &STO_PLL_M1._Super); /*Observer has converged*/
  {
      FOC_InitAdditionalMethods(M1);
      FOC_CalcCurrRef(M1);
      STM_NextState(&STM[M1], RUN);
  }
  STC_ForceSpeedReferenceToCurrentSpeed(pSTC[M1]); /* Init the reference speed to current speed */
      MCI_ExecBufferedCommands(oMCInterface[M1]); /* Exec the speed ramp after changing of the speed sensor */

  break;

case RUN:
  MCI_ExecBufferedCommands(oMCInterface[M1]);
  FOC_CalcCurrRef(M1);
  if(!IsSpeedReliable)
{  
  STM_FaultProcessing(&STM[M1], MC_SPEED_FDBK, 0);
}

break;
Complete Main.c Code for FOC Control

/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "stm32f3xx_hal.h"
#include "motorcontrol.h"

/* Private variables --------------------------------------------------------*/
ADC_HandleTypeDef hadc1;
ADC_HandleTypeDef hadc2;
DAC_HandleTypeDef hdac;
TIM_HandleTypeDef htim1;
UART_HandleTypeDef huart1;

/* Private function prototypes ---------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
static void MX_ADC2_Init(void);
static void MX_DAC_Init(void);
static void MX_TIM1_Init(void);
static void MX_USART1_UART_Init(void);
static void MX_NVIC_Init(void);

void HAL_TIM_MspPostInit(TIM_HandleTypeDef *htim);

/**
 * @brief The application entry point.
 *
 * @retval None
 */
int main(void)
{

/* MCU Configuration--------------------------------------------------------*/

/* Reset of all peripherals, Initializes the Flash interface and the Systick. */
HAL_Init();

/* Configure the system clock */
SystemClock_Config();

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_ADC1_Init();
MX_ADC2_Init();
MX_DAC_Init();
MX_TIM1_Init();
MX_USART1_UART_Init();
MX_MotorControl_Init();

/* Initialize interrupts */
MX_NVIC_Init();

} /* End of System Clock Configuration section */

/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{

    RCC_OscInitTypeDef RCC_OscInitStruct;
    RCC_ClkInitTypeDef RCC_ClkInitStruct;
    RCC_PeriphCLKInitTypeDef PeriphClkInit;

    /**Initializes the CPU, AHB and APB busses clocks */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
    RCC_OscInitStruct.HSEState = RCC_HSE_ON;
    RCC_OscInitStruct.HSEPredivValue = RCC_HSE_PREDIV_DIV1;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
    RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL9;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }

    /**Initializes the CPU, AHB and APB busses clocks */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                               |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_APB1_DIV2;
    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }

} /* End of System Clock Configuration section */
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART1|RCC_PERIPHCLK_TIM1;
PeriphClkInit.Usart1ClockSelection = RCC_USART1CLKSOURCE_PCLK2;
PeriphClkInit.Tim1ClockSelection = RCC_TIM1CLK_HCLK;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

/**Enables the Clock Security System
 */
HAL_RCC_EnableCSS();

/**Configure the Systick interrupt time
 */
HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);

/**Configure the Systick
 */
HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);

/* SysTick_IRQn interrupt configuration */
HAL_NVIC_SetPriority(SysTick_IRQn, 4, 0);
}

/**
 * @brief NVIC Configuration.
 * @retval None
 */
static void MX_NVIC_Init(void)
{
    /* TIM1_BRK_TIM15_IRQn interrupt configuration */
    HAL_NVIC_SetPriority(TIM1_BRK_TIM15_IRQn, 4, 1);
    HAL_NVIC_EnableIRQ(TIM1_BRK_TIM15_IRQn);

    /* ADC1_2_IRQn interrupt configuration */
    HAL_NVIC_SetPriority(ADC1_2_IRQn, 2, 0);
    HAL_NVIC_EnableIRQ(ADC1_2_IRQn);

    /* USART1_IRQn interrupt configuration */
    HAL_NVIC_SetPriority(USART1_IRQn, 3, 1);
HAL_NVIC_EnableIRQ(USART1_IRQn);
}

/* ADC1 init function */
static void MX_ADC1_Init(void)
{

ADC_MultiModeTypeDef multimode;
ADC_InjectionConfTypeDef sConfigInjected;
ADC_ChannelConfTypeDef sConfig;

/**Common config */

hadc1.Instance = ADC1;
hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV1;
hadc1.Init.Resolution = ADC_RESOLUTION_12B;
hadc1.Init.ScanConvMode = ADC_SCAN_ENABLE;
hadc1.Init.ContinuousConvMode = DISABLE;
hadc1.Init.DiscontinuousConvMode = DISABLE;
hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
hadc1.Init.DataAlign = ADC_DATAALIGN_LEFT;
hadc1.Init.NbrOfConversion = 2;
hadc1.Init.DMAContinuousRequests = DISABLE;
hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
hadc1.Init.LowPowerAutoWait = DISABLE;
hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
if (HAL_ADC_Init(&hadc1) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

/**Configure the ADC multi-mode */
multimode.Mode = ADC_MODE_INDEPENDENT;
if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

/**Configure Injected Channel */
sConfigInjected.InjectedChannel = ADC_CHANNEL_8;
sConfigInjected.InjectedRank = ADC_INJECTED_RANK_1;
sConfigInjected.InjectedSingleDiff = ADC_SINGLE_ENDED;
sConfigInjected.InjectedNbrOfConversion = 2;
sConfigInjected.InjectedSamplingTime = ADC_SAMPLETIME_7CYCLES_5;
sConfigInjected.ExternalTrigInjecConvEdge = ADC_EXTERNALTRIGINJECCONV_EDGE_RISING;
sConfigInjected.ExternalTrigInjecConv = ADC_EXTERNALTRIGINJECCONV_T1_TRGO;
sConfigInjected.AutoInjectedConv = DISABLE;
sConfigInjected.InjectedDiscontinuousConvMode = DISABLE;
sConfigInjected.QueueInjectedContext = ENABLE;
sConfigInjected.InjectedOffset = 0;
sConfigInjected.InjectedOffsetNumber = ADC_OFFSET_NONE;
if (HAL_ADCEx_InjectedConfigChannel(&hadc1, &sConfigInjected) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

/**Configure Injected Channel */
sConfigInjected.InjectedChannel = ADC_CHANNEL_9;
sConfigInjected.InjectedRank = ADC_INJECTED_RANK_2;
if (HAL_ADCEx_InjectedConfigChannel(&hadc1, &sConfigInjected) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

/**Configure Regular Channel */
sConfig.Channel = ADC_CHANNEL_1;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.SamplingTime = ADC_SAMPLETIME_61CYCLES_5;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

/**Configure Regular Channel */
sConfig.Channel = ADC_CHANNEL_5;
sConfig.Rank = ADC_REGULAR_RANK_2;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

/* ADC2 init function */
static void MX_ADC2_Init(void)
ADC_InjectionConfTypeDef sConfigInjected;

/**Common config*/
hadc2.Instance = ADC2;
hadc2.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV1;
hadc2.Init.Resolution = ADC_RESOLUTION_12B;
hadc2.Init.ScanConvMode = ADC_SCAN_ENABLE;
hadc2.Init.ContinuousConvMode = DISABLE;
hadc2.Init.DiscontinuousConvMode = DISABLE;
hadc2.Init.DataAlign = ADC_DATAALIGN_LEFT;
hadc2.Init.NbrOfConversion = 1;
hadc2.Init.DMAContinuousRequests = DISABLE;
hadc2.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
hadc2.Init.LowPowerAutoWait = DISABLE;
hadc2.Init.Overrun = ADC_OVR_DATA_PRESERVED;
if (HAL_ADC_Init(&hadc2) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

/**Configure Injected Channel*/
sConfigInjected.InjectedChannel = ADC_CHANNEL_9;
sConfigInjected.InjectedRank = ADC_INJECTED_RANK_1;
sConfigInjected.InjectedSingleDiff = ADC_SINGLE_ENDED;
sConfigInjected.InjectedNbrOfConversion = 2;
sConfigInjected.InjectedSamplingTime = ADC_SAMPLETIME_7CYCLES_5;
sConfigInjected.ExternalTrigInjecConvEdge = ADC_EXTERNALTRIGINJECCONV_EDGE_RISING;
sConfigInjected.ExternalTrigInjecConv = ADC_EXTERNALTRIGINJECCONV_T1_TRGO;
sConfigInjected.AutoInjectedConv = DISABLE;
sConfigInjected.InjectedDiscontinuousConvMode = DISABLE;
sConfigInjected.QueueInjectedContext = ENABLE;
sConfigInjected.InjectedOffset = 0;
sConfigInjected.InjectedOffsetNumber = ADC_OFFSET_NONE;
if (HAL_ADCEx_InjectedConfigChannel(&hadc2, &sConfigInjected) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

/**Configure Injected Channel*/
sConfigInjected.InjectedChannel = ADC_CHANNEL_6;
sConfigInjected.InjectedRank = ADC_INJECTED_RANK_2;
if (HAL_ADCEx_InjectedConfigChannel(&hadc2, &sConfigInjected) != HAL_OK)
{   
    _Error_Handler(__FILE__, __LINE__);  
}

/* DAC init function */
static void MX_DAC_Init(void) {

    DAC_ChannelConfTypeDef sConfig;

    /**DAC Initialization */
    hdac.Instance = DAC;
    if (HAL_DAC_Init(&hdac) != HAL_OK) {
        _Error_Handler(__FILE__, __LINE__);  
    }

    /**DAC channel OUT1 config */
    sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
    sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
    if (HAL_DAC_ConfigChannel(&hdac, &sConfig, DAC_CHANNEL_1) != HAL_OK) {
        _Error_Handler(__FILE__, __LINE__);  
    }

    /**DAC channel OUT2 config */
    if (HAL_DAC_ConfigChannel(&hdac, &sConfig, DAC_CHANNEL_2) != HAL_OK) {
        _Error_Handler(__FILE__, __LINE__);  
    }

} /* TIM1 init function */
static void MX_TIM1_Init(void) {

    TIM_SlaveConfigTypeDef sSlaveConfig;
    TIM_MasterConfigTypeDef sMasterConfig;
    TIM_OC_InitTypeDef sConfigOC;
    TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig;

    TIM_SlaveConfigTypeDef sSlaveConfig;
    TIM_MasterConfigTypeDef sMasterConfig;
    TIM_OC_InitTypeDef sConfigOC;
    TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig;
htim1.Instance = TIM1;
htim1.Init.Prescaler = 0;
htim1.Init.CounterMode = TIM_COUNTERMODE_CENTERALIGNED1;
htim1.Init.Period = 2250;
htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV2;
htim1.Init.RepetitionCounter = 1;
htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_Base_Init(&htim1) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

if (HAL_TIM_PWM_Init(&htim1) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

sSlaveConfig.SlaveMode = TIM_SLAVEMODE_TRIGGER;
sSlaveConfig.InputTrigger = TIM_TS_ITR1;
if (HAL_TIM_SlaveConfigSynchronization(&htim1, &sSlaveConfig) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

sMasterConfig.MasterOutputTrigger = TIM_TRGO_OC4REF;
sMasterConfig.MasterOutputTrigger2 = TIM_TRGO2_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

sConfigOC.OCMode = TIM_OCMODE_PWM1;
sConfigOC.Pulse = 0;
sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
sConfigOC.OCNIdleState = TIM_OCNIDLSTATE_RESET;
if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_2) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}
if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_3) != HAL_OK)
{
>Error_Handler(__FILE__, __LINE__);
}

sConfigOC.OCMode = TIM_OCMODE_PWM2;
sConfigOC.Pulse = 2124;
if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_4) != HAL_OK)
{
.getErrorHandler(__FILE__, __LINE__);
}

sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_ENABLE;
sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_ENABLE;
sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_1;
sBreakDeadTimeConfig.DeadTime = 28;
sBreakDeadTimeConfig.BreakState = TIM_BREAK_ENABLE;
sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
sBreakDeadTimeConfig.BreakFilter = 0;
sBreakDeadTimeConfig.Break2State = TIM_BREAK2_ENABLE;
sBreakDeadTimeConfig.Break2Polarity = TIM_BREAK2POLARITY_LOW;
sBreakDeadTimeConfig.Break2Filter = 0;
sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_DISABLE;
if (HAL_TIMEx_ConfigBreakDeadTime(&htim1, &sBreakDeadTimeConfig) != HAL_OK)
{
.getErrorHandler(__FILE__, __LINE__);
}

HAL_TIM_MspPostInit(&htim1);

/* USART1 init function */
static void MX_USART1_UART_Init(void)
{

huart1.Instance = USART1;
huart1.Init.BaudRate = 115200;
huart1.Init.WordLength = UART_WORDLENGTH_8B;
huart1.Init.StopBits = UART_STOPBITS_1;
huart1.Init.Parity = UART_PARITY_NONE;
huart1.Init.Mode = UART_MODE_TX_RX;
huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
huart1.Init.OverSampling = UART_OVERSAMPLING_16;
huart1.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
if (HAL_UART_Init(&huart1) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

/** Configure pins as
 * Analog
 * Input
 * Output
 * EVENT_OUT
 * EXTI
 */

static void MX_GPIO_Init(void)
{

    GPIO_InitTypeDef GPIO_InitStruct;

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOE_CLK_ENABLE();
    __HAL_RCC_GPIOF_CLK_ENABLE();
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOA_CLK_ENABLE();

    /*Configure GPIO pin : Start_Stop_Pin */
    GPIO_InitStruct.Pin = Start_Stop_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_EVT_FALLING;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(Start_Stop_GPIO_Port, &GPIO_InitStruct);
}

/**
 * @brief  This function is executed in case of error occurrence.
 * @param  file: The file name as string.
 * @param  line: The line in file as a number.
 * @retval None
 */

void _Error_Handler(char *file, int line)
{
}

#ifdef USE_FULL_ASSERT
/**
 * @brief  Reports the name of the source file and the source line number

* where the assert_param error has occurred.
* @param  file: pointer to the source file name
* @param  line: assert_param error line source number
* @retval None
*/

void assert_failed(uint8_t* file, uint32_t line)
{
}
#endif /* USE_FULL_ASSERT */
References


