
Motor System IC TLE956x

Infineon Technologies AG

Mar 03, 2022

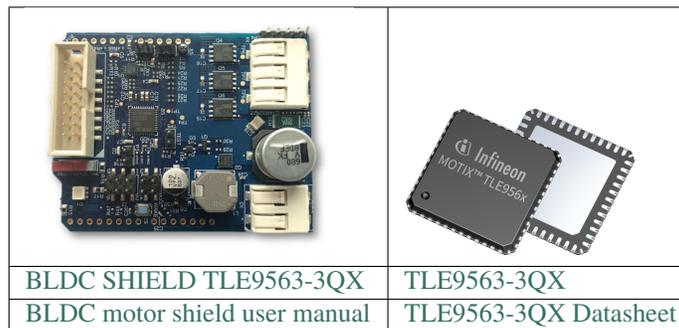
1	BLDC Motor Shield with TLE9563	3
2	DC Motor Shield with TLE9562	5
3	License	7
3.1	Acronyms	7
3.2	Related Links	7
3.2.1	Related Products	7
3.2.2	Related Repositories	8
3.2.3	More	8
3.3	BLDC Motor Shield with TLE9563	8
3.3.1	Pinout Diagram	8
3.3.2	Pin Description	9
3.3.3	Jumper settings	9
3.4	DC Motor Shield with TLE9562	10
3.4.1	Pinout Diagram	10
3.4.2	Pin Description	10
3.4.3	Jumper Settings	11
3.5	Library Architecture	11
3.5.1	Core Library	12
3.5.2	Platform Abstraction Layer (PAL) Interface	13
3.5.3	Framework PAL	13
3.5.4	Framework API Wrapper	13
3.5.5	Predefined Hardware Platforms	14
3.6	Porting Guide	14
3.6.1	Framework PAL Implementation	14
3.6.2	Framework API Wrapper	14
3.7	Adaptive Gate Control (AGC)	15
3.7.1	Example codes	15
3.7.2	Parameter defines	16
3.8	BLDC Motor Tuning	17
3.8.1	Safety	17
3.8.2	RPM Regulation	17
3.8.3	Startup	17
3.9	Arduino Getting Started	17
3.9.1	Arduino Compatible Kits	17
3.9.2	Arduino Library Installation	18
3.9.3	Arduino API	19
3.9.4	Arduino Examples	24
3.9.5	Software	25

3.9.6	Hardware	25
3.9.7	Ready!	25

Index		27
--------------	--	-----------

Welcome to the Infineon Motor System IC TLE956x library docs!

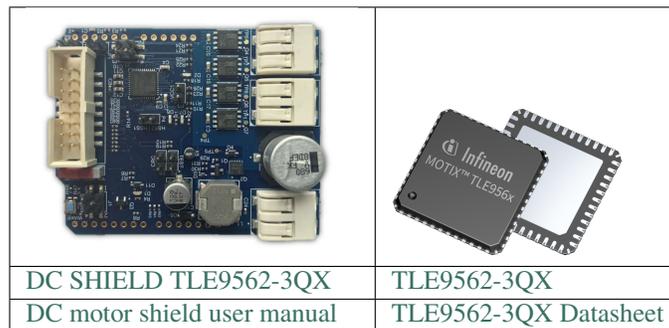
BLDC MOTOR SHIELD WITH TLE9563



Main Features:

- support for sensor-less brushless motors using onboard BEMF comparator
- support for brushless motors with hall-sensor (field weakening range possible)
- RPM function that keeps a desired RPM speed
- control onboard RGB LED
- platform independent C++ architecture
- various parameters configurable for adaptive gate control (AGC)

DC MOTOR SHIELD WITH TLE9562



Main Features:

- control two DC motors independently of each other
- control onboard LEDs
- platform independent C++ architecture
- various parameters configurable for adaptive gate control (AGC)

Please find the license file for this library [here](#).

3.1 Acronyms

Short	Long term
AGC	Adaptive Gate Control
API	Application Programming Interface
BEMF	Back-Electromotive Force
BLDCM	Brushless Direct Current Motor
CRC	Cyclic Redundancy Check
CSA	Current Sense Amplifier
FOC	Field-Oriented Control
HAL	Hardware Abstracted Layer
HSS	High-Side Switch
PAL	Platform Abstraction Layer
PWM	Pulse Width Modulation
RPM	Rounds per Minute
SBC	System Basis Chip
SPI	Serial Peripheral Interface
XFP	Cross Framework Platform

3.2 Related Links

3.2.1 Related Products

- [XMC1100 Boot Kit](#)
- [XMC4700 Relax Kit](#)
- [Arduino Uno Rev3](#)
- [Arduino IDE](#)

3.3.2 Pin Description

Arduino Pin	Symbol	Type	Function
GND	GND	-	Ground
D2	INTN	Output/digital	Interrupt output of TLE956x configurable via SPI.
D3	PWM1	Input/PWM	Input PWM for Phase 1
D5	BEMF_U_IO	Output/digital	Output of BEMF comparator for Phase 1
D6	BEMF_V_IO	Output/digital	Output of BEMF comparator for Phase 2
D7	BEMF_W_IO	Output/digital	Output of BEMF comparator for Phase 3
D8	SPI CSN	Input/digital	Chip select pin for SPI communication
D9	PWM 3	Input/PWM	Input PWM for Phase 2
D10	PWM5	Input/PWM	Input PWM for Phase 3
D11	SPI SDI	Input/digital	Serial Data Input from Microcontroller to TLE
D12	SPI SDO	Output/digital	Serial Data Output from TLE to Microcontroller
D13	SPI CLK	Input/digital	SPI Clock
A0	Hall C	Output/digital	Signal of Hallsensor C (from BLDC motor)
A1	Hall B	Output/digital	Signal of Hallsensor B (from BLDC motor)
A2	Hall A	Output/digital	Signal of Hallsensor A (from BLDC motor)
A5	CSO	Output/analog	Output of current sense amplifier

3.3.3 Jumper settings

For plug & play operation with the provided example code, it's recommended to set the default jumpers:

Jumper	Default	Function
J1	None (only in-/output)	In and output of the high speed CAN transceiver
J2	set 3 Jumpers	Connect each High-Side-Switch of the TLE9563 with a color of the RGB LED
P1	set	Connect RESET of Arduino with RSTN of TLE9563
P2	set	Connect the INTN pin of TLE9563 with a pulldown to GND. This enables software development mode.
P3	set	Connect green LED with VCC1 of TLE9563 indicating chip is powered on
P4	None (only input)	Connector for hall-sensor
P5	set	Connect VIN of Arduino with 5V regulator on TLE9563 shield

For more information refer to the [BLDC motor shield user manual](#) and [TLE9563-3QX Datasheet](#).

3.4 DC Motor Shield with TLE9562

3.4.1 Pinout Diagram

The

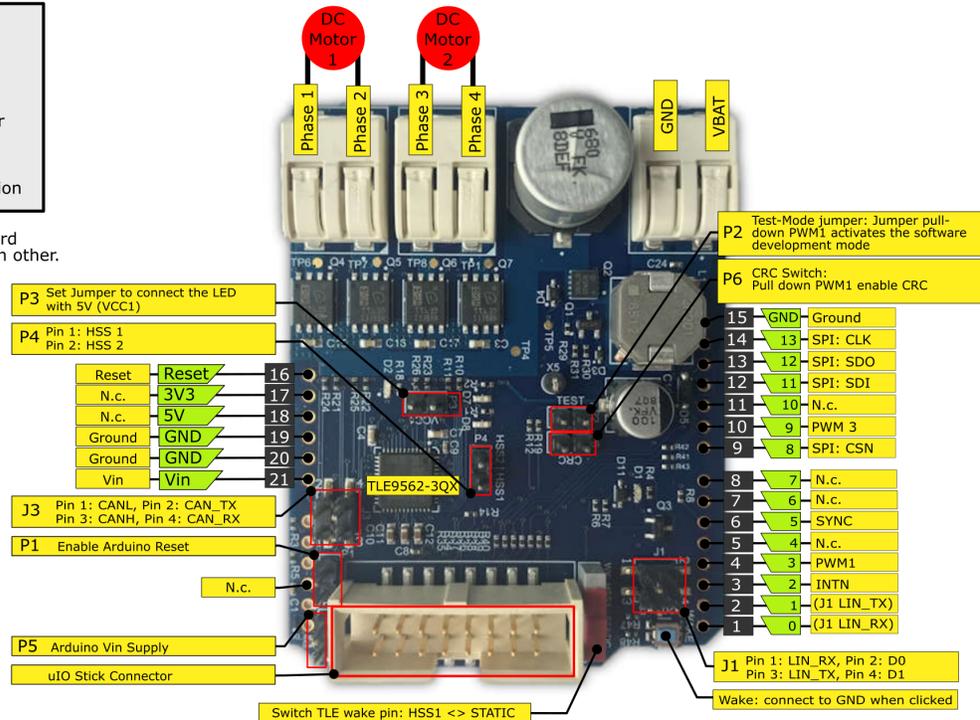
TLE9562 DC Motor Control Shield



Legend

- Information
- Labelling of Pins
- Pin Number Arduino
- Physical Pin Number
- Warning
- Additional Information

Ground pins on board connected with each other.



If INTN pin for TLE-error detection is used, the wake-switch has to be in place "STATIC".

V1.0.0

www.infineon.com

3.4.2 Pin Description

Arduino Pin	Symbol	Type	Function
GND	GND	-	Ground
D2	INTN	Output/digital	Interrupt output of TLE956x configurable via SPI.
D3	PWM1	Input/PWM	Input PWM for Phase 1
D5	SYNC	Input/digital	Synchronization for wake input
D8	SPI CSN	Input/digital	Chip select pin for SPI communication
D9	PWM 3	Input/PWM	Input PWM for Phase 2
D11	SPI SDI	Input/digital	Serial Data Input from Microcontroller to TLE
D12	SPI SDO	Output/digital	Serial Data Output from TLE to Microcontroller
D13	SPI CLK	Input/digital	SPI Clock

3.4.3 Jumper Settings

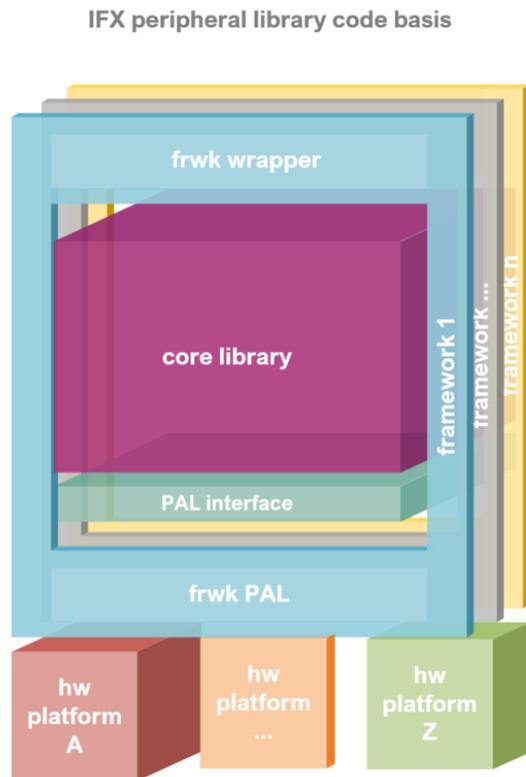
For plug & play operation with the provided example code, it's recommended to set the default jumpers:

Jumper	Default	Function
J1	None	Connect D0 and D1 with LIN transceiver
J3	None (only in-/output)	In and output of the high speed CAN transceiver
P1	set	Connect RESET of Arduino with RSTN of TLE9562
P2	set	Connect the INTN pin of TLE9562 with a pulldown to GND. This enables software development mode.
P3	set	Connect green LED with VCC1 of TLE9562 indicating chip is powered on
P4	None (only out-put)	Output of remaining High-Side-Switches HSS1 and HSS2.
P5	set	Connect VIN of Arduino with 5V regulator on TLE9562 shield
P6	None	Connect the PWM1 pin of TLE9562 with a pulldown to GND. This enables CRC.

For more information refer to the [DC motor shield user manual](#) and [TLE9562-3QX Datasheet](#).

3.5 Library Architecture

The TLE956x Motor System IC library follows the architecture pattern shown in the stack diagram:



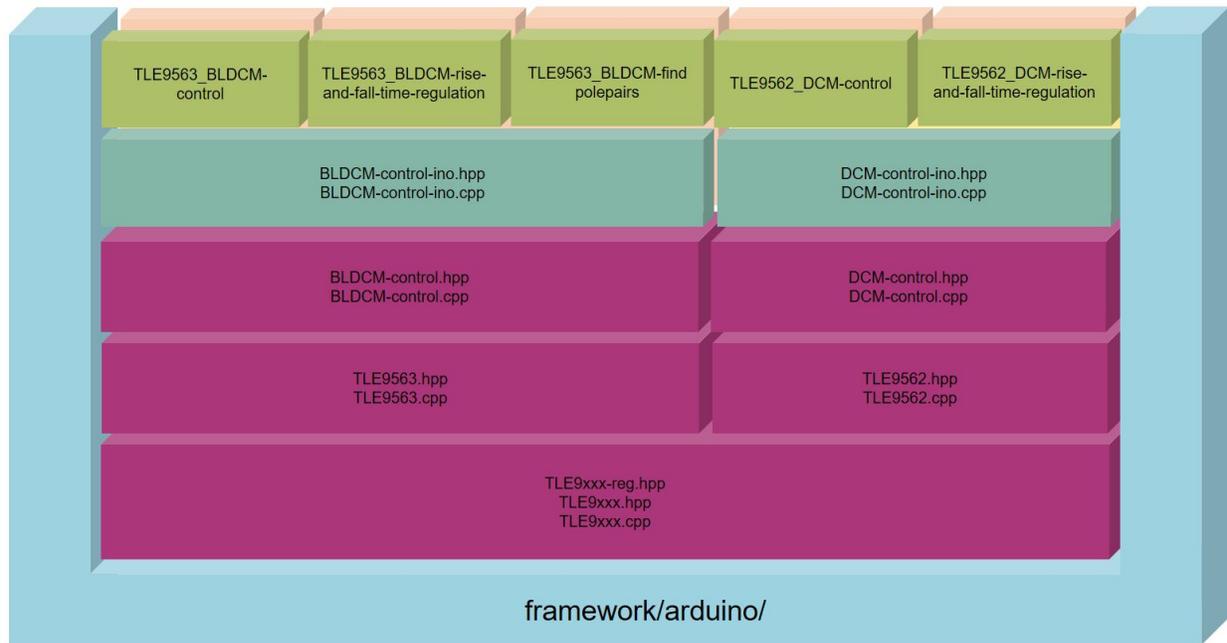
The monolithic core library can be universally integrated across any low level peripheral drivers, embedded operating system, and middleware of each software framework.

The reusability and interoperability is achieved by defining a Platform Abstraction Layer (PAL) interface which is implemented by each framework for its specific hardware abstraction layer and operating system resources APIs.

Additionally, the core library API is accommodated and adapted to the particularities of each software framework. The framework API wrappers intend to harmonize the core library API with that of the software development framework in which it is integrated, making it easier for the users already familiarized with the development framework.

The support for the multiple hardware platforms is then provided by each development framework.

The specific TLE956x Motor System IC library modules are depicted more specifically below:



In the next sections, the information of the main architecture modules for the TLE956x Motor System IC library is extended.

3.5.1 Core Library

The core library contains all the library logic and high level functionalities of the gTLE956x Motor System IC peripheral. The core library remains C++ agnostic by interacting with the specific platform (and framework) through a Platform Abstraction Layer interface. Almost no other dependencies than standard C/C++ modules and the PAL are found in these sources, only the Serial.print() is an Arduino specific command that is still needed for error communication.

There is one base class that is necessary for all applications with this chip:

```
class Tle9xxx  
    Subclassed by Tle9562, Tle9563
```

There are two derived classes, that represent a specific version of the chip.

```
class Tle9562 : public Tle9xxx  
class Tle9563 : public Tle9xxx
```

Furthermore there are classes for each board available, that use instances of the TLE9562 / TLE9563 classes.

class **DCMcontrol**
 Subclassed by *DCMcontrolIno*

class **BLDCMcontrol**
 Subclassed by *BLDCMcontrolIno*

These classes are then derived to be specific for Arduino UNO. Here nothing but the pin definition will happen:

```
class DCMcontrolIno : public DCMcontrol
class BLDCMcontrolIno : public BLDCMcontrol
```

These code sources can be found under “src/corelib”.

3.5.2 Platform Abstraction Layer (PAL) Interface

The Platform Abstraction Layer Interface is implemented via abstract C++ classes declaring all the necessary platform resources and functionalities that need to be provided by the specific framework-platform implementation.

The simple BLDCMcontrol specifies in its PAL modules an ADC class, a GPIO class, and a Timer class. Its implementation is located in the “src/pal” folder.

3.5.3 Framework PAL

The PAL interfaces is defined for each embedded software framework through its low level peripheral drivers and operating system resources APIs. The ADC, GPIO and Timer interface abstract classes are inherited and defined in this layer.

The “src/framework/sample_fmwk/pal” folder contains the pal implementation for the particular framework.

Find more information about the supported software development frameworks in the Software Frameworks section.

3.5.4 Framework API Wrapper

The idea behind this layer is to adapt the library in order to comply with the programming conventions of the integrated programming framework or ecosystem.

Operating system libraries, low level driver of hardware peripherals (digital input/outputs, PWM, analog conversion, etc.) or other middleware resources are implemented for each development framework following certain patterns for functions, parameters, and primitive types.

At these level, certain platform functionalities can be already defined and adapted to the API available resources: functions prototypes, framework core libraries, low level driver HAL, programming patterns, and even framework feel and look aspects.

The frameworks wrapper API files and pin configuration are located in the “src/framework/sample_fmwk/wrapper” folders.

Find more information about each software development frameworks API in the Software Frameworks section.

3.5.5 Predefined Hardware Platforms

Given a particular hardware platform and development software framework, most of the resources and its configuration can be already determined by default. This part of the framework API just define some pre configured instances for common and officially supported evaluation kits based configuration.

These instances are available in the “src/framework/sample_fmwk/wrapper/zzz-platf-xxx.hpp/cpp” source files.

3.6 Porting Guide

Porting the library to a new software development framework and hardware platform entails the implementation of the corresponding ADC, GPIO and Timer PAL classes. In the following sections, some additional explanations and hints are provided:

3.6.1 Framework PAL Implementation

Implement the abstract PAL interface for you framework. The **ADC class**, **GPIO class** and **Timer class** are mandatory.

The Doxygen comments on the “src/pal/adc.hpp”, “src/pal/gpio.hpp” and “src/pal/timer.hpp” describe the required behavior of each function of the PAL Interface.

Consider the existing framework implementations as reference examples for you design: “src/framework/sample_fmwk/pal”. Some of the functions are optional depending on your framework and intended usage of the library.

That is the case of *init()* and *deinit()*, which take care of the hardware peripherals init/deinitialization. If this is done in your main application (or somewhere else outside the library), there is no need of delegating such initialization to the High-side Switch library. The definition of these functions can just be a return with the success return code.

3.6.2 Framework API Wrapper

The framework API wrapper implementation is optional, it is meant to ease the usage. Mostly the main help is to avoid the creation of the ADC, GPIO and Timer object instances for the developer.

To illustrate this approach, it is easier to evaluate a concrete implementation of the Arduino wrapper. For example have a look in “src/corelib/DCM-control.cpp”:

```
pwmA->ADCWrite(_DutyCycle);  
pwmB->ADCWrite(_DutyCycle);
```

is wrapped for Arduino like this:

```
analogWrite(ARDUINO_UNO.PWM_U, _DutyCycle);  
analogWrite(ARDUINO_UNO.PWM_V, _DutyCycle);
```

using this instantiation in “src/DCM-control-ino.cpp”:

```
DCMcontrol::pwmA = new ADCIno(ARDUINO_UNO.PWM_U);  
DCMcontrol::pwmB = new ADCIno(ARDUINO_UNO.PWM_V);  
  
DCMcontrol::timer = new TimerIno();
```

where the pin configuration is stored in the ARDUINO_UNO struct in “*src/framework/arduino/wrapper*”.

While it does not seem to simplify much in number of arguments, an Arduino developer can simply pass the pin number as argument, and does not need to deal with the (probably unknown) GPIO classes, neither specify further GPIO configuration as the mode (input, output, pull-up..), positive/negative logic, etc in the core library.

For each ecosystem and framework, any other criteria can be chosen, hopefully matching as well its code conventions, implementation principles and paradigms.

3.7 Adaptive Gate Control (AGC)

The Infineon’s Motor System ICs (TLE956x) and Multi MOSFET driver ICs (TLE9210x) include a MOSFET driver with multi-stage current source gate control which is configured over SPI. Over this interface, the turn-on (tDONx) and turn-off (tDOFFx) delay can be controlled in PWM operation, as well as the rise (tRISEx) and fall (tFALLx) times. Therefore the algorithms explained in [Rise fall time regulation with current source MOSFET gate drivers](#) were implemented in this library.

3.7.1 Example codes

There are two software examples available, that directly execute the regulation like shown in the animated GIF.

- [/examples/TLE9562_DCM_rise-and-fall-time-regulation.ino](#) for use with DC motors and static loads
- [/examples/TLE9563_BLDCM_rise-and-fall-time-regulation.ino](#) for use with BLDC motors

For optimal results it’s recommended to use **TLE9562_DCM_rise-and-fall-time-regulation.ino** with an ideal R-L Load instead of a real motor to avoid side effects. This code can be used on both TLE956x boards despite the specific name.

```
#define HALFBRIDGE          PHASE1      // [PHASE1;Phase4] Select the phase on which
↳you want to regulate Rise/Fall time
#define SPEED_INCREASE_STEP 100        // [1;511] speed step increase/decrease when
↳pressing a key
#define CONTROL_LOOP_DELAY 400         // [ms] time between regulation executions
```

First choose the phase on which you want to apply the AGC algorithm.

```
enum Tle9xxx::_Halfbridges
```

Values:

```
enumerator PHASE1
```

```
enumerator PHASE2
```

```
enumerator PHASE3
```

```
enumerator PHASE4
```

All other phases will be always connected to ground. In practice it’s usually sufficient to execute the algorithm on one phase, as the other MOSFETS should meet the same specifications and come from the same charge.

```
uint8_t trise_tg = 11;                // [0;63] Initial Risetime target. Can be
↳changed via keyboard input.
uint8_t tfall_tg = 11;                // [0;63] Initial Falltime target. Can be
↳changed via keyboard input. (continues on next page)
```

Here you can define your initial target Rise- and Falltime, which will be set by

```
void DCMcontrol::setTrisefallTarget(uint8_t trise_tg, uint8_t tfall_tg)
    Set the T_Rise and T_Fall target times where the regulation loop should go to.
```

Parameters

- **trise_tg** – rise time target [0;63]
- **tfall_tg** – fall time target [0;63]

If this function is not used (like in normal motor operation) the values from the defines will be taken as described below. The entered values in the examples are suited as a starting point for the DC and BLDC shield. However if other MOSFETS are used, refer to the [TLE9560/1/2 Gate Driver Setting Guide](#) in order to estimate start values for rise and fall times, turn-on and turn-off delay times and recommendations for the settings of the cross-current protection time and of the blank times.

```
void DCMcontrol::riseFallTimeRegulation(uint8_t hb, uint8_t *iCharge, uint8_t *iDischarge, uint8_t
    *risetime, uint8_t *falltime)
```

reads out the actual MOSFET rise-time (fall-time) and compares it to the desired rise-(fall-)time. The algorithm then adjusts the charge current ICHG for the active MOSFET of the selected halfbridge.

Parameters

- **hb** – on which halfbridge should the algorithm be applied. Must be the same halfbridge where the PWM is routed to.
- **risetime** – hands over the actual rise-time
- **falltime** – hands over the actual fall-time

This function executes the algorithm one time and hands over the variables to read back the actual rise- / falltimes and charge-/discharge currents.

3.7.2 Parameter defines

In order to constantly change the initial charge current (ICHG) / initial discharge current (IDCHG) go to [/src/corelib/TLE9xxx.hpp](#). There you find the defines listed below. Just replace the values there by the values you found out experimentally.

CONF_TRISE_TG

[0;63] initial Target tRISE (CONF_TRISE_TG * 53.3 ns). The variable can be changed afterwards.

CONF_INIT_ICHG

[0;63] Starting charge current that will be first used by the algorithm

CONF_TFALL_TG

[0;63] initial Target tFALL (CONF_TFALL_TG * 53.3 ns). The variable can be changed afterwards.

CONF_INIT_IDCHG

[0;63] Starting discharge current that will be first used by the algorithm

3.8 BLDC Motor Tuning

If you have no or bad startup behavior with your BLDC motor, there are some parameters outside the example sketches that you can change. Therefore navigate to </src/corelib/BLDCM-control.hpp> in your local library.

3.8.1 Safety

CONF_TIMEOUT

Main place to configure BLDC motor parameters. All defines beginning with “CONF_” are intended to be changed by the user. All other defines should remain as they are. milliseconds. How long no commutation may occur until it can be assumed, the motor got stuck

In order to prevent damage to your motor when it's mechanically blocked, there is a timeout feature. After this time (500ms) when no commutation occurred, all three phases will be switched off.

3.8.2 RPM Regulation

CONF_PI_UPDATE_INTERVAL

milliseconds. How often should the PI regulator be called. Affects precision if too low.

3.8.3 Startup

CONF_RPM_DUTYCYCLE_AT_START

duty cycle when motor starts to turn before RPM controller will be switched on

CONF_OPEN_LOOP_DUTYCYCLE

duty cycle for blind commutation at motor start (open loop)

CONF_OPEN_LOOP_DELAY_START

microseconds. This is the delay between the first commutations when starting a BLDC motor

CONF_OPEN_LOOP_DELAY_LIMIT

microseconds. The smallest delay that is used before open loop commutation turns into closed loop

CONF_OPEN_LOOP_DELAY_SLOPE

microseconds. The amount CONF_OPEN_LOOP_DELAY_START will be decreased every open loop commutation. You can calculate the amount: $O_L_commutations = (5000-1200)/200$

3.9 Arduino Getting Started

3.9.1 Arduino Compatible Kits

This library is designed for multiple platforms with Arduino Uno compatible headers and different SDKs. The following hardware platforms are compatible and tested:

Hardware Platform	Type	SDK	File Marker	Checked
Arduino	Uno Rev3	Arduino IDE	*.ino	yes
Infineon XMC	XMC4700 Relax Kit	Arduino IDE	*.ino	yes
Infineon XMC	XMC1100 Boot Kit	Arduino IDE	*.ino	yes

Other MCU platforms which have an Arduino port may not work, as high frequency PWM (30kHz) is required on pins 3, 9 and 10 what is not available on all boards. Second limitation is the SPI bound to pins 11 (MOSI), 12 (MISO) and 13 (SCK) what is only available on the UNO and XMC boards. However if you wire it manually, the shield might work on more MCUs.

XMC for Arduino

3.9.2 Arduino Library Installation

Required Software

1. **Install the Arduino IDE.** If you are new to Arduino, please [download](#) the program and install it first.
2. **Include the XMC boards in the IDE (if a XMC is used).** The official Arduino boards are already available in the Arduino IDE, but other third party boards as the Infineon XMC MCU based ones need to be explicitly included. Follow the instructions in the [link](#) to add the XMC board family to the Arduino IDE.
3. **Install the library.** In the Arduino IDE, go to the menu *Sketch > Include library > Library Manager*. Type **motor system IC TLE956x** and install the library.

Installation Methods

The library can be installed in several ways:

- Arduino IDE library manager
- Arduino IDE import .zip library
- Arduino IDE manual installation

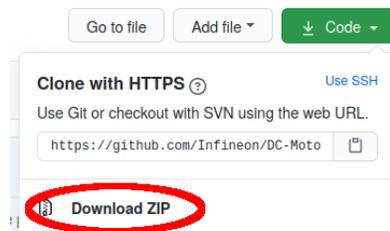
These installation processes are conveniently described on the official [Arduino](#) website.

- **Arduino IDE Library Manager**

Library name: motor system IC TLE956x

- **Arduino IDE Manual Installation**

Download the desired .zip library version from the repository [releases](#) section.



Warning: As a general recommendation, downloading directly from the master branch should be avoided. Even though it should not, it could contain incomplete or faulty code.

3.9.3 Arduino API

Example TLE9562_DCM-control

Here we go through each function and variable used in this sketch and show up other control possibilities.

void setup()

Include the library and create an instance of the class **DCMcontrolIno**:

```
#include <Arduino.h>
#include <DCM-control-ino.hpp>

#define MOTOR_OUTPUT      3           // [1;3]
#define SPEED_INCREASE_STEP 100      // [1;511] speed step increase/decrease
↳when pressing a key

uint16_t speed = 400;
uint8_t direction = 0;

// Create an instance of DCMcontrol called 'MyMotor'.
DCMcontrolIno MyMotor = DCMcontrolIno();
```

The MOTOR_OUTPUT define is used to configure which motor will be controlled according to the table below. Motor 1 has to be wired to PHASE1 and PHASE2, Motor 2 to PHASE3 and PHASE4.

MOTOR_OUTPUT	Motor 1	Motor 2
1	yes	no
2	no	yes
3	yes	yes

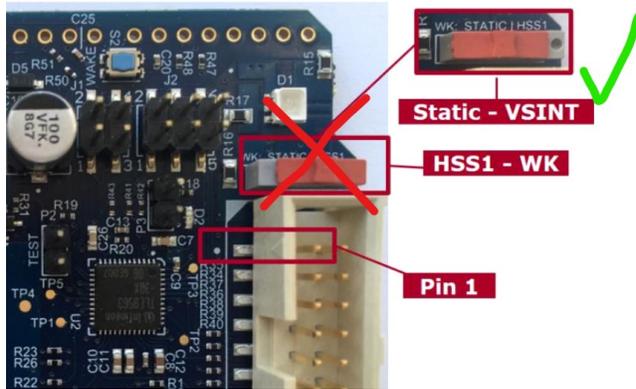
The not-controlled motor will always keep its last state.

Now set up a GPIO interrupt routine bound to Pin 2 (the interrupt Pin of the TLE956x shield). By default, the library configures the TLE to throw an interrupt if an error in one or more status register occurs:

```
// Enable GPIO interrupt for pin 2
attachInterrupt(digitalPinToInterrupt(2), TLEinterrupt, LOW);
```

When you jump to the last lines in the example sketch, you see, the TLEinterrupt function only sets the variable *interrupt_status_changed* to one. Just keep that in mind for now, we come to the reason later on.

Important note: In order to use the interrupt function properly, make sure the HSS switch of the board is in position *Static*. Otherwise the interrupt is bound to the PWM of HSS1 and thus called periodically, if this HSS is used.



Next initialize the pins and configure the interrupt mask (which TLE956x-errors cause an error message on the serial monitor). The default settings will be applied. The TLE9562 DC motor shield also features two red LEDs that can be controlled individually by using two HSS outputs of the TLE9562. Using the `setLED()` function, the brightness of both LEDs can be set using a 10-bit value:

```
MyMotor.begin();  
MyMotor.configDCshield();  
MyMotor.setLED(0,100); // Switch on LED 2
```

At the end of the `setup()` function, the initial set speed and direction for the selected motor(s) will be applied to the shield:

```
MyMotor.setDCspeed(speed, direction, MOTOR_OUTPUT);  
MyMotor.startDCM();
```

void loop()

In order to change speed, direction, motor outputs, start or stop the motor, an if-routine has been implemented, that scans the Serial-input line. Have a look in *Keyboard commands* to see which key to press:

```
if (Serial.available() > 0)  
{  
    uint8_t in = Serial.read();  
    if(in == '+')  
    {  
        speed += SPEED_INCREASE_STEP;  
        Serial.println(speed);  
    }  
    if(in == '-')  
    {  
        speed -= SPEED_INCREASE_STEP;  
        Serial.println(speed);  
    }  
    if(in == 'd')  
    {  
        direction = 0;  
        Serial.println(F("forward"));  
    }  
    if(in == 'e')
```

(continues on next page)

(continued from previous page)

```

    {
    direction = 1;
    Serial.println(F("backward"));
    }
    if(in == 'a')
    {
    MyMotor.stopDCM(BRAKEMODE_PASSIVE);
    Serial.println(F("Motor stopped"));
    }
    if(in == 'q')
    {
    MyMotor.startDCM();
    Serial.println(F("Motor started"));
    }

    MyMotor.setDCspeed(speed, direction, MOTOR_OUTPUT);
}

```

If a key was pressed, the changes will be applied to the board using the *setDCspeed(speed, direction, MOTOR_OUTPUT)* function again.

Example TLE9563_BLDCM-control

Here we go through each function and variable used in this sketch and show up other control possibilities.

void setup()

Include the library and create an instance of the class **BLDCMcontrolIno**:

```

#include <Arduino.h>
#include <BLDCM-control-ino.hpp>

uint16_t speed = 400;
uint8_t direction = 0;
uint8_t weakening = 0;

// Create an instance of BLDCMcontrolIno called 'MyMotor'.
BLDCMcontrolIno MyMotor = BLDCMcontrolIno();

```

Set up a GPIO interrupt routine bound to Pin 2 (the interrupt Pin of the TLE956x shield). By default, the library configures the TLE to throw an interrupt if an error in one or more status register occurs:

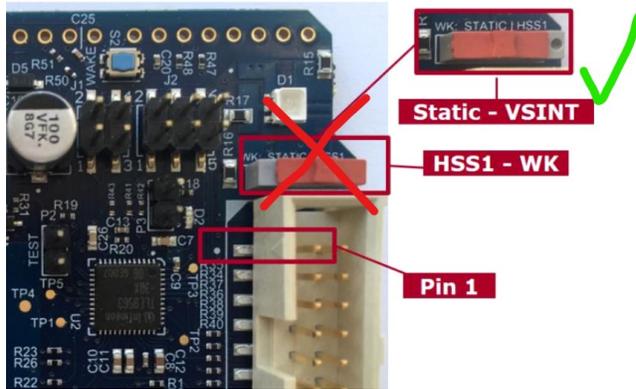
```

// Enable GPIO interrupt for pin 2
attachInterrupt(digitalPinToInterrupt(2), TLEinterrupt, LOW);

```

When you jump to the last lines in the example sketch, you see, the *TLEinterrupt* function only sets the variable *interrupt_status_changed* to one. Just keep that in mind for now, we come to the reason later on.

Important note: In order to use the interrupt function properly, make sure the HSS switch of the board is in position *Static*. Otherwise the interrupt is bound to the PWM of HSS1 and thus called periodically, if this HSS is used (e.g. the green LED is on).



Motor control functions

First, we need to call the `begin()` function, that configures all input/output pins, PWM frequencies and so on. The function `setLED(red, green, blue)` let us set the color of the onboard RGB-LED, driven by the high-side-switches of the TLE. It takes 10-bit values as argument, so you can enter values from 0 (off) to 1024 (max brightness). Make sure your HSS-jumpers are in place.

```
MyMotor.begin();
MyMotor.setLED(0, 20, 0);
```

Now comes the important part: You need to select which position-feedback and which speedmode you want to use:

```
MyMotor.MotorParam.feedbackmode = BLDCMcontrol::BLDC_HALL;
MyMotor.MotorParam.speedmode = BLDCMcontrol::BLDC_BLDC_DUTYCYCLE;
MyMotor.MotorParam.MotorPolepairs = 4;
```

MotorParam	type	arguments	input range
feedbackmode	mandatory	BLDC_HALL BLDC_BEMF	
speedmode	mandatory	BLDC_DUTYCYCLE BLDC_RPM	0 - 1023 0- 2E32
MotorPolepairs	mandatory for BLDC_RPM	integer value	0-255
PI_reg_P	optional (default = 0.01)	float value	
PI_reg_I	optional (default = 0.01)	float value	

If you don't know the amount of pole-pairs in your BLDC, you can use the `find_polepairs_BLDCM.ino` sketch provided in the `examples` folder. If you use a wrong number, your actual RPM-speed might be imprecise.

In order to actually set the previous defined parameters, call the following function:

```
MyMotor.configBLDCshield();
```

Finally, we can set our default speed and direction and start the BLDC motor:

```
MyMotor.setBLDCspeed(speed, direction);
MyMotor.startBLDCM();
```

Depending on your configuration above, the `speed` - parameter will be interpreted as a percentage-value (a permit-value to be precise) or as a desired RPM-speed. `direction` can be 0 or 1. A third argument `weakening_range` would be

possible as well that can be 0 (default) or 1, but is only applicable if BLDC_HALL was selected. *weakening range* uses a different commutation pattern, that let's the motor turn with its double speed but less torque.

startBLDCM() applies an open-loop commutation to your motor and enables the usage of *serveBLDCshield()* which actually commutates the motor.

void loop()

In order to change speed, direction, weakening range (only for BLDC_HALL), start or stop the motor, an if-routine has been implemented, that scans the Serial-input line. Have a look in *Keyboard commands* to see which key to press:

```

if (Serial.available() > 0)
{
    uint8_t in = Serial.read();
    if(in == '+'){
        speed += SPEED_INCREASE_STEP;
        Serial.println(speed);}
    if(in == '-'){
        speed -= SPEED_INCREASE_STEP;
        Serial.println(speed);}
    if(in == 'd'){
        direction = 0;
        Serial.println(F("forward"));}
    if(in == 'e'){
        direction = 1;
        Serial.println(F("backward"));}
    if(in == 's'){
        weakening = 0;
        Serial.println(F("Field weakening disabled"));}
    if(in == 'w'){
        weakening = 1;
        Serial.println(F("Field weakening enabled"));}
    if(in == 'a'){
        MyMotor.stopBLDCM(BRAKEMODE_PASSIVE);
        Serial.println(F("Motor stopped"));}
    if(in == 'q'){
        MyMotor.startBLDCM();
        Serial.println(F("Motor started"));}
    MyMotor.setBLDCspeed(speed, direction, weakening);
}

```

For example, if you press a, the function *stopBLDCM(brakemode)* is called. As the name says, it stops the commutation and prohibits the use of *serveBLDCshield()*, where *brakemode* defines, whether the phases are left floating (*BRAKEMODE_PASSIVE*) or actively tied to ground (*BRAKEMODE_ACTIVE*). The *F()* function which wraps the strings in the serial prints is called the F-macro and helps to save dynamic memory.

Last but not least, you may not forget to call the most important function, where all the magic happens: *serveBLDCshield()*

Depending on the previously defined configuration, this function checks, if the hall-sensor or BEMF-sensor state changed since the last time the function was called and if so, it commutates the output phases. This means, this function needs to be called **as often as possible** and the time between calling this function must be **as short as possible**.

```

MyMotor.serveBLDCshield();           // MUST BE CALLED HERE. This function does the
↳BLDC commutation.
if(MyMotor.checkTLEshield() )       // Check, if interrupt flag was set and read
↳status register of TLE
{
    MyMotor.setLED(50,0,0);         // Set onboard RGB-LED to red.
}
    
```

The function *checkBLDCshield()* is not mandatory to run the BLDC, but handles error codes and prints debug messages. If you remind the interrupt setting at the beginning, I can now tell you, this function will only be executed if *interrupt_status_changed* was set to 1.

3.9.4 Arduino Examples

To run these examples use either the Arduino IDE or something similar like the PlatformIO extension for Visual Code or Atom.

examples/TLE9562_DCM-control	<i>Default example sketch to run DC motors.</i>
examples/TLE9562_DCM_rise-and-fall-time-regulation	<i>Perform the rise- / falltime control loop on a single output phase.</i>
examples/TLE9563_BLDCM-control	<i>Default example sketch to run a 3 Phase BLDC motor with Hallsensor or BEMF.</i>
examples/TLE9563_BLDCM_rise-and-fall-time-regulation	<i>Perform the Rise- / Falltime control loop while controlling a BLDC motor. Might be experimental.</i>
examples/TLE9563_BLDCM_find-polepairs	Simple script to identify the amount of polepairs of a connected BLDC motor.

Keyboard commands

These are common key-operations for the serial monitor used by the various examples stated above. Not all operations are available in every example.

	Up / Enable	Down / Disable
Speed	+	-
Motor enable	q	a
Weakening Range enable (BLDCM)	w	s
Direction	e	d
Risetime target	r	f
Falltime target	t	g
Rise- / Falltime Regulation enable	u	j
AGC enable	i	k

3.9.5 Software

1. Follow the instructions in *Arduino Library Installation* to install the required software and libraries.
2. Choose a *suitable example sketch* and upload the code to your Arduino.

3.9.6 Hardware

1. Connect your shield to a *Microcontroller board* supported by this library.
2. Set some jumpers on the board like explained *here*
3. Connect your motor(s) to the screw terminals.
4. Connect a 12V DC power supply with current limitation.

3.9.7 Ready!

Switch on the power supply and control your motor(s) with the serial monitor in the Arduino IDE. Refer to *Arduino API* and *Keyboard commands* for more details on how to use the example sketches.

B

BLDCMcontrol (C++ class), 13
 BLDCMcontrolIno (C++ class), 13

C

CONF_INIT_ICHG (C macro), 16
 CONF_INIT_IDCHG (C macro), 16
 CONF_OPEN_LOOP_DELAY_LIMIT (C macro), 17
 CONF_OPEN_LOOP_DELAY_SLOPE (C macro), 17
 CONF_OPEN_LOOP_DELAY_START (C macro), 17
 CONF_OPEN_LOOP_DUTYCYCLE (C macro), 17
 CONF_PI_UPDATE_INTERVAL (C macro), 17
 CONF_RPM_DUTYCYCLE_AT_START (C macro), 17
 CONF_TFALL_TG (C macro), 16
 CONF_TIMEOUT (C macro), 17
 CONF_TRISE_TG (C macro), 16

D

DCMcontrol (C++ class), 12
 DCMcontrol::riseFallTimeRegulation (C++ function), 16
 DCMcontrol::setTrisefallTarget (C++ function), 16
 DCMcontrolIno (C++ class), 13

T

Tle9562 (C++ class), 12
 Tle9563 (C++ class), 12
 Tle9xxx (C++ class), 12
 Tle9xxx::_Halfbridges (C++ enum), 15
 Tle9xxx::_Halfbridges::PHASE1 (C++ enumerator), 15
 Tle9xxx::_Halfbridges::PHASE2 (C++ enumerator), 15
 Tle9xxx::_Halfbridges::PHASE3 (C++ enumerator), 15
 Tle9xxx::_Halfbridges::PHASE4 (C++ enumerator), 15