P&O EAGLE
Motor control

Uytterhoeven Roel

August 1, 2017

1 Introduction

Drones require high speed motors to drive their propellers and generate enough lift to get airborne. Brushless-DC (BLDC) motors (see figure 1) are the preferred choice for this task due to their high power-to-weight ratio and compact size. These small motors are capable of delivering more than 200 W peak output power and approximately 70 W continuous power. Despite their name, these motors do not operate on a DC voltage source, but require dedicated three-phase voltage and current control, with feedback based on the actual motor speed and position. The electronic speed controller (ESC) is the block assigned with this control and the conversion from the DC battery voltage to the three-phase AC voltage.

Figure 1: Typical BLDC motor for a drone.

1.1 Brushless-DC motor structure

As any motor the BLDC motor consists of a stator and a rotor as shown in figure 2. The stator contains a number of stator poles around which coils are wound creating the stator windings. Feeding the stator windings with current magnetizes the stator poles leading to interaction between these poles and the permanent magnet poles of the rotor. If the current through the stator windings is controlled appropriately, the combined effect of the stator poles results in a rotating magnetic field. Since the rotor is fitted with permanent magnets it will 'try' to rotate at the same speed as this rotating magnetic field and hence the motor turns.

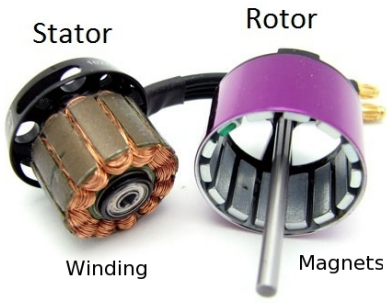

Figure 2: The stator windings and the rotor magnets are clearly distinguishable in this unmounted state.

The stator windings/coils of the brushless motors used in this project are connected in a three-phase star configuration as shown in figure 3. The star point in the middle is the common or neutral node. A simplified schematic of the connections of the coils inside the stator is depicted as well. The arrows indicate the direction of the current flow from one phase to the other. Note that the total amount of stator poles is always a multiple of three (since there are three phases) and that one leg of the star network can contain multiple stator windings (e.g. 9 poles stator means that each leg consists of 3 coils in series).

1.2 Brushless-DC motor current waveforms

The stator windings are supplied with sequential square-wave currents following one of the six arrows (see fig. 3). The order of this sequence has to be correct in order to create a rotating magnetic field that spins the rotor. Figure 4 shows this correct order. Note that the only difference between the three phases (A,B and C) is 120 degrees phase shift, i.e. a delay of 1/3 of a rotation.

The magnitude of these square-wave currents dictates the strength of the rotating magnetic field and thus the torque (moment of force) that the motor generates ($T \propto I$). Depending on the load torque this will either lead to a

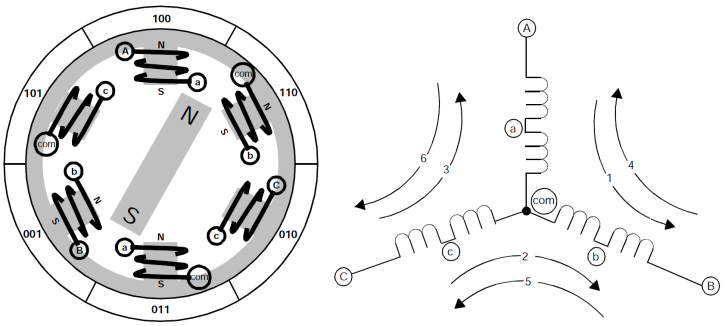

Figure 3: Simplified schematic representation of the stator windings (left) and electrical equivalent circuit (right). The arrows indicate the direction of the current flow during the different stages of the three phase signal.

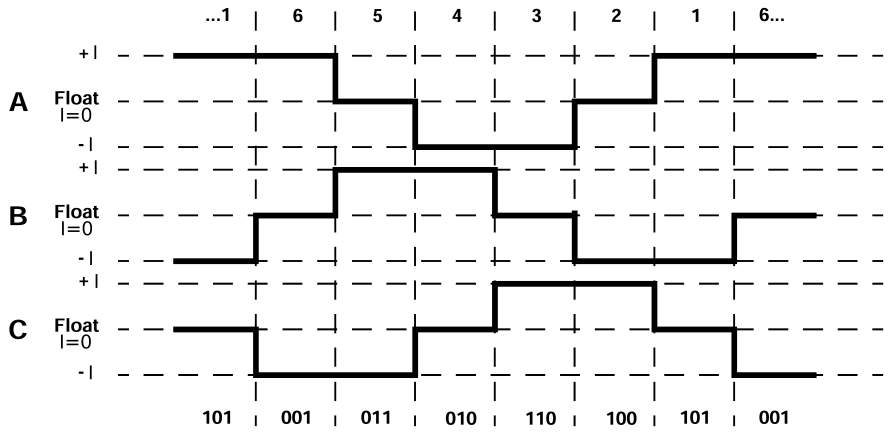

Figure 4: Three phase square-wave currents to drive the BLDC motor. The three phases are A, B and C.

decrease, constant or increase in speed. The frequency and phase of the three-phase square-wave signal have to be synchronized with the motor's speed and position. This to ensure that the rotor magnets are always attracted to the stator pole in front of them and not behind them (otherwise the motor would just vibrate). Achieving this synchronization requires feedback from the motor speed and position.

1.3 Brushless-DC motor back electromotive force.

When spinning, the BLDC motor generates (as any motor) an electric field that tries to oppose the applied motor voltages (these voltages generate the current

pulses). This is more generally called the back electromotive force of the motor or BEMF. The BEMF is directly proportional to the motor speed and therefore the magnitude of the applied voltage has to increase as the motor accelerates to keep generating current pulses of equal magnitude. For a BLDC motor the KV number indicates how many rounds per minute ($\times 1000$) can be expected from every applied volt ($KV = \frac{Krpm}{V_{applied}}$), hence it predicts the magnitude of the BEMF.

Since the BEMF finds its origin in the movement of the rotor magnets in front of the stator coils, it contains information about the speed and position of the rotor of the motor. Therefore it will be used to provided the necessary feedback to operate the motor correctly. Fig. 5 shows the BEMF, note that this signal is also a three-phase signal.

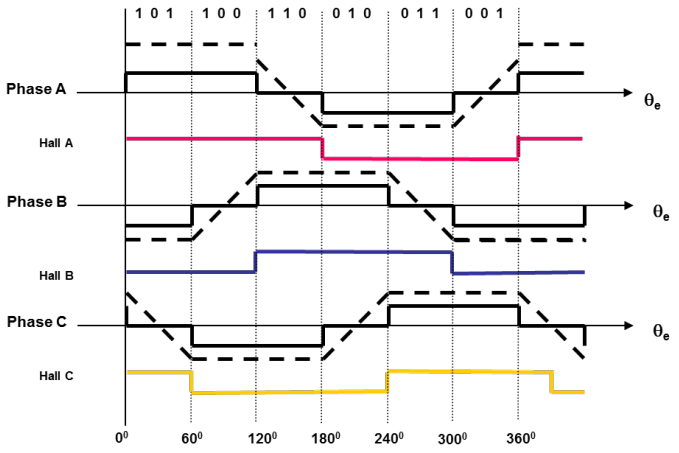

Figure 5: Ideal BEMF waveform (dashed line), current pulses (black line), hall sensor output (colored line) for the three phases during one electrical cycle (period of the 3-phase signal) of the motor.

The point of interest for the feedback is the cross point between the BEMF of each phase and the neutral voltage of the motor (star point of the stator windings). This point should reside in the middle between changes in current (commutations[1]) and therefore indicates how fast the rotating magnetic field should turn. More precisely, it allows to predict when the next commutation must take place (so that the cross point is in the middle) based on the time between the previous commutation and the cross point itself.

As observed in fig. 5 the transition of the BEMF from high to low or low to high always takes place when there is no current flowing through a phase. Therefore the phase where the BEMF can be measured is also often called

[1]A commutation is the transition from current in one phase/direction to another phase/direction and can be seen as switching current on or off in one of the stator windings.

the floating phase. In the controller this phase can easily be recognized and measured with an analog comparator or ADC.

1.4 DC/AC converter to generate the current pulses

Since the BLDC motors require a three-phase square-wave input signal as shown in fig. 4, the DC-battery can not be connected directly to these motors. A converter is necessary in between to generate the three-phase current waveforms. Such a converter is an DC/AC converter also called an inverter. Figure 6 shows the circuit of this converter, the outputs are the three phases A, B and C, the inputs are the DC-battery supply and the six control signals for the power MOSFETs (NMOS transistors) that switch the current of each phase.

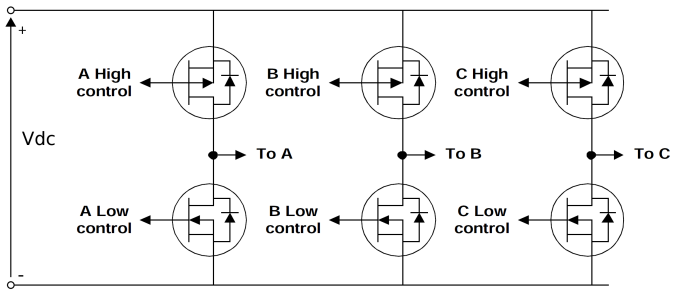

Figure 6: DC/AC converter built from six n-type power MOSFETS with built-in reverse current diodes to allow reverse current flow. A, B and C form the three phase output signal towards the motor.

A microprocessor has to control the switches (MOSFETs) of the DC/AC converter so that the output of the converter is the correct three-phase signal to drive the motor. However, the MOSFETs' gates have a significant capacitance and hence the microprocessor itself cannot directly drive these switches. An additional driver is inserted in between as a buffer with low input capacitance and high output drive. This driver also helps to boost[2] the high side control signal above the battery supply voltage in order to switch on the high side NMOS transistors.

[2]NMOS gate voltage has to go above the source voltage to turn on the transistor. As the source voltage can equal the battery voltage boosting is required.

2 Task

The goal is to design and construct the electronic speed controller (ESC) for the brushless-DC (BLDC) motors of the drone. This block is of vital importance to the rest of the system since without ESCs flying (and also crashing) is impossible! Failures during operation will almost certainly result in a crash, potentially damaging other parts of the drone. Hence reliability and thorough testing are of the utmost importance when constructing the ESCs.

First an open loop implementation of the motor drive should be made without the BEMF feedback. The rotor is assumed to remain synchronized to the rotating magnetic field at low speed with little dynamic behavior (i.e. very slow acceleration/deceleration and no load changes). This mode of operation allows to verify the correct operation of the hardware components and switching scheme of the DC/AC converter. In other words, does the motor get the right current pulse at the right time? [3]

Once the motor is able to operate in open loop, sensing the crossing of the BEMF and neutral voltage provides the required feedback information. This information is to be used to create a closed loop control system that allows to operate the motor dynamically (i.e. accelerate and decelerate rapidly and handle load changes). Operation in closed loop also requires the detection of synchronization loss to prevent damage to the motor/controller in such an event.

Optionally extra circuitry can be added to implement a current measurement. Monitoring the current output from the DC/AC converter allows to intervene sooner when the motor controller malfunctions or loses synchronization. This can be a valuable asset to the design as by the time synchronization loss at high speed is detected some of the circuits components might already be damaged. Further other features can be added based on the current measurement. Note that this is an optional task that should only be attempted when closed loop operation is throughly understood as it is more important that the ESCs work (at least once).

Finally the last step in the design is to provide communication between the four ESCs controlling the motors and the flight controller of the drone. Each ESC should expect a throttle signal from the flight controller. To provide easy swapping between commercial ESCs and your own designed ESCs, the throttle input signal should be the same, a pulse-width modulated (PWM) signal where the duty cycle contains the throttle information.

2.1 Open loop design

The first step in operating the BLDC motors is the generation of the three-phase square-wave current pulses controllable in both magnitude and frequency. Therefore a DC/AC converter has to be controlled correctly by a microprocessor.

[3] Keep in mind at this stage that the timing of the commutations in the closed loop system will depend on the BEMF measurements. Hence it is advisable to ensure from the beginning that this timing variable/setting can be updated at runtime as to avoid having to rewrite software.

This converter is shown in figure 6. Note that the converter actually supplies a PWM voltage switching between $0V$ and V_{bat} at 'high' frequency $f \geq 32kH$. The output current magnitude directly links to the duty cycle of this PWM voltage by a generalized Ohm's law (the stator coils form an impedance $Z_s = j\omega f$ for the fast switching PWM input voltage and hence act as an 'averaging' resistor for this signal).

Once the three-phase signal is generated, the next step is to supply it to the stator windings (see figs. 3 and 4) of the brushless DC motor. The stator windings generate a rotating magnetic field proportional to the current supplied by the DC/AC inverter. As stated above, this current is controlled through the duty cycle of the applied voltage pulses. The permanent magnets of the rotor lock to this rotating magnetic field and hence the motor spins.

The speed of the rotating magnetic field should start from zero and then increase very slowly if the rotor has to remain synchronized without feedback. This means that the frequency of the three-phase signal (and thus also the switching frequency from the DC/AC converter) should start from DC and increase very gently. At the end of this task the motor should be able to start up and spin slowly ($rpm < 500$) at a steady pace without feedback.

2.2 Closed loop design

Once the open loop design works, feedback should be added to the system to adapt the rotating magnetic field to sudden changes in motor speed and load torque and to maintain synchronization under any condition. More specifically the frequency and phase of the rotating field should follow exactly the speed and position of the rotor. This will also allow the motor to speed up naturally when the duty cycle of the applied voltage pulses increases. Note that this implies that the **motor speed is not dictated by the speed of the rotating magnetic field** in closed loop, but rather the other way around, the rotating field must always follow the motor speed. The motor speed itself is determined by the equilibrium between generated torque[4] and load torque.

Two techniques exist to determine the motor speed and position for this feedback, sensored control and sensorless control (see fig. 7).

- Sensored control: hall-sensors (magnetic sensors) sense the passing by of the rotors magnets, which allows them to determine speed and position of the rotor.
- Sensorless control: the BEMF voltage in the floating step/phase of the DC/AC converter is compared to the neutral motor voltage. As discussed in 1.3 the moment the BEMF crosses this neutral voltage can be used as indication for the rotor speed and position.

Since the BLDC motors for drones typically come without hall-sensors this sensorless technique is required to operate the ESCs. Figure 7 gives a complete

[4]The generated torque is linked to the duty cycle of the DC/AC converter. This duty cycle forms the throttle of the motor.

overview of the sensorless control system.

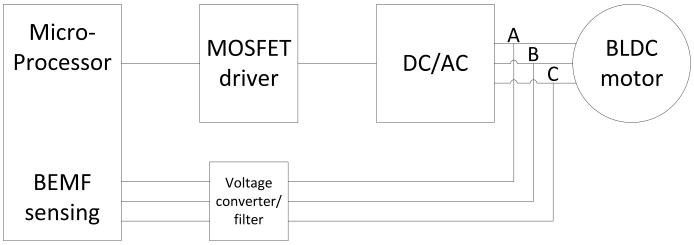

Figure 7: Simplified system overview of the sensorless control based on sensing the BEMF during the floating phase.

Be aware that loosing synchronization at high speed can seriously damage the motors by creating short circuit currents ($I > 100A$). These currents arise when at high duty cycle the voltage pulses are suddenly no longer opposed by the BEMF of the motor. In normal operation the effective voltage over the stator windings $V_{winding_i}$ is limited by the BEMF since it opposes the applied voltages as shown in 1 and hence $V_{winding_i} < 0.1\,\mathrm{V}$. The impedance of the winding $Z_{winding_i}$ thus suffices to limit the current (2). However, when synchronization is lost, the BEMF does no longer oppose the applied voltage pulses and the effective voltage over the windings equals the applied voltage pulses creating extremely large currents.

$$V_{winding_i} = V_{DC} \cdot D - V_{BEMF_i} \tag{1}$$

$$I_{winding_i} = \frac{V_{winding_i}}{R_{winding_i}} \tag{2}$$

To prevent motor damage, safety measures must be implemented so that any synchronization loss is detected as soon as possible. All connections to the motor must then immediately be opened (turn off all the DC/AC switches) so that all currents are interrupted.

2.3 Current measurement (optional)

This is an optional part of the task and should only be attempted after a discussion with the task expert.

The main advantage of monitoring the current is rapid detection of over currents and thus better protection of the motor (controller) in case of a malfunction or synchronization loss. Also other features could be extracted from the current measurement (e.g. Power consumption information, quick start up,...). The design and implementation of this measurement comes without further guidelines and is thus open to all possible ideas. However, it might be wise to consult the expert to check feasibility.

2.4 Communication with the flight controller

The final task is implementing the communication between the flight controller and the ESCs. The flight controller typically outputs a PWM signal containing the throttle information. The developed ESCs thus require a decoder for this signal. Once the throttle signal is decoded it can simply be used to set the duty cycle of the DC/AC converter as this acts as the throttle of the motor.

This decoder must not interfere with the control of the motor itself since this can lead to highly unpredictable errors that do only show up sporadicly (Murphy's law dictates that this happens in flight). Yet the update rate of the throttle should be fast enough to avoid instabilities in the flight control loop. Therefore it is advised to discuss with the flight controller team what the required update rate is.

2.5 Provided building blocks

- Schematic of the DC/AC inverter
- PCB layout of the DC/AC inverter
- 6 Power MOSFETs
- 3 MOSFET driver ICs
- Arduino nano/Nucleo PCB footprint
- 4 Arduino nanos with ATMega processor or 4 Nucleos with ARM Cortex M0 processor
- Test equipment

2.6 Different sub-tasks

First semester

1. Material reading	2 students / 1 week
2. Open loop design	2 students / 2 weeks
3. Closed loop design (+ current measurement)	2 students / 4 weeks
4. PCB design	2 students / 3 weeks

Second semester

1. PCB assembling	2 students / 2 weeks
2. PCB testing	2 students / 3 weeks
3. Communication with flight controller	2 student / 2 weeks
4. Solving reliability issues (integration on the drone)	2 student / 3 weeks

2.7 Milestones

- "Understand and plan" phase: T1
  1. Understand working principles of BLDC motors and DC/AC converters.
  2. Study available ESCs designs.
  3. Design your own ESCs.
- Modeling phase: T2
  1. Breadboard design: open loop
  2. Breadboard design: closed loop
  3. Current Measurement design (optional)
  4. PCB design (must be finished before Christmas, production takes place during exams/holidays)
- Module phase: T3
  1. PCB assembling
  2. PCB testing and debugging
  3. PWM input decoder
  4. One working ESC on PCB should be finished
- Integration phase: T4
  1. Integration with flight controller $\rightarrow$ 4 working ESCs on PCB
  2. Improve reliability

3 Sub-tasks: detail

- *Material reading:* Make a general block diagram of the ESC's hardware components that shows how these blocks interact and which kind of signals will be used for this interaction (e.g. 1 bit digital, analog,...). Also add a schematic software structure required to operate the hardware with indications on how the closed loop operation will work and which hardware timers and interrupts from the processor will be used. It is advisable to discuss your diagrams with the expert early on to ensure their feasibility.
- *Open loop design:* Implement software that enables correct switching of the DC/AC-converter and makes the motor spin. Gather the required hardware components and test the hardware configuration on breadboard with a current limited ($< 1\,\mathrm{A}$) source. The rotor of the motor should be able to rotate at a fixed speed, but may stall if the load torque or throttle is changed (rapidly).

- *Closed loop design:* Add the feedback loop to the open loop system. The actual motor speed has to be measured through the BEMF zero crossing visible on the floating phase of the DC/AC-converter. The controller should adapt and synchronize the commutation frequency (switching speed of the DC/AC converter) with this measurement. Once implemented, the motor should be able to operate dynamically (adapt speed to both throttle and load torque changes) without stalling. Additionally safety measures should be taken to avoid motor burnout during an involuntary loss of synchronization after removing the current limit (only remove this limit once the safety measures have been tested!). At this point the software on the microcontroller is more or less complete and no major pin changes should follow in the future, hence the PCB design can start. **Without a working closed loop design the PCB design can not start!**

- *Current measurement design:* (optional) Design a method to measure the current flow through the DC/AC converter. If possible already test this on breadboard. If this design is to be added to the overall PCB design consult with the expert first!

- *PCB design:* As soon as the pin use of the microcontroller is fixed & proven to work on breadboard, the design of the PCB can start. The DC/AC inverter and the required drivers are already provided in the form of an Altium schematic and layout. The rest of the design has to be added to this schematic and layout. Changes to the provided layout are allowed but of course at your own risk.

- *PCB assembling:* It is advised to start assembling one PCB first and only start on the others once this one 'works' to avoid wasting time in the case of a fault in the PCB design. If the PCB design is proven to work, the three other ESCs can be assembled and tested.

- *PCB testing:* Once assembled the functionality of the PCBs can be tested. It is advised to take this testing step by step. Start by checking for any short circuits even before turning on the supply (use a multimeter to do this). Next the functionality of each block can gradually be tested by checking the output signals (starting from the processor and gradually moving to the motors). Only when it is certain that all signals and components behave as expected, the motor should be added to the testing. Again take it step by step gradually increasing the speed and current limit at which the motor is tested. This way it might be possible to spot problems before they blow up components...

- *PCB redesign:* This step is only required if the PCB design appears to contain one or more irreparable faults. If faults are detected do not stop testing after the first one, but try to locate the problem, to find out which parts work and which do not. With this information modifications to the design can be made and a new PCB can be processed if time allows. Since

reprocessing PCBs is expensive and time consuming, it is advised to check the initial design thoroughly, the additional effort will save you time!

- *Communication with drone:* Communication should first be implemented using a PWM encoded signal as in commercial ESCs to allow interchanging them if the own designed ESCs do not work. Once this communication scheme works a more accurate communication scheme may be developed in collaboration with the flight controller team as an additional challenge. However, do remember that the communication should never interfere with the control of the motor itself since this will lead to unpredictable and random errors, hard to debug and that might not show up until longer test flights are performed!

- *Solving reliability issues:* Once everything is proven to work, the design can be finalized by solving remaining reliability issues that are inconvenient but hopefully not critical to the working of the ESC.

4 Building blocks explanation

- *Schematic of the DC/AC inverter:* The schematic of the DC/AC inverter including the drivers is provided since it will be the same in every design.

- *PCB layout of the DC/AC inverter:* This is the PCB layout to start from. It also defines the broad shape.

- *Power MOSFETs:* The six power MOSFETs (NMOS type transistors) are the core of the DC/AC-inverter and act as transistors (switches) and diodes at the same time as can be seen in fig. 8. Although the MOSFETs are rated for more than 100 A (giving more than enough margin for ESCs that should consume around 15 A at max speed) care is still to be taken to circumvent high power consumption in these components. As a general rule it should be avoided that both current and voltage are present in a component at the same time since this leads to power consumption ($P = I \cdot V$). This means that turn-on and turn-off of the MOSFETs should go as quickly as possible, hence the special drivers. Additionally it is preferable to not operate the MOSFETs diodes with a forward current since the forward diode voltage (0.7 V) in combination with the current causes a significant power dissipation in the MOSFETs. This does require the generation of a non-overlapping inverted or complementary PWM signal.

- *MOSFET driver ICs:* For each pair of MOSFETs (high + low side in a leg of the DC/AC) there is one driver IC. This driver has as main task to improve the slew rate (speed at which the gate capacitance is charged) when switching the MOSFETs ensuring fast turn-on/off times. Additionally they also help to boost the high side gate voltage above the normal supply voltage to be able to turn on the NMOS transistors on the

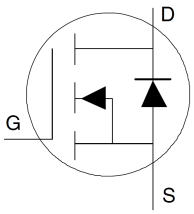

Figure 8: Power MOSFET symbol.

high side (V_{gs} has to be larger than V_t for the transistor to be on, but this is not trivial when $V_s = V_{bat}$.

- *Arduino nano/Nucleo PCB footprint:* The Arduino nano and the Nucleo share the same footprint. This footprint has to be included in each PCB design to connect the chosen processor.

- *4 Arduino nanos or 4 Nucleos:* For the microcontrollers there is a choice between two options, an Arduino nano or a Nucleo. Both processor are able to do the job, yet this choice should not be made lightly. In general the Arduino nano is easier to use and comes with more online support, it however does not allow for much flexibility in the software design as it provides only just enough hardware resources. The Nucleo is harder to use and comes with a lot less online support and a complexer datasheet. However, it allows for more flexibility in the software design as it comes with more hardware resources that also have (a lot) more features. As a general rule of thumb: only choose the Nucleo if you are a confident programmer and have some experience with embedded processors and (most of all) their datasheet. Appendix A provides more information on both processors.

5 Getting started: tips and tricks

This section contains various tips and tricks that will be helpful throughout the project.

5.1 Stepping through different motor states at a fixed speed

The basic operation of the BLDC motor control is stepping through states that apply the correct motor currents as shown in fig. 4. There are six states each denoted with a code number. The order of execution of these state always remains fixed, only the stepping speed varies. This speed determines the frequency of the three-phase generated signal and should be controlled by the BEMF measurement in closed loop operation.

Tip: a timer in combination with one or more interrupts allows to execute a specific block of code at fixed intervals. More information on timers and interrupts can be found in the appendix or in the datasheet of the processor.

Tip: a switch-case statement allows to determine several software states that are always executed in a fixed order.

5.2 Operating the DC/AC efficiently

The DC/AC inverter has to provide all the currents that feed the motor and since these currents can easily exceed 10 amps the efficiency of the DC/AC converter is important. The largest issue with a Dc/AC that operates inefficient is heat, as losses in electronics usual mean that things get (too) hot. To avoid melting or exploding transistors care should be taken to control this converter as efficient as possible.

Ensuring efficiency in a blocks that conducts large amount of power generally boils down to one simple rule: do not allow current and voltage at the same time since $P = VI$. This means that a switch should either be fully on (current through the switch but no voltage over it) or fully off (voltage over the switch but no current through it). If the transistors are controlled properly[5] this requirement should be satisfied.

However, current can and will also flow through the reverse diodes of the transistors. This is caused by the inductance[6] of the stator windings. These inductors create a freewheeling current during the off-part from the throttle PWM, see figure 9. As a result there is both voltage (forward diode voltage) and current present in the transistors causing significant losses. This should be averted!

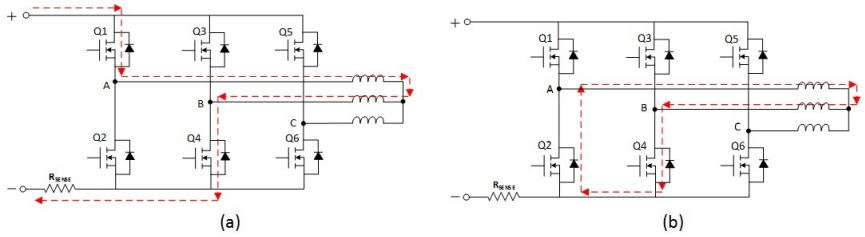

Figure 9: Current flow during one PWM cycle of the DC/AC converter. (a) ON-part PWM cycle, (b) OFF-part PWM cycle.

Tip: a complementary PWM signal can be generated to reduce the inherent losses due to the freewheeling current. To that end this signal should be used to turn on the transistor while the freewheeling current flows and hence ensure that this current flows through the transistor itself and no longer through the reverse diode.

[5] Gate controlled by a driver that guarantees fast turn-on and turn-off times.
[6] The current through an inductor can not stop immediately.

5.3 Measuring and comparing the actual BEMF signal

In the explanation of the sensorless feedback two assumptions where implicitly made. First it was assumed that the BEMF signal looks perfectly as in figure 5. Secondly the availability of the neutral voltage of the motor was assumed.

However as fig. 10 clearly shows the real BEMF looks quite different. The PWM modulation of the DC/AC converter controlling the current is visible on the floating phase and distorts the ideal waveform. Besides, in the beginning of the rising floating phase there is a spike on the BEMF voltage, this is caused by the demagnetization of the stator coils after a commutation event.

Furthermore, the neutral voltage of the motor is only available inside the motor itself and hence the reference voltage for the comparison is missing.

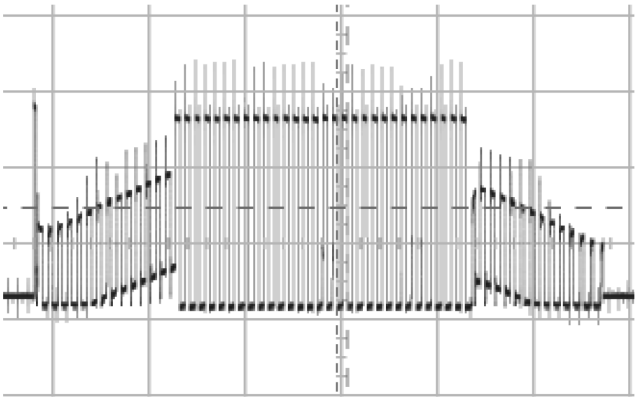

Figure 10: Actual measured phase voltage during one cycle. PWM current/throttle control is still clearly visible on the BEMF signal during rising and falling making the detection difficult.

Two different solutions exist for these problems:

1. Filter out the PWM signal superposed on the BEMF using analog low pass filters. This ensures that we get a BEMF signal such as the one in fig. 5. Create a virtual external neutral voltage through the use of a resistive star-network. This technique makes the processing afterwards very easy since the PWM signal is completely removed and can be compared to the virtual neutral, but causes problems at high speed as the low pass filter also delays the BEMF signal itself.

2. Look at the BEMF during the on-phase of the PWM (top of the pulses) and compare with $\frac{V_{bat}}{2}$ instead of $V_{neutral}$ (when PWM is high $\frac{V_{bat}}{2} = V_{neutral}$. This avoids the delay problems the filter causes, but it is harder to implement since the BEMF should only be sampled when the PWM is high.

Although the latter solution is harder to implement in software, it is still the preferred solution. Using analog filters causes so much delay on the BEMF signal at high speeds that the time measurement resulting from it is no longer representative for the actual motor behavior. This results in inefficient operation causing large currents in the stator of the motor.

Tip: the rising side of the BEMF does not actually require special sampling since the first time it crosses the reference voltage $\frac{V_{bat}}{2}$ *is at the 'correct' BEMF crossing. Yet on the falling side the pulses of the PWM cause the BEMF signal to cross the reference voltage several times. This while the point of interest is but the last crossing, thus special sampling is required. It might however be possible to operate the motor on only the rising side...*

Tip: by playing with the sampling frequency and phase a digital notch filter can be realized without the introduction of any delay on the BEMF signal. Whether this is possible in your particular case depends on the processor and sampling method used.

5.4 Decoding the PWM input throttle information

The obvious method to decode a PWM signal is through the use of a timer and pin change interrupt. However the control of the motor itself is also (probably) based on interrupts. Using interrupts for anything else as the control of the motor will cause interference with the timing critical control interrupts. This can lead to errors that are not deterministic and hard to debug. Hence avoid using interrupts that are not related to any of the motor control timers!!

Tip: the main loop has a large amount of 'spare time'. This allows the use of a technique called polling to detect pin changes. Both simple and complex processors are capable of polling a signal.

Tip: complexer processors often offer more timers with more options that might allow direct decoding in hardware of an input PWM signal without the use of interrupts. Check the datasheet of your processor to see if this is possible.

5.5 Debugging

The both the arduino and nucleo support serial communication over the usb-interface towards a PC. This is very interesting for debugging as it allows to print out the values of variables during execution. However, using *Serial.print (Arduino)* or *printf (Nucleo)* can block the execution of the program for several *ms*. This interferes with timing critical code and can thus not always be used.[7]

Tip: use Serial.write and Serial.read for Arduino and USARTSendData for Nucleo, these methods are faster and do not block the execution of the code. They

[7] This is not only true for serial print statements but for any blocking function (function that halts all execution until completion) and in general care should be taken to avoid blocking code segments in timing critical applications. If it is not possible to avoid the blocking statement you should guarantee that under no circumstances it will cause issues.

simply send or receive one byte without further processing. Remember that the serial transmission still takes time and thus is 'bad' for debugging timing issues.

Tip: using output port toggling allows to accurately measure timing between events in the code using the oscilloscope. Switching the logical value of an output port takes but one clock cycle if done correctly. See the documentation on direct port manipulation for more information.

The Nucleo comes with a build in debugger that can be activated through the IDE. This debugger allows to halt the code at user specified breakpoints. At any breakpoint the full state of the processor is visible to the users. This means that all the 'current' values of variables, registers, output ports,... are available for verification. This is an extremely powerful feature. The only drawback is that the execution of the code is halted at each breakpoint hence stopping all real time (motor) control. The Arduino does not have this feature.

Tip: a breakpoint can be added by double clicking the line number in the STM32 workbench IDE. Next run the program in debug mode (button left of the normal run button).

6 Documentation

6.1 Brushless motors

- BLDC fundamentals:
  http://ww1.microchip.com/downloads/en/AppNotes/00885a.pdf
- Different methods for sensorless control:
  http://www.st.com/content/ccc/resource/technical/document/application_note/aa/b4/69/3f/75/58/4a/a1/CD00020086.pdf/files/CD00020086.pdf/jcr:content/translations/en.CD00020086.pdf
- Video introduction to BLDC sensorless control:
  https://www.youtube.com/watch?v=NZsbE47Qcms
- Animation of voltages during one revolution:
  https://www.youtube.com/watch?v=oFI7VW6WGR4

6.2 Arduino

- Arduino reference site:
  https://www.arduino.cc/en/Reference/HomePage
- Arduino direct port manipulation:
  https://www.arduino.cc/en/Reference/PortManipulation
- ATMega processor datasheet:
  http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88
  datasheet_Complete.pdf

6.3 Nucleo

- Nucleo reference site:
  http://www.st.com/en/evaluation-tools/nucleo-f031k6.html
- ARM Cortex-M0 reference site:
  http://www.st.com/en/microcontrollers/stm32f031k6.html
- ARM Cortex-M0 datasheet:
  http://www.st.com/content/ccc/resource/technical/document/reference_manual/c2/f8/8a/f2/18/e6/43/96/DM00031936.pdf/files/DM00031936.pdf/jcr:content/translations/en.DM00031936.pdf
- Nucleo timer cookbook:
  http://www.st.com/content/ccc/resource/technical/document/application_note/group0/91/01/84/3f/7c/67/41/3f/DM00236305/files/DM00236305.pdf/jcr:content/translations/en.DM00236305.pdf

6.4 PCB design

- Altium tutorial:
  https://techdocs.altium.com/display/ADOH/Tutorial+-+Getting+Started+with+PCB+Design
- Very extensive PCB design manual:
  http://server.ibfriedrich.com/wiki/ibfwikien/images/d/da/PCB_Layout_Tutorial_e.pdf

6.5 Components

- MOSFET IRLB8743PBF datasheet:
  http://www.infineon.com/dgdl/irlb8743pbf.pdf?fileId=5546d462533600a4015356605d6b2593
- MOSFET driver datasheet:
  http://www.infineon.com/dgdl/ir2110.pdf?fileId=5546d462533600a4015355c80333167e

6.6 General

- For all other information see:
  www.google.com
- Prayer
- Task expert

A Arduino vs Nucleo

A.1 Arduino

Arduino boards are specifically designed to be easy to use and allow total newbies but also more experienced users to do quick prototyping. The dedicated Arduino integrated development environment (IDE) takes care of all packages and libraries required to use the desired board and compiles (most of the time) without issue all software code. At the same time it also does the tricky programming of the board itself without most users even noticing it. Thanks to its success Arduino comes with a large community and by consequence also with lots of (un)official support. Nearly for any issue or question a fairly dedicated answer can be found somewhere on the web.

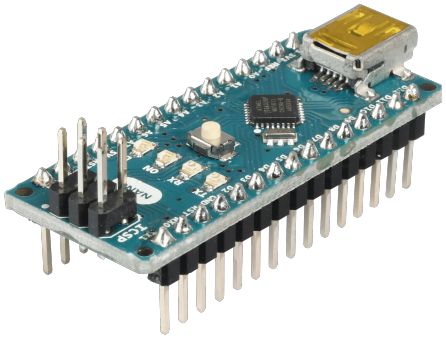

Figure 11: Arduino Nano board.

The Arduino board available for this project is the Nano board. The Nano board is a downsized version of the Arduino Uno board that uses exactly the same processor as the Uno board, the AtMega328P. Obviously this means that any information found for the Uno board is also valid for the Nano board.

The AtMega328P processor is a 8-bit RISC processor operating at a clock frequency of 16MHz. The datasheet of this processor is relatively easy to understand and gives an elaborate explanation of all the hardware components of the processor. The hardware components carried by the processor perfectly fit the requirements of the project.

The matching hardware, the (easily) readable datasheet[8] and the large online support base make this processor the most 'compliant' processor to carry out the project. However, since only just enough hardware is available, using this processor already implies some design choices and limits the freedom in the design. Note that this is not necessarily bad as this significantly reduces the complexity of the project.

The complete overview of the Arduino Nano and its processor can be found in the documentation 6.

[8]The datasheet is almost written as "the AtMega328P for dummies."

A.2 Nucleo

The Nucleo boards are developed to allow quick prototyping of semi-professional systems before starting mass production. The Nucleo boards typically implement an ARM Cortex Mx processor. To enable the use of hardware modules originally designed for Arduino boards they incorporate a similar footprint. The NUCLEO-F031K6 board available in this project has an identical footprint to the Arduino Nano board as can been seen in figure 12.

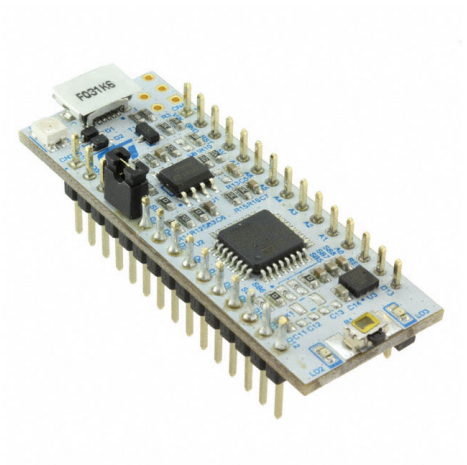

Figure 12: Nucleo F031K6.

The NUCLEO-F031K6 carries the stm32f031k6 ARM Cortex M0 processor. This is a 32-bit RISC processor that runs at a 48MHz clock. Compared to the AtMega328P it offers more precision and more performance. However, the largest difference between the two processors is the peripheral hardware they are equipped with. Contrary to the AtMega that carries just enough hardware for the job, the stm32f031k6 comes with a broader range of hardware timers, GPIO devices, analog devices and programmable interrupts.

The complexer processor on the Nucleo board allows for more freedom in the design, but requires an even better comprehension of the processor's datasheet. Unlike the AtMega's datasheet, the datasheet for the M0 processor is not for dummies. Understanding the datasheet already requires basic knowledge on RISC processor architecture, register operation, timer operation, GPIO operation, c-code,... It should thus be clear that using the Nucleo board will increase the complexity of the project while removing some of the limitations imposed by the more basic Arduino board.

The full details on the Nucleo board and ARM Cortex M0 processor can be found in the documentation 6.

B Interrupts

B.1 What are interrupts

Interrupts are external or internal events that can happen at any time (i.e. asynchronous events). The processor needs to handle these events (immediately) besides the execution of the main loop. There are two main groups of interrupts, external and internal interrupts. The former are interrupts originating from outside the main code, e.g. pin change or timer interrupts. The latter are interrupts originating from special conditions inside the main code of the program (these are less important for this project).

When an interrupts occurs several steps are executed by the processor, figure 13 shows these steps. First the current state of the main program is saved on the memory stack. Next the processor jumps to the specific location of the occurred interrupt in the interrupt vector table (IVT). From this interrupt vector table the processor gets the location of the interrupt service routine (ISR) and jumps to this routine. The ISR handles the event linked to this specific interrupt and reset the interrupt flag once it finishes (the interrupt can now occur again). Finally the context of the main loop is restored and normal execution of the program can resume.

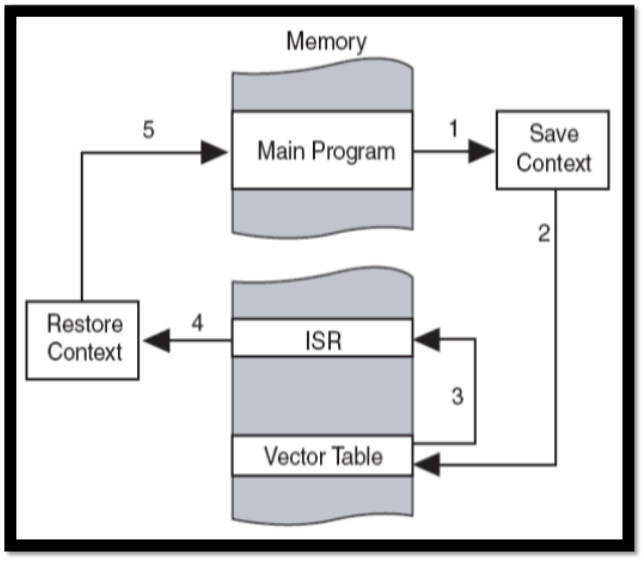

Figure 13: Steps to handle an interrupt.

B.2 How to use interrupts

A processor typically contains several different interrupts sources that each link to their own ISR. All these possible interrupt sources are all listed in the inter-

rupt vector table (found in the datasheet). To use a specific interrupt several steps have to be taken.

1. *Turn on the global interrupt enable flag.* By default all interrupts are disabled using the global interrupt enable flag. On the arduino processor *sei* sets the global interrupt enable flag, *cli* clears the global interrupt enable flag.
2. *Set the specific interrupt enable flag/mask.* Each individual interrupt can be turned on and off using its interrupt enable control bit, this is sometimes also called the interrupt mask since it allows to hide some interrupts while others still come through. The correct register and bit-position to enable a specific interrupt can be found in the datasheet of the processor.
3. *Implement the ISR in the code* Each interrupt has its own ISR that is called when the interrupt occurs. The name of the ISR links it to a specific interrupt. For instance, *TIMER0 OVF* links to a timer overflow interrupt. **Note that if the ISR for a triggered interrupt is not available the code will crash.**

B.3 Useful interrupts on the ATMega

Below some useful interrupts of the ATMega processor are listed and briefly described. More information on these interrupts and their options can as always be found in the datasheet of the processor.

1. *TIMERx COMPy:* timer x reaches a predefined value COMPy.
2. *TIMERx OVF:* interrupt generated when a timer x reaches it maximum value (this value is either $2^n - 1$ or a value set in software). On overflow the timer automatically resets itself.
3. *TIMER1 CAPT:* interrupt when timer1 captures its current value based on a pin change or analog comparator edge.
4. *PCINTx:* pin change interrupts can be configured to generate an interrupt when the value on a specific or a set of input pins changes from low to high or high to low.
5. *ADC:* the ADC can generate an interrupt when a conversion is complete.
6. *ANALOG COMP:* aside from triggering the input capture event in timer1 the analog comparator itself can also directly generate an interrupt.

B.4 Useful interrupts on the Nucleo

The Nucleo has similar interrupts to the AtMega but under different names. Further the Nucleo offers additional interrupts and allows interrupts to be triggered by software (this comes in handy when debugging).

For more information about the Nucleo's interrupts the datasheet is available in the documentation [6].

C Timers

C.1 What are timers

Timers or counters are valuable hardware components on a processor that allow the programmer to do timed operations for real-time applications such as motor control. The operating principle of such timers is simple, when enabled they increment (or decrement) their value at each rising edge of their clock signal. By scaling the frequency of the clock signal the timer either runs faster or slower (note that this scaling can be set in software, see datasheet).

The highest possible value of a timer depends on the amount of bits it has. An n-bit timer can have as highest value $2^n - 1$. When a timer reaches its highest value it does not simply stop (unless programmed to do so), but overflows an restarts from zero. This overflow event can be used to trigger an interrupt.

Further timers often provide the option to compare the timer value with a certain value that is written to one of the compare registers of the timer. When the timer reaches a value in one of its compare register a signal toggle is generated (if the option is enabled, see datasheet), this toggle can sometimes be linked directly to an output port. This allows the user to create a PWM signal. Besides the compare match event can also trigger an interrupt making it easy to schedule the execution of several blocks of code with one timer.

C.2 How to use timers

A processor can contain several different timers each with their own specifications and features (e.g. number of bits, frequency pre-scale settings,...).

Below the steps to use a timer are discussed.

1. *Setup the timer:* each timer has one or more control registers in which their behavior is to be defined.

2. *Setting the timer frequency:* setting the correct 'speed' for the timer. Too fast and the timer constantly overflows, too slow and the (real-time) timing resolution degrades.

3. *Set the output compare value:* this value is constantly compared against the current value of the timer. Can be used to trigger interrupts or generate fast hardware PWM signals.

4. *Reading the timer value:* the current value of the timer is available at all time through a simple register read operation.

C.3 ATMega timers

The ATMega processor comes with three hardware timers, two 8-bit timers and one 16-bit timer. All three these timers provide two output compare values that can be used to generate PWM signals. Hence the processor can at most output six different PWM signals generated directly from hardware.

Note that the 16-bit timer is the only timer with the input capture capability. Thus if the input capture in combination with the analog comparator is to be used to accurately measure the time of the BEMF zero crossing, this timer has to be used to generate the control signals of the DC/AC converter.

C.4 Nucleo timers

The Nucleo has six hardware timers, all of these timers provide at least the same basic functionality as the timers from the AtMega. Al most each timer of the Nucleo has its own dedicate features that make it very interesting to use it for a certain application or task. The datasheet contains the full description of these timers and their features.