Implementing Complex Motor Control Algorithms with a Standard ARM® Processor Core

Author: Mike Copeland

Introduction/Abstract

In the real-time MCU world, cost-effective complex motor control designs have been dominated by specialized cores. In many cases dual-core systems have been used, with the main core handling the control algorithm and a second "mini" core managing the real-time I/O and data manipulation.

This article describes how complex motor control algorithms can be easily and straight forwardly implemented using MCUs that contain a single Cortex™-M4F core, when used in combination with smart connected peripherals such as those found in the new Infineon XMC4000 family of products.

As an example, we will look at the equations involved in Field Oriented Control (FOC) of a Permanent Magnet Synchronous Motor (PMSM), and show how they can be handled using the CMSIS DSP Library. The same principles used in this example can be applied to other control algorithms and other motor types. We will see how smart peripherals eliminate the need for a second core, and describe some of the many benefits of using a single industry standard core with the CMSIS DSP Library.

PMSM Structure and Operation

Before we look at the control algorithms and equations, it is important to understand the structure and operation of a PMSM.

A simple three-phase PMSM consists of a permanent magnet rotor and a stationary stator with three sinusoidally distributed windings. Figure 1 shows a cross-section of a simple PMSM (Yes, it did me take a long time to draw all of those circles!)

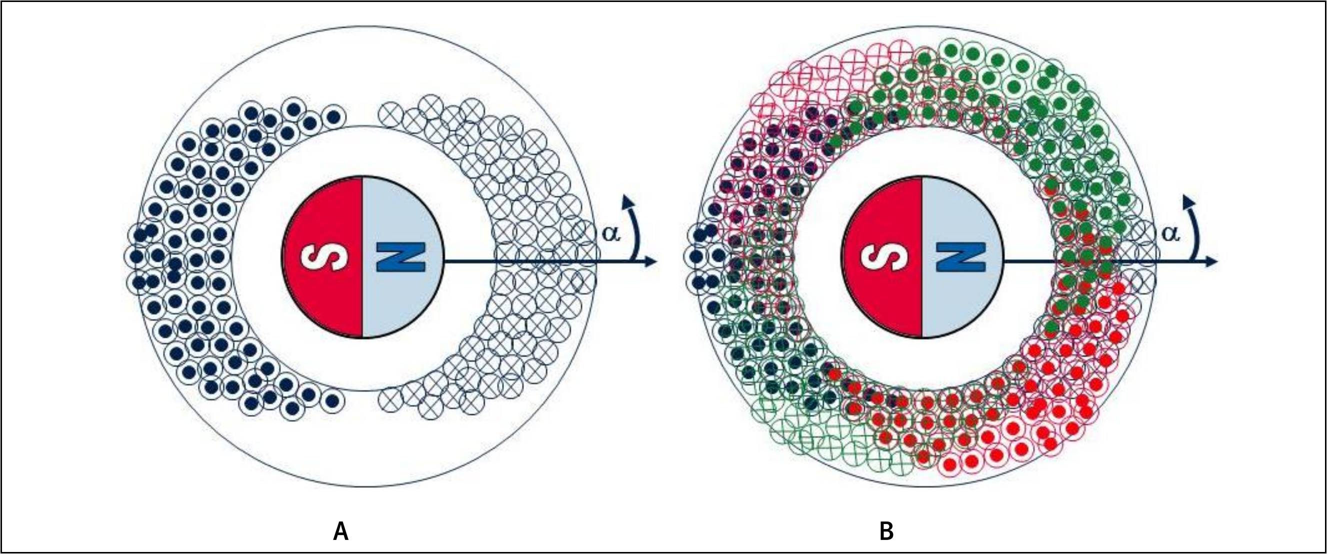


Figure 1 Cross-section of a basic PMSM with a 2 pole rotor and a single stator winding (A), and the complete three phase windings separated by 120 degrees (B). The stator windings are distributed sinusoidally. Circles with a dot or an X in the middle represent wires directed out of or into (respectively) the page.

Author: Mike Copeland

The sinusoidally distributed windings in the stator are similar to what would be found in a three-phase induction motor. In one phase (e.g. phase A) the number of turns of wire at any angle (α) is approximately NS cos(α). The other two phases are identical, but shifted 120 degrees. In reality, the windings are only approximately sinusoidally distributed. Fortunately for FOC, close is good enough.

As Mr. Tesla discovered, if you apply three phase sinusoidal currents to this type of stator a rotating sinusoidally distributed flux is created. We can prove this mathematically. The flux generated by a winding is proportional to the current through the winding and the number of coils in the winding. So by multiplying the number of coils by the current in that phase, we get an indication of the flux generated by that stator phase. This is called the Magnetomotive Force (MMF). When combined with the result from the other two phases, we get the total flux in the stator. Then, find a high school student and ask him or her to use the “Law of Cosines” to reduce the long equation into a simple one, as shown below.

$$\mathrm{MMF} = N_S\cos(\alpha)\bullet I_S\cos(\omega t) + N_S\cos\left(\alpha - \frac{2\pi}{3}\right)\bullet I_S\cos\left(\omega t - \frac{2\pi}{3}\right) + N_S\cos\left(\alpha - \frac{4\pi}{3}\right)\bullet I_S\cos\left(\omega t - \frac{4\pi}{3}\right) = \frac{3}{2} I_S N_S\cos\left(\omega t - \alpha\right)$$

To efficiently control the motor the goal is to produce stator flux that is 90 degrees out of phase with the rotor flux. The torque of the motor is then proportional to the amplitude of the stator flux.

Simplified FOC Theory

Several years ago I worked on an MCU based Uninterruptable Power Supply (UPS) project. I used the MCU with an integrated ADC and PWM module connected to a half bridge and a transformer to generate sinusoidal voltages. The output voltage was supposed to be a clean 110V 60Hz sinusoid independent of the load. To generate the PWM values, I first used a simple PI controller with a 110V 60Hz reference signal. The intention was to read the actual voltage via an ADC channel and compare it to the desired voltage, feed the error into a PI controller and use the output to control the PWM value. Unfortunately this did not work very well and I nearly fried the board before I had to give up.

I thought about implementing a complex non-linear control law instead of a PI controller. Non-linear control was not one of my strengths in college, so I decided to look for an easier solution. I created a look-up table of sinusoidal PWM values and stored it in the MCU memory. Once every PWM interrupt, I incremented the index into the look-up table and copied the value into the PWM register. This produced a nice 60Hz sinusoidal signal, but it was very dependent on the load. To control the amplitude of this sinusoidal voltage, I multiplied the values from the look-up table by a scale factor before transferring them to the PWM register. Then I used a simple PI controller to control the scale factor. This worked much better. With some trigonometry (in this case just a simple sin() calculation via a lookup table), the simple PI controller could work just on the magnitude of the voltage which was approximately DC.

FOC works on a similar principle. Remove the sinusoidal properties of the system with trigonometry, then apply simple linear PI controllers on the amplitudes.

To see how this works, let’s simplify the drawing of **Figure 1** (B) by using vectors. **Figure 2** is similar to **Figure 1**, except all of those little circles that were so difficult to draw are replaced by the a, b and c axes. Current flowing through phase “a” (the winding that was drawn in dark blue in **Figure 1**) can be represented as a vector on the “a” axis (i_a). The same is true for phases b (the red winding) and c (the green winding).

Since the flux is proportional to the current, the terms flux and current are interchangeable. The total flux produced by the three stator currents is shown by the vector iS, which is the vector sum of $i_a + i_b + i_c$. The goal is to keep i_S 90 degrees from the flux generated by the rotor magnets. In **Figure 2** (B) the stator flux is aligned properly if i_s is aligned with the rotor quadrature or q-axis. In this figure you can see that there is some misalignment.

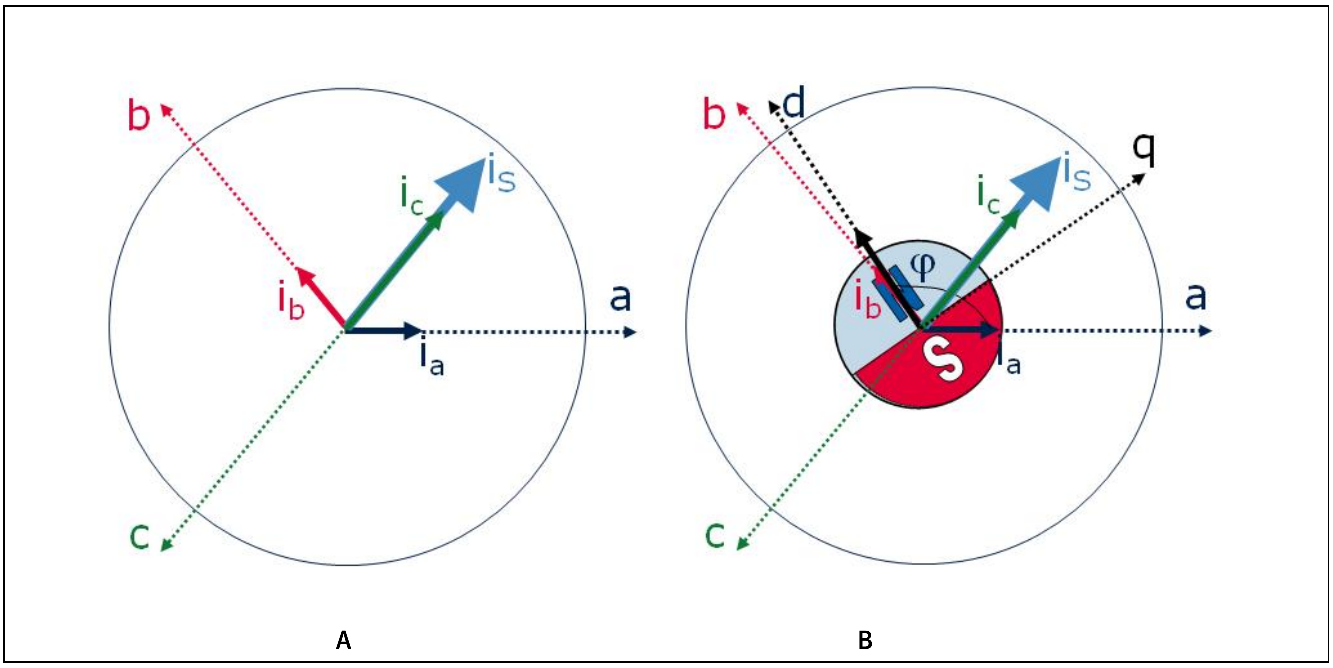


Figure 2 **(A) Vector representation of the three-phase PMSM showing the three phase currents (i_a, i_b, i_c) and their vector sum (i_s).**
In (B) the rotor is also shown with it's flux oriented around the spinning "d" axis. For maximum efficiency, i_s must be aligned with the rotor quadrature (q) axis.

To align the stator flux with the rotor, we must first identify the component of the stator flux that is properly aligned with the rotor. This is the component of the flux that is aligned on the rotor q-axis, called i_q. We must also identify the component that is not properly aligned with the rotor. This is the component of the flux that is aligned on the rotor d-axis, called i_d.

i_d and i_q are simply the projections of i_s onto the d and q axes, as shown in **Figure 3**.

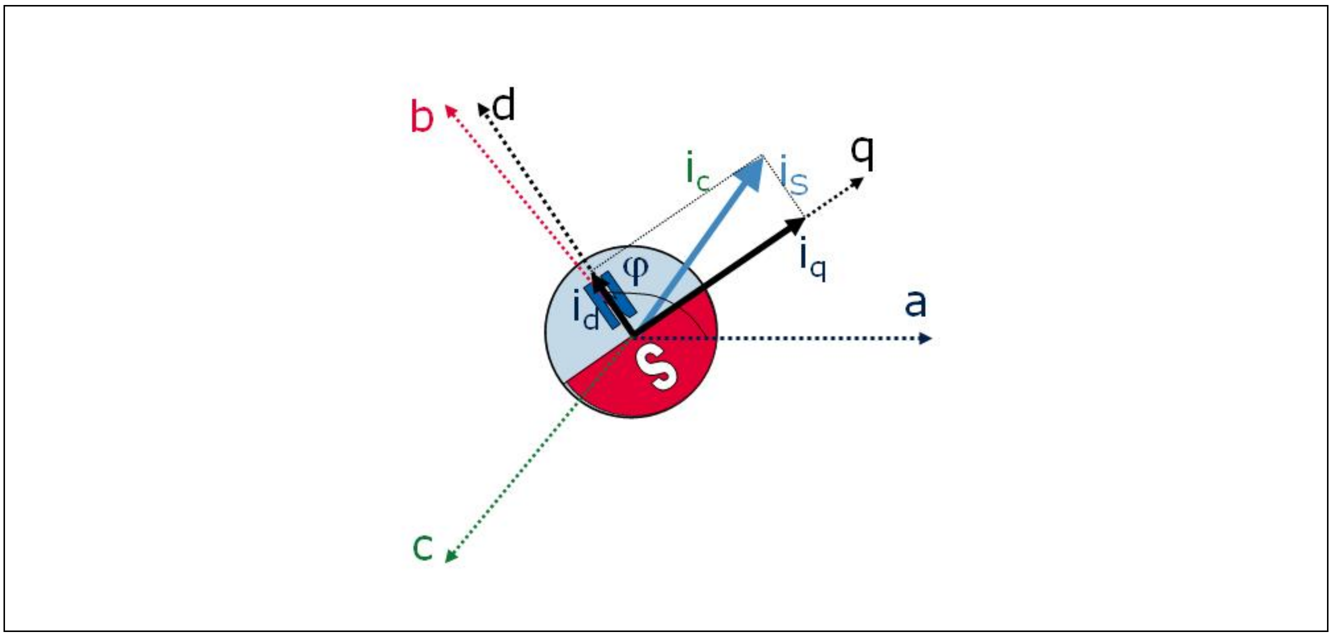


Figure 3 **i_q is the component of the stator flux that is producing useful torque. i_d is the component of the stator flux that should be controlled to zero. They can be found by projecting i_s onto the d and q axes. Note that the rotor angle is represented by φ.**

All of this geometry and trigonometry looks great on paper, but we need some equations to get i_d and i_q given the stator currents and the rotor position. **Figure 4** is included to assist those of you that would like to derive the equations yourself. We typically convert the stator currents and rotor position into i_d and i_q via a two step process. The first step is called the Clarke Transform. The Clarke transform converts the i_a, i_b and i_c values into a fictional i_α and i_β which are located on the orthogonal α and β axes shown in **Figure 4** (A) in light green. The second step is to convert i_α and i_β into i_d and i_q given the rotor angle (φ). This is called the Park Transform. To derive these equations you also need to know that $i_a + i_b + i_c = 0$.

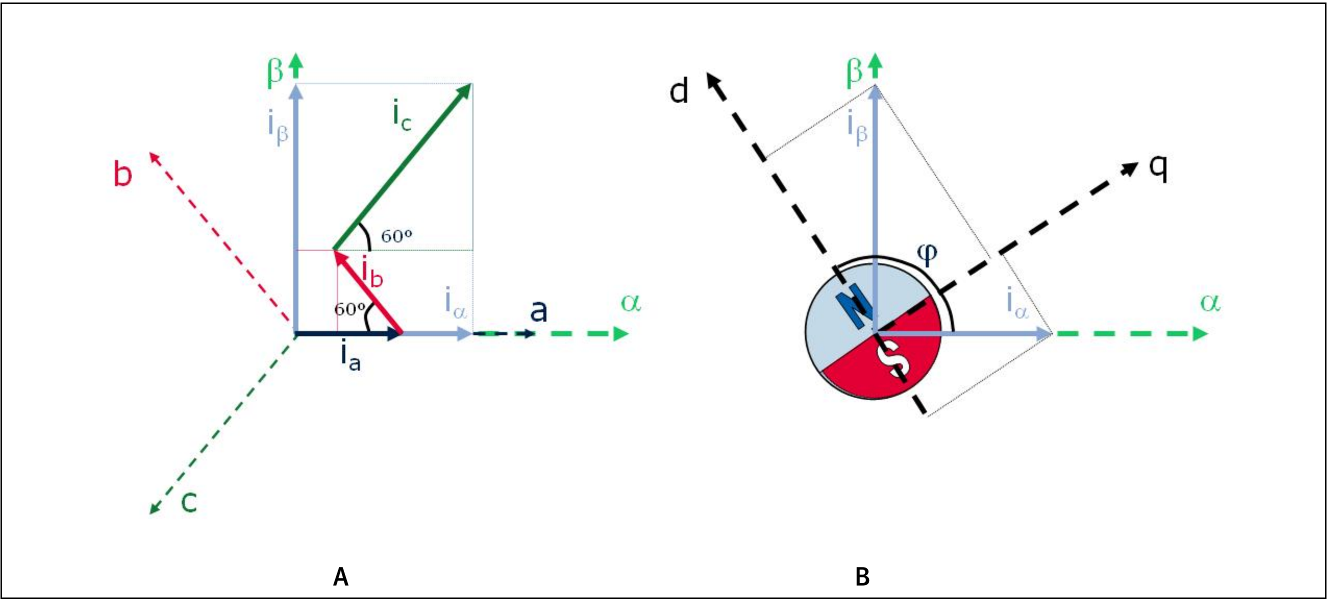


Figure 4 Graphical Representation of the Clarke (A) and Park (B) Transforms

Clarke Transform:

$$i_\alpha = i_a$$
$$i_\beta = \frac{1}{\sqrt{3}} i_b - i_c$$

Park Transform:

$$i_d = i_\alpha \cos\varphi + i_\beta \sin\varphi$$
$$i_q = i_\beta \cos\varphi - i_\alpha \sin\varphi$$

Once you have i_d and i_q, they can be run through the PI controllers. One PI controller controls i_d to zero to ensure that the stator flux is 90 degrees from the rotor flux, and the other PI controller controls i_q to the commanded motor torque.

The CMSIS DSP Library contains fixed point and floating point versions of the Clarke and Park transforms, as well as a PID controller (a PI controller is simply a PID controller with the derivative constant, K_D, set to zero). **Figure 5** shows the block diagram of a simple PI controller. The error signal e(t) is the difference between the commanded id or iq and the actual i_d or i_q that is calculated from the measured stator currents and rotor position.

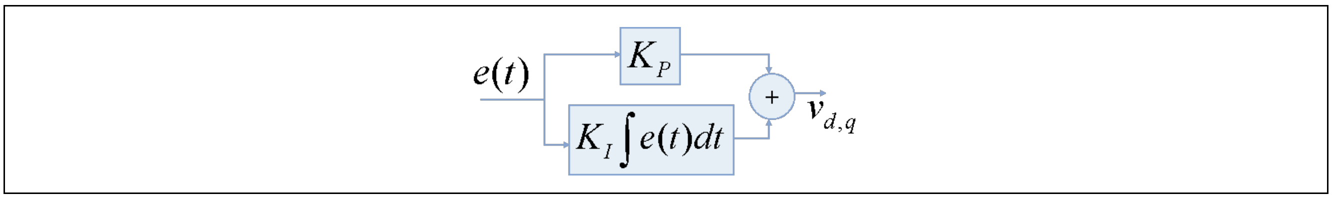


Figure 5 A simple PI controller in continuous time.

Note that the output of the PI controller is a voltage, v_d or v_q. These are voltages that have no real physical meaning. You cannot directly convert v_d and v_q into duty cycles for a PWM unit, so more trigonometry is required.

There are many ways to convert v_d and v_q into duty cycles for a three phase inverter. We will look at just one method called Space Vector Modulation (SVM). An entire article could easily be devoted to the theory of SVM (indeed I wrote one several years ago!), but for the sake of brevity we will focus on the basics.

v_d and v_q are referenced to the rotor position. We need to project these voltages onto actual voltage vectors that can be produced by a three phase inverter.

Figure 6 shows a three-phase inverter. There are six non-zero voltages that can be produced by the inverter. Each vector is produced by turning on one high-side switch and two low-side switches, or two high-side switches and one low-side switch. These voltage vectors and the inverter state that is used to reach them are shown in **Figure 7B**.

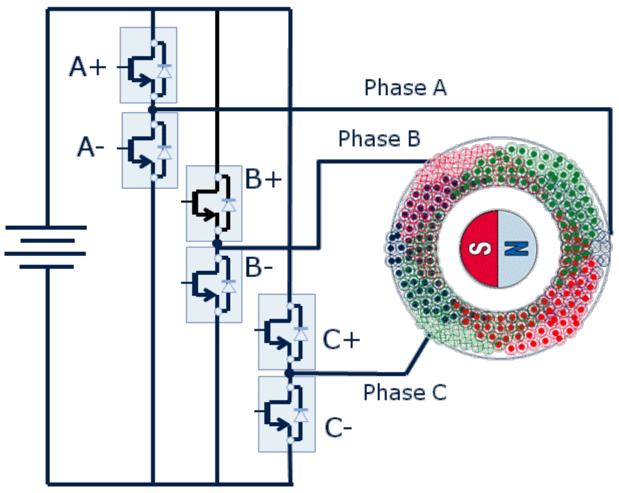


Figure 6 A three phase inverter connected to a three-phase PMSM

We will use a two step process to project v_d and v_q onto inverter voltage vectors. The first step is the Inverse (or reverse) Park Transform. As the name implies, it converts the v_d and v_q quantities into v_α and v_β as shown in **Figure 7A**. The Inverse Park Transform is included in the CMSIS DSP Library.

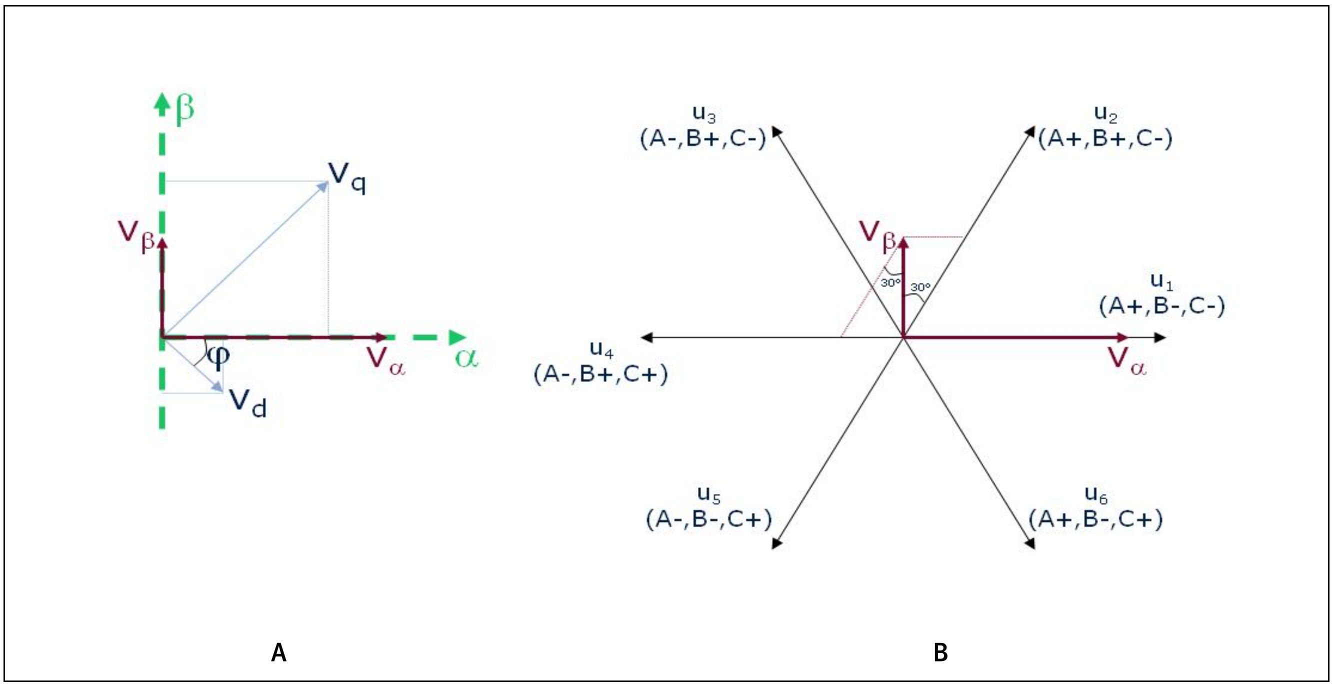


Figure 7 **(A) shows the conversion of v_d and v_q into v_α and v_β via the Inverse Park Transform and (B) the conversion of v_α and v_β into the switching states u_1 and u_2.**

Inverse Park Transform:

$$v_\alpha = v_d\cos\varphi - v_q\sin\varphi$$
$$v_\beta = v_q\cos\varphi = v_d\sin\varphi$$

Once v_α and v_β are known, they can be projected onto the switching state vectors that are closest to their vector sum. This projection is tricky because it is not done orthogonally, but along 60 degree lines (see **Figure 7B**). Below is the formula for the projections when the vector sum of $v_\alpha + v_\beta$ is between u_1 and u_2. Note that this formula varies depending on the location of the vector sum of $v_\alpha + v_\beta$:

$$t_1 = v_\alpha - \frac{v_\beta}{\sqrt{3}}$$
$$t_2 = \frac{2}{\sqrt{3}} v_\beta$$
$$t_1 = 1 - t_1 - t_2$$

t_1 and t_2 are the projections of v_α and v_β onto the two inverter voltage vectors that are the closest to the vector sum of $v_\alpha + v_\beta$. They represent the percentage of time that each switching state should be active to produce the desired voltage. When $t_1 + t_2 < 100\%$, there is some time left called t_0. During t_0 all of the high-side or all of the low-side switches can be turned on.

Some additional rectangular to polar conversions are required to find the angle and magnitude of $v_\alpha + v_\beta$. SVM can be complicated, and the details are outside of the scope of this article.

Author: Mike Copeland

Complex Motor Control Systems with a Standard Core

In the previous section we saw many of the basic equations that must be handled by an MCU core to perform advanced motor control. There are also other additional equations that come into play for a full implementation, and the complexity is doubled in systems with no rotor position sensor.

Although the equations are complex, they can and have been implemented in many (typically 16 and 32-bit) MCUs. It is clear that a fast CPU with DSP extensions and floating point capability enables the calculations to be performed fast and easily. This in turn enables more complex algorithms (e.g. FOC without a position sensor) to be implemented.

So why are MCUs with asymmetric dual cores so popular in motor control systems? The answer to this question lies in the fact that a motor control MCU must do more than crunch numbers. Before any equation can be calculated the input data must first be read. This data comes from ADC peripherals, I/O and Serial ports, and is extremely real-time sensitive. And once the calculations have been completed the results need to be scaled and transferred to output peripherals. The synchronization of all of the input/output functions is just as critical as the ability to perform the high level calculations.

Using an additional CPU to schedule and read ADC results, setup PWM values, read and filter sensor inputs, etc., can be very helpful. However there are also many disadvantages in using a separate asymmetric core. Extra effort is required to determine the best way to partition and synchronize the tasks between the cores for example. More time is required to learn the additional instruction set, architecture quirks and tool-chain for the additional core. When all of that work is finished, almost none of it is portable to MCUs from another supplier.

Even though the tasks performed by the additional core are usually quite simple, those tasks must be performed with hard, real-time constraints. Using a single core would clearly be beneficial, but meeting the real-time requirements while performing the higher level algorithms and responding to the user interface can be a challenge, even for a Cortex™-M4F core.

Most of the additional tasks the second CPU would be used for consist of moving data and synchronizing I/O events. These tasks can equally be accomplished by smart peripherals linked together via a flexible connection matrix.

As a simple example, consider the task of decoding the rotor position by processing the signals of a quadrature encoder. A quadrature encoder is a rotor position sensor that produces pulses on two different pins that are 90 degrees out of phase. Each time an edge is detected on one of the pins, the rotor has moved some small fraction of a degree. The direction of the rotation is detected by the phase shift (+90 degrees or -90 degrees) between the sensor pins. Additionally there is usually a third index pin that indicates when one complete revolution has occurred. **Figure 8** shows the signals from a quadrature encoder.

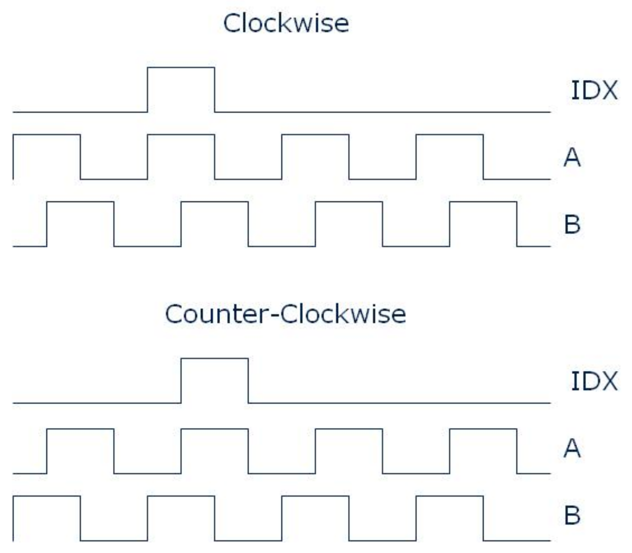


Figure 8 Output from a quadrature encoder when the motor is spinning clockwise and counter clockwise.

Figure 9 shows an example of how the smart peripherals in the Infineon XCM4000 family of products can interface to a quadrature encoder without the need for a second CPU. The Position Interface peripheral (POSIF) is set up in Quadrature Decoder mode and conditions the signals from the encoder. It is also connected via a connection matrix to one of the capture/compare modules (CCU4). The counter in Slice 0 of the CCU4 is incremented and decremented automatically (depending on the motor direction) and contains the motor position. The counter in Slice 1 is incremented each revolution based on the IDX signal to provide a revolution count. Slices 2 and 3 are both used to capture velocity. Slice 2 measures the number of encoder edges per given time interval. Slice 3 measures the time between a fixed number of encoder edges. This allows very fast and very slow speeds to be accurately measured without the need for the CPU to reconfigure the peripheral.

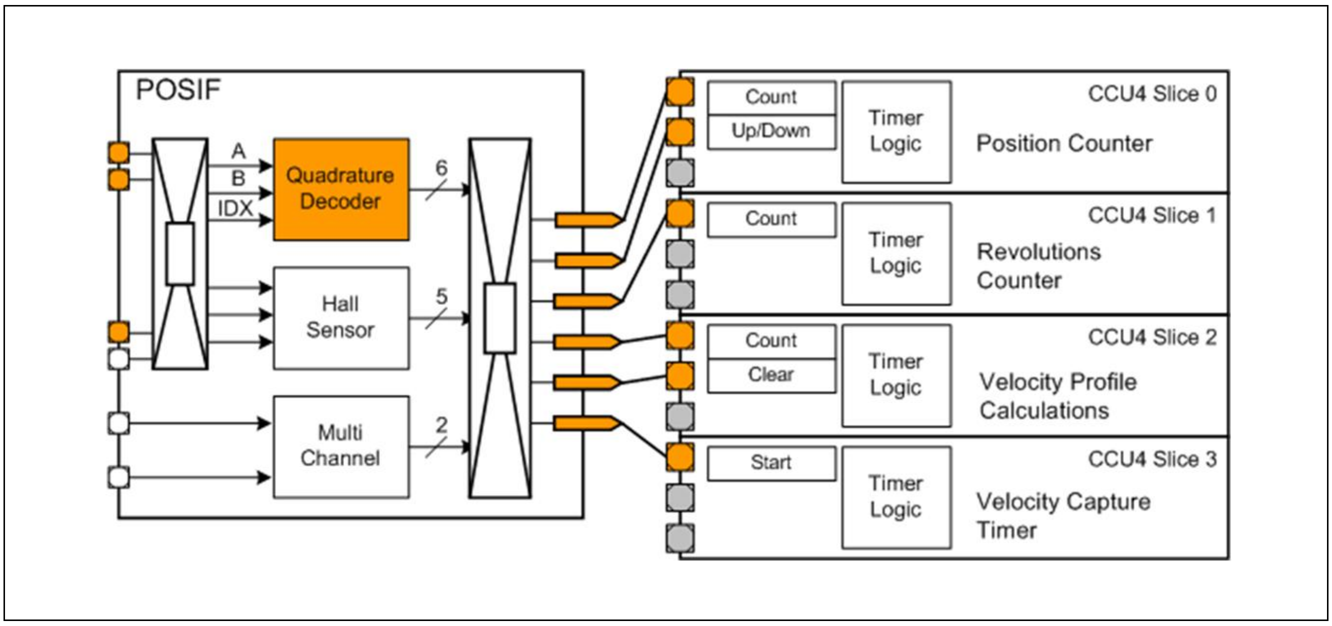


Figure 9 Connection and configuration of the POSIF and CCU4 for decoding a quadrature encoded rotor position and speed.

Using smart connected peripherals, a much more complicated motor control system can be built without the need for a second core to handle the real-time requirements. Figure 10 shows a system where the XMC4000 is controlling a 3 phase motor with a tachometer and a Power Factor Correction (PFC) circuit using 2 ADC modules, 3 PWM modules (two CCU4s and one CCU8) and a flexible external interrupt unit called the External Request Unit (ERU). The basic use and configuration of each module is described in the figure.

Author: Mike Copeland

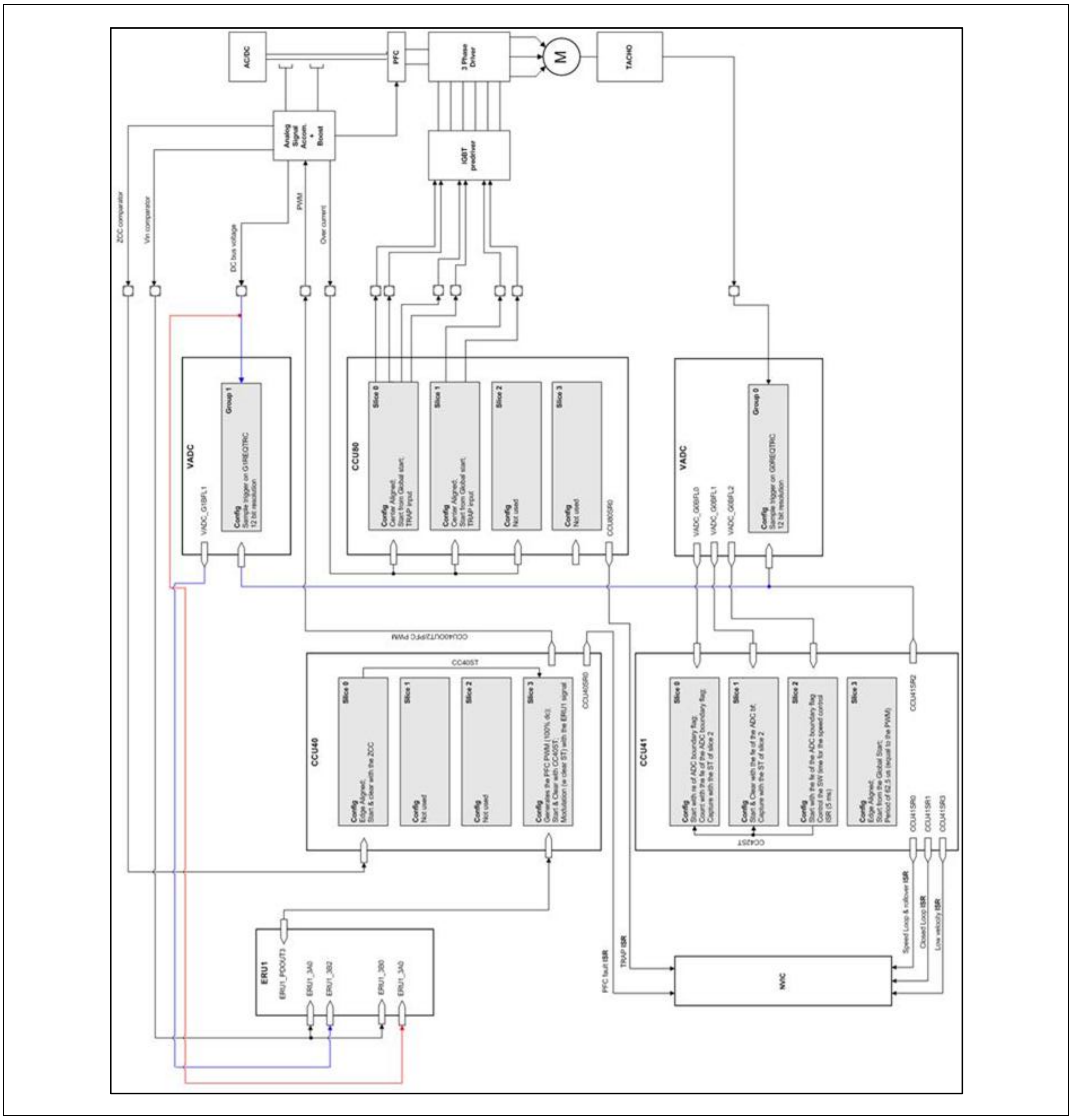


Figure 10 Use of several smart peripherals and a connection matrix to eliminate the need for a second CPU in a motor control system

Conclusion

Complex motor control algorithms require sophisticated mathematics and hard, real-time performance. The ARM® Cortex™-M4F CPU operating at >100MHz with DSP extensions and a hardware floating point unit, has the CPU bandwidth to perform the high level calculations required in motor control systems. The CMSIS DSP Library contains many useful functions such as the Clarke and Park Transforms, and PID controllers in both fixed and floating point format. This makes the high level mathematics easier to implement and more portable. With smart connected peripherals such as those found in the Infineon XMC4000 products, complex high-end motor control algorithms can be easily implemented with a single, standard core, reducing development effort and increasing portability.

Author: Mike Copeland

DAVE™ 3 – Infineon's Free Tool-Chain for XMC4000 MCUs, with DAVE™ Apps for Component-Based Programming

DAVE™ 3 is a free tool-chain for Infineon's new XMC4000 family of ARM® Cortex™-M4 microcontrollers. Built on the Eclipse platform, DAVE™ 3 includes the GCC ARM® embedded compiler/assembler/linker-locator, and a hardware debugger.

There are no code size or time restrictions with DAVE™ 3, and a code generator (DAVE™ 3 CE) is included to support component-based programming via DAVE™ Apps.

DAVE™ 3, DAVE™ 3 CE and DAVE™ Apps are all, 100% free.

Eclipse-Based Integrated Development Environment (IDE)

The open source Eclipse IDE has become a standard among embedded developers. Many MCUs are supported via an Eclipse-based IDE provided either by a third party tool supplier or by the silicon vendor themselves. An Eclipse plugin from one supplier can be inserted into the Eclipse environment from another, so DAVE™ 3 can utilize plug-ins from other tool suppliers and vice versa. The DAVE™ Eclipse platform includes plug-ins for:

- GCC ARM® embedded compiler (maintained by ARM®)
- A hardware debugger and simulator. The debugger uses the Infineon DAP MiniWiggler or SEGGER J-Link as the HW interface and includes all the typical debugger features, such as a flash programmer, HW break points and single stepping.
- DAVE™ CE for code generation
- X-Spy for transmitting and receiving MCU data via a virtual COM. X-Spy allows you to create your own GUI to interact with your target hardware without any PC side programming. X-Spy also includes an oscilloscope feature.

Component-Based Programming and DAVE™ Apps

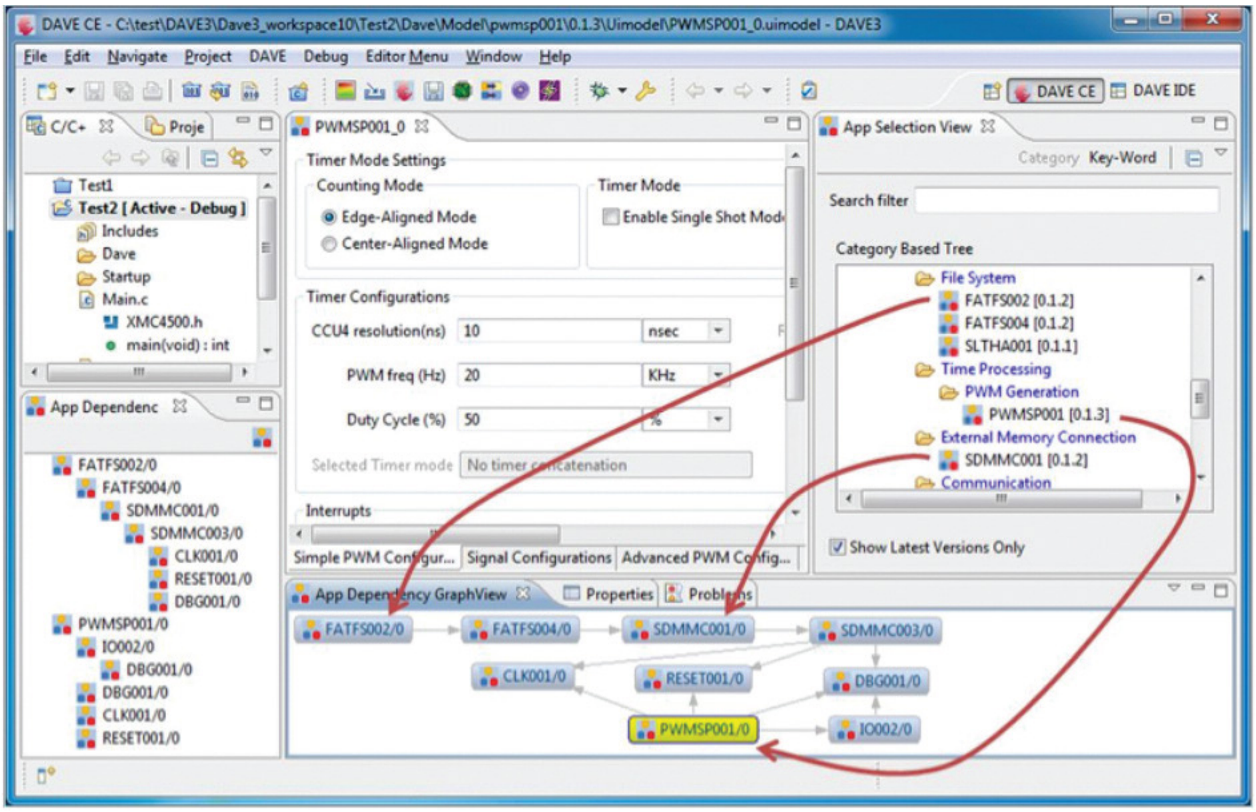
In component-based programming, software is partitioned according to functionality. Programmers often do this partitioning naturally without even noticing. The DAVE™ 3 logo, of a head comprised of small boxes (components), illustrates this principle.

DAVE™ Apps are used to create software components for specific functions.

Features include:

- Optional Graphical User Interface (UI) to configure the DAVE™ App.
- API with initialization and run-time functions. (All source code is included).
- Documentation and example App API use.
- Interconnection between DAVE™ Apps via "virtual signals". Virtual signals represent the connection matrix that connects XMC4000 peripherals to each other and to I/O pins.
- Apps can be written to consume specific pieces of hardware (e.g. CCU40_Slice1, ADC0_Channel6), or can be written to use a generic hardware resource (e.g. any CCU4 slice, any ADC channel). The user is then free to constrain the DAVE™ App to use specific resources as required (e.g. use Port 3 Pin 9 for PWM output, use Port 14 Pin 1 for ADC input). The integrated "Solver" will then choose resources that are not constrained.
- Apps can be used to generate low level drivers for specific peripherals (similar to DAVE™ 2), or they can be used to generate and configure complex SW components such as an RTOS or file system.

As an example, let's look at an application that needs to produce some PWM signal and store data on an SD card. In DAVE™ 3 this type of application can be built in minutes using the DAVE™ Apps.



Timer Mode Settings

Counting Mode: (●) Edge-Aligned Mode ( ) Center-Aligned Mode

Timer Mode: [ ] Enable Single Shot Mod

Timer Configurations

CCU4 resolution(ns) 10 nsec

PWM freq (Hz) 20 KHz

Duty Cycle (%) 50 %

Selected Timer mode No timer concatenation

[✓] Show Latest Versions Only

- Double clicking the PWM App (PWMSP001) in the App Selection View window automatically inserts the PWM App and any other DAVE™ Apps that are required (the Clock App, Reset App, I/O App and Debug App).
- Double clicking the SDMMC App (SDMMC001) adds the SDMMC App and its required Apps (SDMMC low level driver App, Clock App, Reset App and Debug App).
- A FAT32 files system can be added if required by simply double clicking on the App (FATFS002).

Once the Apps are inserted into the project they can be individually configured via their UIs as shown in the center window for the PWM App. Additionally, the “Pin Configurator” can be used to specify exactly which pins are used for WM and SDMMC, if required. Any constraints not specified (e.g. which CCU4 slice or which I/O pins are used) can be automatically selected via the Solver. The Solver is a key feature of DAVE™ 3. The user develops their application based on the functionality that they need, but then the Solver handles the task of mapping that functionality to the actual hardware.

In our example PWM and SDMMC application, we can easily continue to add more features by including more DAVE™ Apps. To add additional PWM channels for example, simply double-click on the PWM App again and a new instance of the App is inserted. The App source code is not duplicated, but a new handle is created to use when calling the API.

The code generated by DAVE™ Apps in DAVE™ 3 is included in the project and is freely available for modification, as are the templates used to generate the source code based on the UI settings. So if there is something you would like to change in an App, you can easily do so.

The Future of DAVE™ 3

DAVE™ 3 continues to grow as Infineon constantly adds Apps for the XCM4000. In the long term, to further increase the number of available Apps, Infineon plans to release a DAVE™ 3 SDK to allow anyone to create their own Apps. It is then the developer’s choice to decide whether to keep the App confidential, sell it via an on-line store, or make it freely available to the wider community.

DAVE™ 3 can be downloaded for free from the Infineon website: http://www.infineon.com/dave3