Technical University of Munich
Department of Electrical and Computer Engineering
Chair of Electrical Drive Systems and Power Electronics

Stefan Baltruweit

Implementation of a Direct Model Predictive Current Control on an FPGA

Report and Manual

Research Internship

(Supervised by Eyke Liegmann)

June 2019

Table of Contents

1 Research Objectives and Overview

The goal for my Research internship was to understand and modify the open source project SPYN [1] while implementing a simple finite control set model predictive current control (FCS-MPC). The SPYN Project is based on the Xilinx® / Trenz Electronic “Electronic Drive Development Platform” [2] (EDDP) using the PYNQ project. The EDDP consists of software and hardware components to prototype and evaluate motor control applications. It runs on boards that have both a multicore microprocessor and a field programmable gate array (FPGA). The project already contains a field-oriented control (FOC) with pulse width modulation (PWM) that is implemented in the programmable logic (PL). The hardware part of the EDDP with the power electronics and measuring circuits is called “Electronic Drive Power Stage” (EDPS). The EDPS contains a three-phase two-level inverter and comes as a kit that already contains a reference permanent-magnet synchronous motor (PMSM) and a 1000 step encoder. Figure 2 shows the physical setup of the EDDP project with Controller board and EDPS. EDDP runs a simple Linux server on the microprocessor that makes it possible to control the motor and monitor the stator currents.

The PYNQ Project [3] on the other hand makes it possible to run Python code on the board’s microprocessor and to program the PL from within a browser on an Ethernet connected computer using Jupyter [4] notebooks. In those notebooks it is also possible to create graphical user interfaces and plot measurements. I was supposed to use the Diligent® PYNQ-Z1 board that is specifically designed to run PYNQ, but the project should also be executable on other boards like the Diligent® Arty Z7.

Figure 1 outlines the basic structure of the unmodified SPYN project. The PL is identical to the EDDP and only the blue parts differ. Instead of the EDDP’s simple server application PYNQ is running on microprocessors. The actual SPYN project solely consists of a Jupyter notebook and some Python libraries used in the notebook. In my research internship I extended the SPYN project and implemented the MPC into the PL besides the FOC. I named the resulting project SPYNX to be able to distinguish it from the unmodified SPYN project.

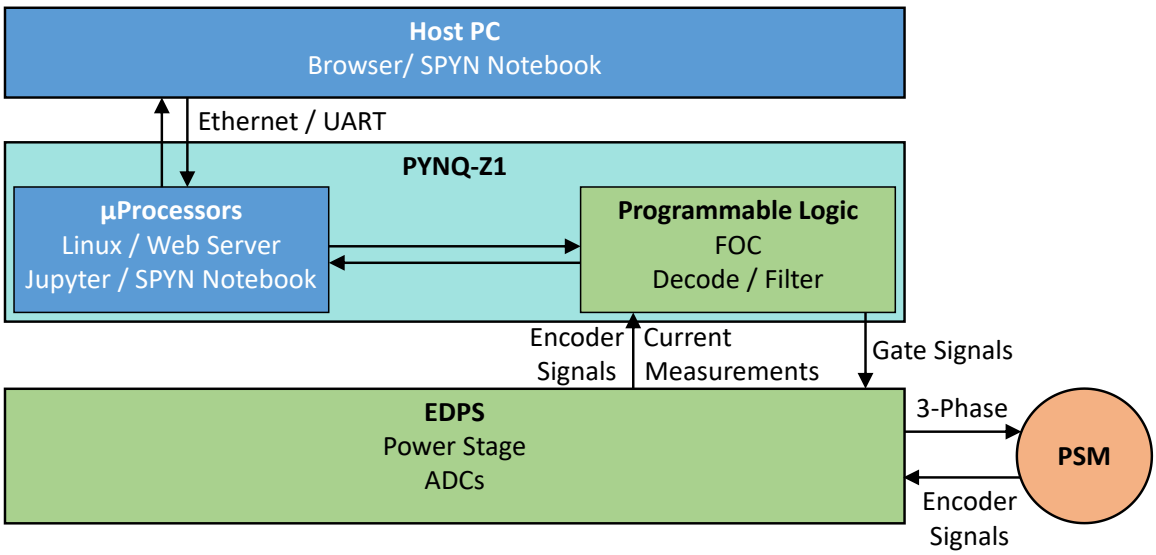


Figure 1 Simplified schema of the SPYN project. The hardware parts are filled green and the software parts blue.

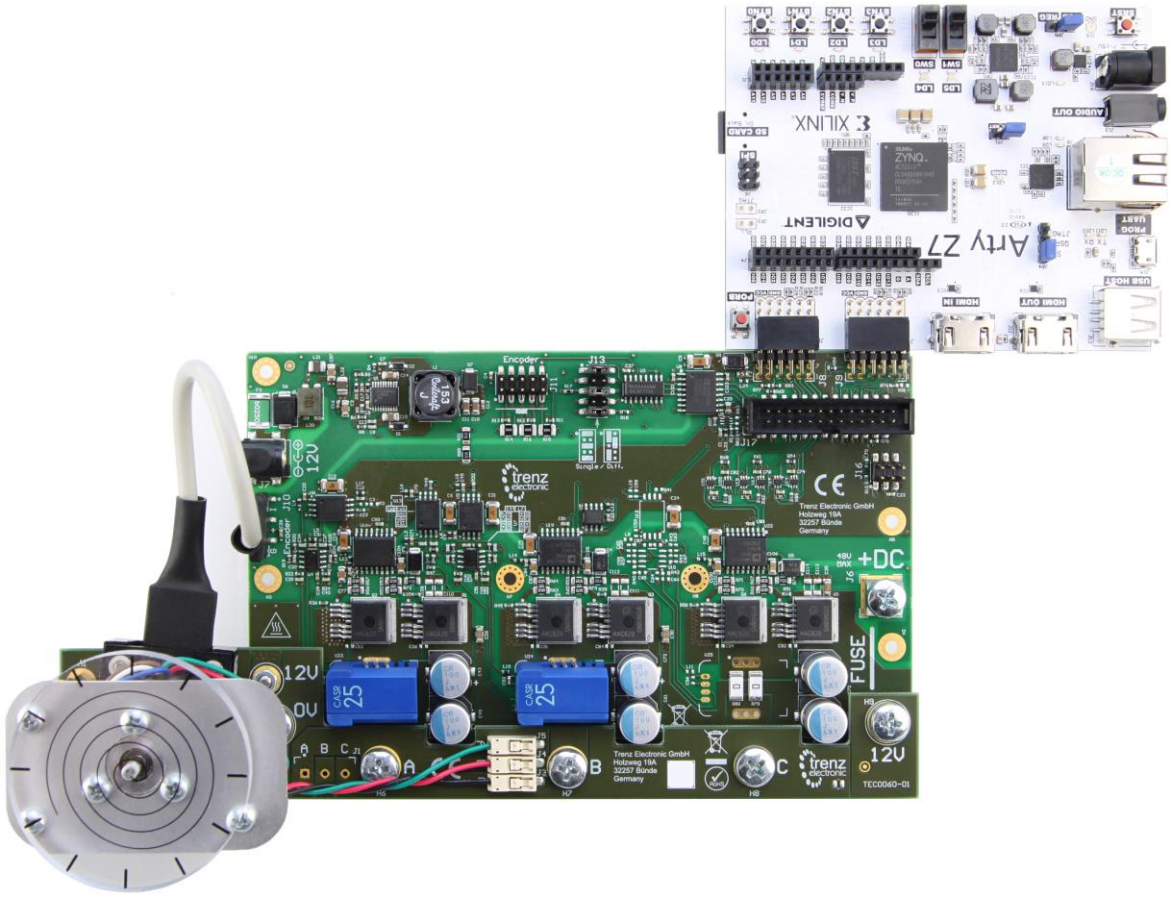


Figure 2 EDDP kit with EDPS, reference motor and the white control board Arty Z7. In this report the better equipped PYNQ Z1 is used as a control board *[2]*.

2 How to Use the resulting SPYNX-Project

2.1 Connecting via UART and Ethernet

First PYNQ needs to be set up on the board. Supported boards and the referring PYNQ images can be found at http://www.pynq.io/board.html [3]. Once the image is flashed to the SD card and the board is booting from the SD card a computer has to be connected via USB to set up a UART connection. With a serial console choose a baud rate of 115200, 8 data bits, and no parity. To log in the username *xilinx* and the password *xilinx* are set by default.

After the board is connected via Ethernet you can find out the IP address of the board in a serial console with the command *ifconfig*. The board can either be connected directly to the computer or via a network.

If the Ethernet connection is set up correctly you can use a browser to connect to the PYNQ's Jupyter user interface by typing the boards IP-address into the browser bar. Chrome, Safari and Firefox are currently supported (as of June 2019). Again, if you need to log in use *xilinx* both for username and password. In Figure 3 you can see how the user interface looks like. Support and guides for PYNQ can be found at http://www.pynq.io/ and at the PYNQ GitHub repository [5].

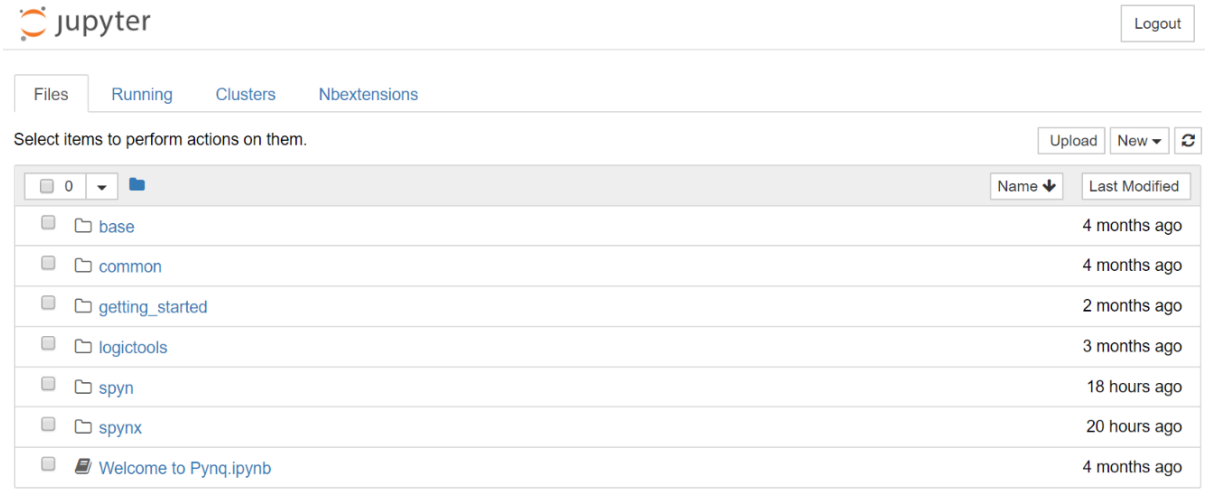


Figure 3 Jupyter user interface in the browser. The SPYNX Files have already been copied to the board.

2.2 Running the SPYNX Notebook

The SPYN library and overlay folders have to be placed at the path /usr/local/lib/python3.6/dist-packages/spynx/. The folder with the notebook must be placed at /home/xilinx/jupyter_notebooks/. The board can conveniently be accessed via SCP.

Once everything is copied to the correct folders you can navigate to the spynx folder in your browser on the Jupyter interface. With a click on *spynx.ipynb* you can open the SPYNX notebook as visible in Figure 4. The first cell has to be executed to download the SPYNX bit stream to the boards PL. In this bit stream both the FOC and the MPC are contained. Then import the SPYN library and run the encoder initialization with the cells 2 and 3. After that you can run cell 4 to create the motor control interface as shown in Figure 5. With the green *Motor* button you can start and stop the motor. With the drop-down menu you can choose whether the motor should run with a current or a speed controller. Note that for the MPC only the current control is available. With the first two sliders you can change the set points of the motor controllers. The third and last slider controls the switch cost weight coefficient λ_u described in section 4.1. The MPC relies on several signals from the

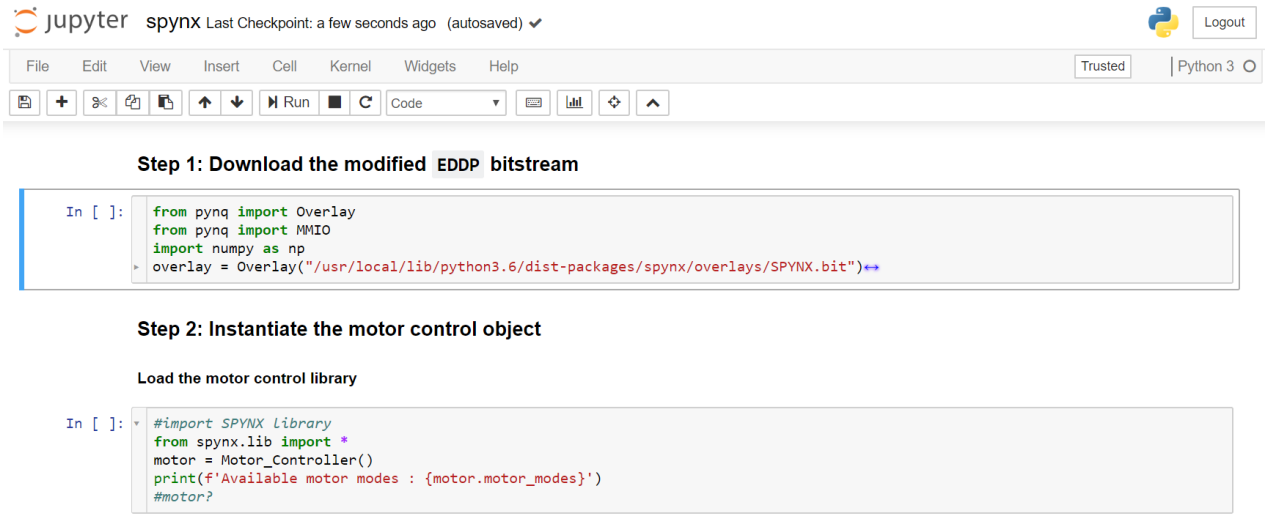


Figure 4 First two steps and cells of the SPYNX notebook.

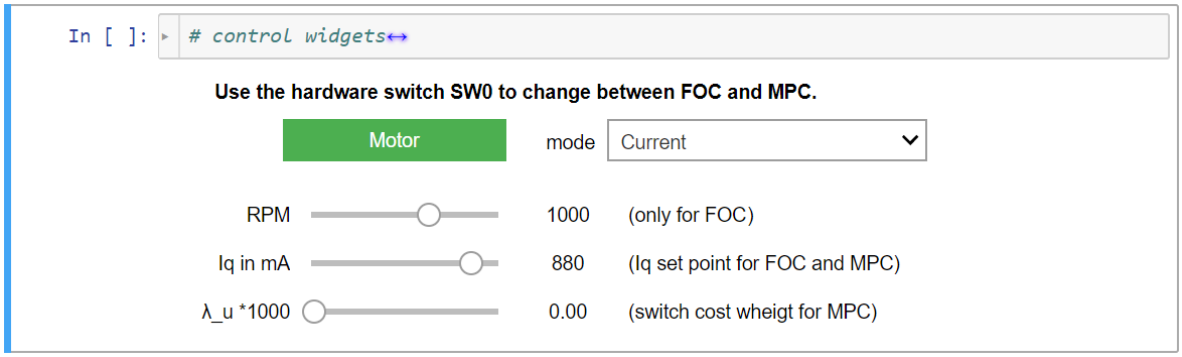


Figure 5 Graphical motor control interface with buttons, sliders and a mode drop-down menu.

FOC that are only available after the FOC has been run at least once. So only after the FOC has been run you can also run the MPC by using the switch SW0 on the control board. An illuminated LED LD0 on the control board indicates that the SW0 is in the MPC position. LD1, LD2 and LD3 are connected to the encoder Signals.

In order to capture data, go to step 5 and allocate memory. Only then you can run step 6 and create the capture interface shown in Figure 6. You can choose from 8 different data sources to be captured. Also, you can choose a decimation rate. A decimation of 2 means that only every other sample is captured, a decimation rate of 3 every third and so on. You can choose in the range from 32 to 8192 how many samples you want to capture. To capture data, click on the *Start the Capture* Button. Only when the FPGA has captured all the specified data you can click on *Load the Capture* to transfer the captured data to the allocated memory.
It is important to note that there are no voltage measurements performed on the EDPS. The voltage signals you can capture are only the PWM voltage set points created by the FOC. The MPC does not have any impact on these signals. Therefore it only makes sense to capture the actually measured current signals for the MPC.

Once data is captured you can proceed to step 7. The *pyplot* library has to be loaded first. Then you can create a variety of plots. In Figure 8 for example the three stator currents were plotted over the mechanic angle and in Figure 7 the current in phase B was plotted over the current in phase A.
Besides plotting, it is also possible to export the captured data as a CSV database. However, with SPYNX it is only possible to capture AXI bus signals. Also, there is no capture trigger available. So if other signals such as the phase gates are to be captured or a trigger is necessary, the Vivado debug capability must be used instead. The easiest way is to set up an Integrated Logic Analyzer (ILA) in the block design and connect all the required signals. Once the implementation is run the bit

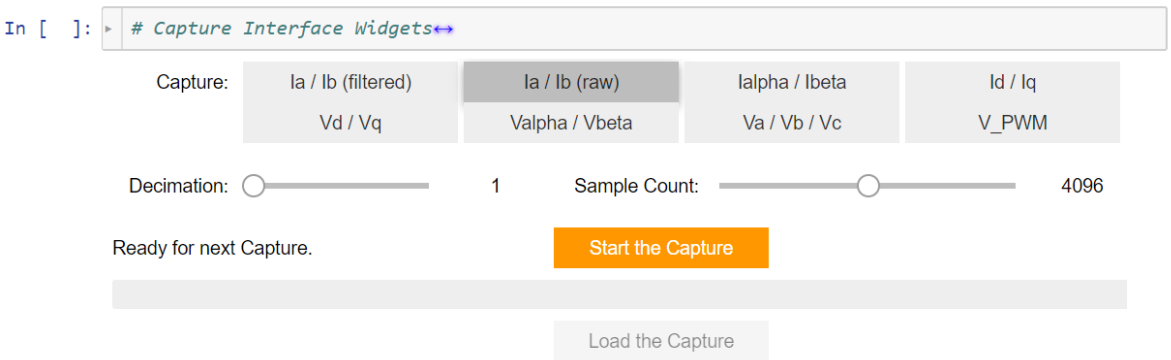


Figure 6 Graphical capture interface with buttons, sliders and a progress bar.

stream must be loaded onto the FPGA. This can either be done in PYNQ as an overlay or directly from within the Vivado hardware manager. Note that the encoder has to be initialized again and the FOC needs to be executed before the MPC can run correctly. The Vivado hardware manager allows you to access all signals and AXI busses connected to the ILA core. The sample rate can be as low as the 10 ns PL clock. If the ILA is configured accordingly, extended trigger and capture options are available.
After the required data has been captured it can again be exported as a **.CSV* file and handled with other programs. Usually there are a number of unnecessary signals that have to be removed before the file can be processed further.

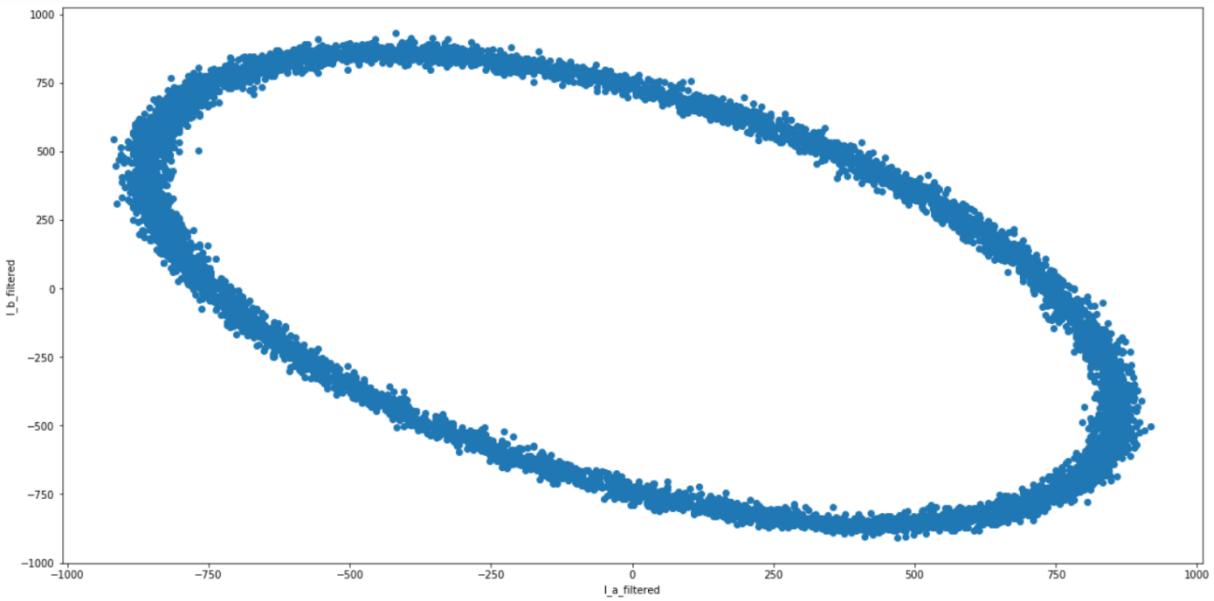


Figure 7 Plot of the MPC's filtered i_a vs i_b with an i_q set point of 880mA (axis units are mA).

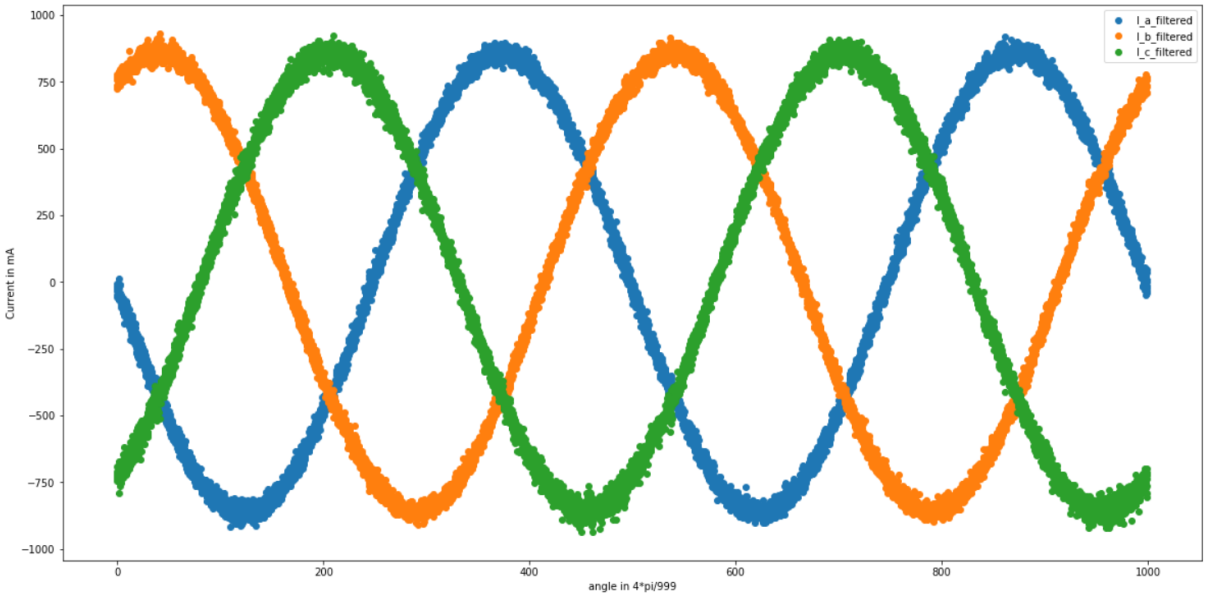


Figure 8 Plot of the MPC's filtered stator currents over the angle with an i_q set point of 880mA

3 Fixes and Modifications to the SPYN-Project

3.1 Changes to the EDDP-Project

Small modifications of the Vivado Project
I removed the spread spectrum capability of the EDDP as this feature already leads to timing problems in the original unmodified project during synthesis. In addition I removed the rpm_check IP. This IP is barely documented and I did not see how the SPYNX project could profit from it.
Also I increased the Size of the Capture FIFO from 4096 to 8192 to be able to capture more data with SPYNX. I added the filtered stator currents to the capture signals and changed the order of the capture signals.
I changed the LED LD0 to indicate the switch SW0 position and LD1, LD2 and LD3 to indicate the encoder signals.

Adaption of the AXI4-Stream Monitor IP
The AXI4-Stream Monitor IP shown in Figure 9 acts as a multiplexer that can have up to 10 AXIS inputs. One of those inputs can be selected and its data is passed on to the capture interface. The IP is written in VHDL and does not participate in the input bus communication but is only monitoring the signals. Originally it checks only the *TVALID* signal of the monitored AXIS-bus. As soon as *TVALID* is set to '1' the monitor IP passes the data on to the capture interface. The Problem is, that often *TVALID* is high all the time since AXIS transfers also require the receiving slave to set the signal *TREADY* to '1'. So, if the slave is not ready yet but the master has already new data available, *TVALID* will be '1' for a longer period, often even constantly. So, if the Monitor IP is monitoring such a bus connection, the identical data would it sent to the capture unit multiple times with a too high rate. Consequently, most signals cannot be monitored correctly with the original Monitor IP.
To ensure that only data that was actually transferred on the monitored bus is passed on, I had to change the VHDL-code of the Monitor IP. Now only if both *TVALID* and *TREADY* signal are '1' data is sent to the capture unit.

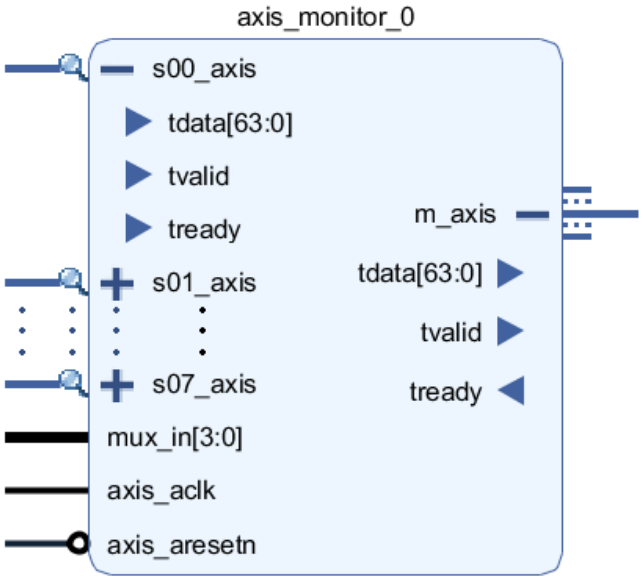


Figure 9 The AXI4-Stream Monitor IP. With the signal mux_in the source to be passed on to the capture unit can be selected. Note that the bus signals on the left hand side are all inputs since the IP is only monitoring the bus but not taking part in the communication.

3.2 Changes to the SPYN-Library

I added a new *init_mode* besides the existing modes *reset_mode*, *rpm_mode* and *torque_mode*. In this mode the motor turns slowly backwards in order to initialize the encoder IP. This encoder IP only outputs the correct angle once the encoder is turned backwards over the zero position. The encoder value gets always simply set to the value 0. So this will only work if the encoder is aligned with phase A (should always be the case with EDPS). If the encoder is aligned differently the values of the Register 10 "*SHIFT*" need to be changed in the SPYNX library file *constants.py* to match the orientation of the motor.

In contrast to the original SPYN-project that uses the overlay of the SDSoC version, I used the HLS version of EDDP in order to make use of the decimation feature. However, the Registers 13 and 14 are used differently in the HLS version. In the SDSoC version they are used as filter coefficients for the FOC Filter IP that is filtering the measured values. In HLS-EDDP those filter coefficients are constants and register 13 is used as the decimation value. I therefore renamed this register to *DECTIMATION*. Register 14 was designed as a Capture trigger, but this feature is not fully implemented in the released HLS version of EDDP. Therefore, register 14 can be used for the MPC control and has been named *CONTROL_MPC.*

3.3 Changes to the PYNQ-SPYN-Notebook

When looking at the acceleration of the motor I realized that the current must be much higher than calculated with the conversion factor of *0.00039A* (or *2564.1/A*) given in the EDDP. Before me others have already wondered why the motor gets so hot at apparently low currents. So we measured the currents with a current probe and an oscilloscope in a static setup and came to the conclusion that the conversion factor for the int16 value to ampere must to be around *0.0028A* (or *355.31/A*) which is over seven times more than stated in EDDP.
Similarly the given voltage conversion factor *0.0003662V* does not make sense. This is not very important because all voltage signals are exclusively created and handled by the FOC-PWM. Nevertheless I adjusted the factor to *0.0002113V* since the signals can also be captured and plotted. Now all the voltage signals stay in the reasonable range between -12V and 12V. Both the current and the voltage scaling factor just had to be adapted in the SPYN notebook since this is the only place where they are applied in SPYN.

In the first cell of the notebook the call *overlay.download()* was redundant and could be removed since the overlay is automatically downloaded with the O*verlay()* command. I inserted a cell into the notebook where I run the new *init_mode* for 4 seconds in order to initialize the encoder IP. The motor control interface has also been reworked to control both the FOC and the MPC (see Figure 5).
For the capture process I decreased the number of recordable data packages (of 32 samples each) from 1000 to a user controlled range between 1 to 256 since the capture FIFO in the overlay is 8192 (= 256*32) samples wide.
The AXI Data Capture IP is controlled by capture registers. The two least significant bits of the capture register 0 are directly controlling the Capture IP. They need to be set to '0' after each capture to reset the IP and make it ready again. This is why I needed to add the line *write_capturereg(0,0)* at the end of the capture function.
Since I added the decimation feature of EDDP to SPYNX, the time to fill the capture FIFO can now be much higher. This led to aliasing effects when using high

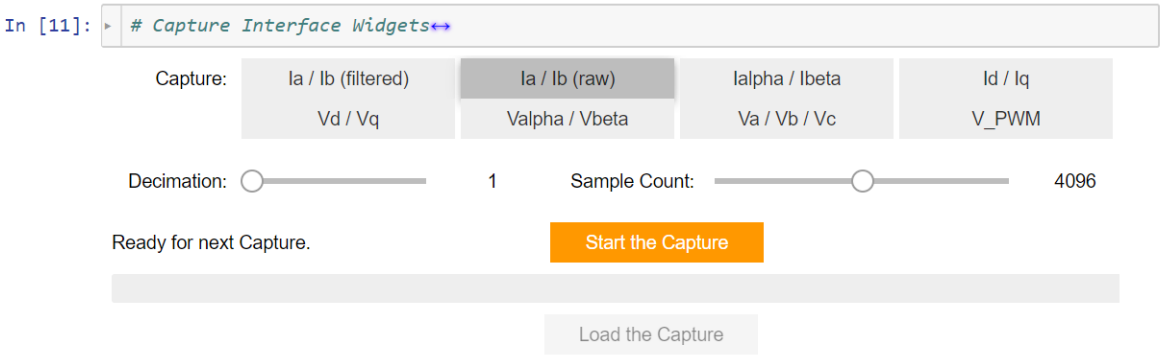


Figure 10 Graphical capture interface with buttons, sliders and a progress bar.

decimation rates, because the capture FIFO was read out faster by the Capture IP than it was filled with new data. Therefore, I had to add a wait statement to the capture cell in dependence of the chosen decimation rate. So now the bit to fill the FIFO is set first with *write_capturereg(0,2)*. Only when the FIFO had time to fill with data, the LSB is set with *write_capturereg(0,3)* to hold the data.
To make the capture process more user friendly, I created a graphical interface that can be seen in Figure 10. It lets the user capture data more intuitively. With toggle buttons the capture source can be chosen and with sliders both the decimation rate and the number of data samples can be selected. A progress bar depicts the progress of the capture process. Depending on the selected capture source, the captured signals are automatically scaled and labels are assigned. In this way, any plots that are being created can be labelled accordingly.

4 Implementation of the Model Predictive Controller

4.1 Preliminary Considerations

The MPC is supposed to control the reference motor stator currents i_d and i_q to given set points. For this purpose the control board can operate the inverter switches of the EDPS. To evaluate all possible switch positions and to find the optimal one the MPC needs to contain a mathematical model of the system.

As a starting point for the system model I used the system equations for the PMSM from the lecture *"Drive Control for Electric Vehicles"* held by Prof. Dr.-Ing. Ralph Kennel and Florian Bauer at TUM. In the datasheet of the reference PMSM only the line-to-line inductance is given. From a few simple step response measurements L_d and L_q seem identical or at least very similar. If we assume $L_d = L_q$, the motor does not produce any reluctance moment and the system equations simplify to Equation 1.

$$\frac{d}{dt}\begin{bmatrix} i_d \\ i_q \\ \omega_m \end{bmatrix} = \begin{bmatrix} -\frac{R}{L} & 0 & 0 \\ 0 & -\frac{R}{L} & -p\frac{\Psi_{PM}}{L} \\ 0 & \frac{3}{2}p\frac{\Psi_{PM}}{J} & -\frac{B}{J} \end{bmatrix}\begin{bmatrix} i_d \\ i_q \\ \omega_m \end{bmatrix} + \begin{bmatrix} p\,\omega_m i_q \\ -p\,\omega_m i_q \\ \frac{-M_L}{J} \end{bmatrix} + \begin{bmatrix} \frac{1}{L} & 0 \\ 0 & \frac{1}{L} \\ 0 & 0 \end{bmatrix}\begin{bmatrix} u_d \\ u_q \end{bmatrix}$$

Equation 1 System equations for the PMSM if $L_d = L_q$ and therefore no reluctance moment is exerted

Since the MPC is supposed to run with a frequency above 100 kHz the mechanical speed ω_m can be regarded as constant over the prediction horizon. Therefore, the system equations can be linearized and simplified to Equation 2.

$$\frac{d}{dt}\begin{bmatrix} i_d \\ i_q \end{bmatrix} = \begin{bmatrix} -\frac{R}{L} & p\,\omega_m \\ -p\,\omega_m & -\frac{R}{L} \end{bmatrix}\begin{bmatrix} i_d \\ i_q \end{bmatrix} + \begin{bmatrix} 0 \\ -p\frac{\Psi_{PM}}{L}\omega_m \end{bmatrix} + \begin{bmatrix} \frac{1}{L} & 0 \\ 0 & \frac{1}{L} \end{bmatrix}\begin{bmatrix} u_d \\ u_q \end{bmatrix}$$

Equation 2 Linearized system equations with the assumption $\omega_m = const.$ over the prediction horizon

To find the optimal switch position a cost function has to be evaluated for all possible k switch combinations. For the 2-level inverter with 3 phases used in EDDP k is 8. I opted for a classic cost function that can be seen in Equation 3. The differences between the current set points and the predicted currents are squared. The term $\|\Delta u(k)\|_1$ represents the switching effort. For every changed phase-switch it is increased by 1. So for the given setup it can be in the range from 0 to 3. The coefficient λ_u weights the switching effort. With $\lambda_u = 0$ only the current error is taken into consideration. The higher the value of λ_u the more costly is a phase switch. If λ_u is raised over a certain point the MPC will not switch at all. The resulting $J(k)$ with the lowest value has to be chosen and the switch position is set to the respective k.

$$J(k) = (i_d(k) - i_d^*)^2 + (i_q(k) - i_q^*)^2 + \lambda_u \|\Delta u(k)\|_1$$

Equation 3 Cost function for the FCS-MPC algorithm *[6]*.

Only in an ideal system without any measurement, computation, communication and actuation delays can the MPC be calculated directly based on the measurements. For the given SPYN setup however, those latencies have to be taken into account and compensated. The delay compensation is very well described in Chapter 4.2.8 of [6].

First of all the execution delay has to be compensated. If an MPC run starts at *t = k* with input data sampled at *t = k*, a new optimized switch position can only be applied after the computation has finished at *t = k+1*. So if this delay is ignored, the algorithm will always set the gates based on obsolete values.

As illustrated in Figure 11, the current change Δi between *t = k* and *t = k+1* is in fact already determined by the previously computed and applied switch position. The MPC started at *t = k* can only control the currents after *t = k+1*. To compensate for this delay we need to store the prediction of the previous control run. Thus, we can use the current measurements at *t = k* as a base and add the previous current prediction to it. If the MPC is well parameterized this will give us a good estimation of the currents at *t = k+1*. With those estimated currents the MPC starting at *t = k* can be run to choose the best switch position for *t = k+1*. In short, the control run starting at *t = k* does not predict the currents between *t = k* and *k+1* but between *t = k+1* and *k+2* instead.

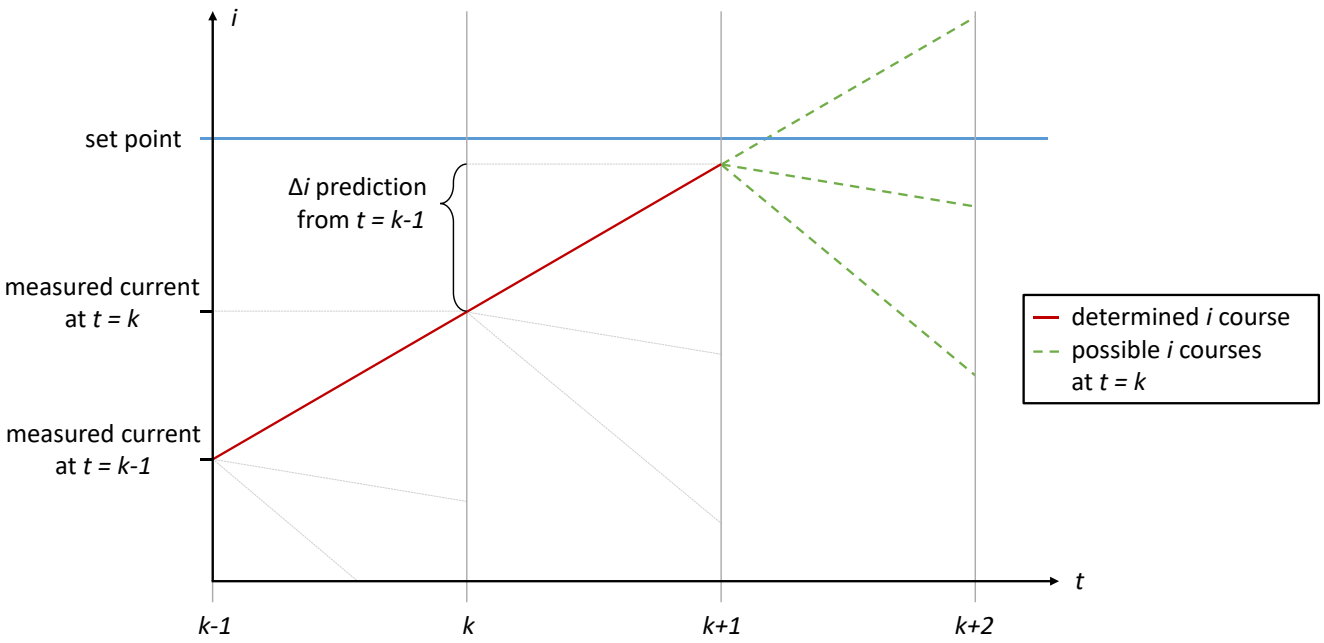
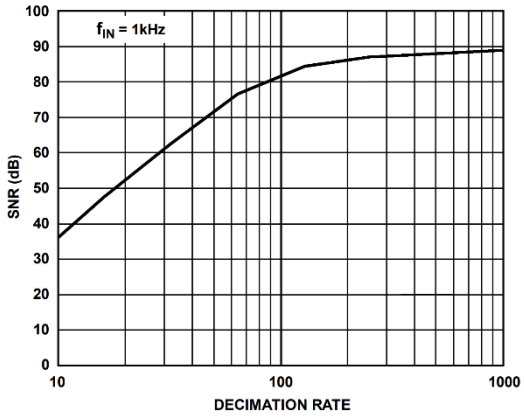


Figure 11 Visualization of the simplified control problem with computation delay compensation. At $t = k$ the current is measured. Only at $t = k+1$ the MPC can set a new optimized switch position. With the prediction by the MPC run started at $t = k-1$ this delay can be compensated.

When measuring the step response of the current signals in EDDP I found that the measurement delay is way too high and that it has to be compensated as well. The EDDP SINC3 ADC Filter IP by Trenz Electronic uses a decimation rate of 128. As can be seen in Figure 12 this leads to a filter response of 40.9 kHz. The MPC was supposed to run with at least 100 kHz. In order to make the measurement faster I reduced the decimation rate to 32. This reduced the filter's response time significantly but on the other hand the noise was increased a lot. Also, the EDDP's FOC is dependent and synchronized with the ADC signals. Therefore, the FOC is now running with a four times higher rate (sample rate of FOC is now 1.6us instead of 6.4us).
Even with the accelerated SINC3-filter it still takes about 10-15 ns between a switch change and an according response of ADC's current signals. In addition to the remaining delay of the SINC3 filter, this is also due to the further FOC Filter IP that filters the measurements with a two-stage infinite impulse response filter.



Decimation Ratio (DR)	Throughput Rate (kHz)	Output Data Size (Bits)	Filter Response (kHz)
32	625	15	163.7
64	312.5	18	81.8
128	156.2	21	40.9
256	78.1	24	20.4
512	39.1	27	10.2

Figure 12 Trade-off between filter response time and noise. The table on the right hand side refers to the SINC3-filter as it is implemented in EDDP and SPYN [7]

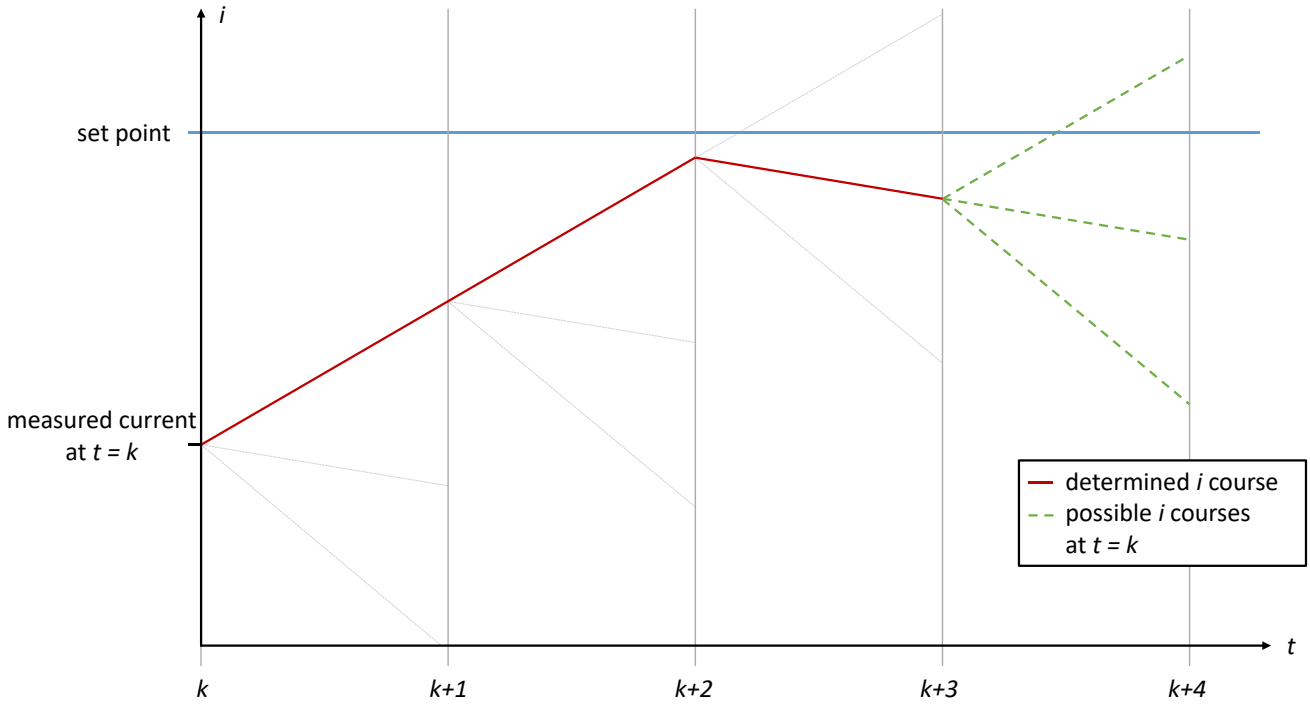


Figure 13 Visualization of the simplified control problem. At $t = k$ the current is measured. At $t = k+1$ the MPC can set the gates. But due to the slow measurement the impact of the new gates will only be visible in the current signals after two more sampling periods.

This remaining delay can be handled and compensated identically to the computation delay. The MPC is executed around every 6 ns, so to control the currents correctly another two stages of delay compensation had to be added. Figure 13 shows the situation for an MPC run that is started at $t = k$. Due to the computation delay the gate signals can be set at $t = k+1$. But it will take approx. two more time steps until the currents change accordingly. So the MPC can analogously to the computation delay use the previous 3 current predictions and the latest measurements to estimate the currents at $t = k+3$. Using this estimation the MPC can control the currents between $t = k+3$ and $k+4$.

4.2 Code Implementation

The code is commented extensively and can be found in the appendix (A3 Complete FCS-MPC Source Code). It was written in C and later synthesized to hardware description language using the Tool Vivado HLS.
Most of the required signals are already available in the EDDP Project. Only u_d, u_q and some control signals had to be created. I decided to transfer most of the constants as function arguments to the MPC so it would be possible to change those values online without a need to re-synthesize. Those arguments become interfaces through C synthesis. Similar to fixed-point numbers, the current values are subject to a scaling factor of *355.311* in order to handle them as 16-bit integers.

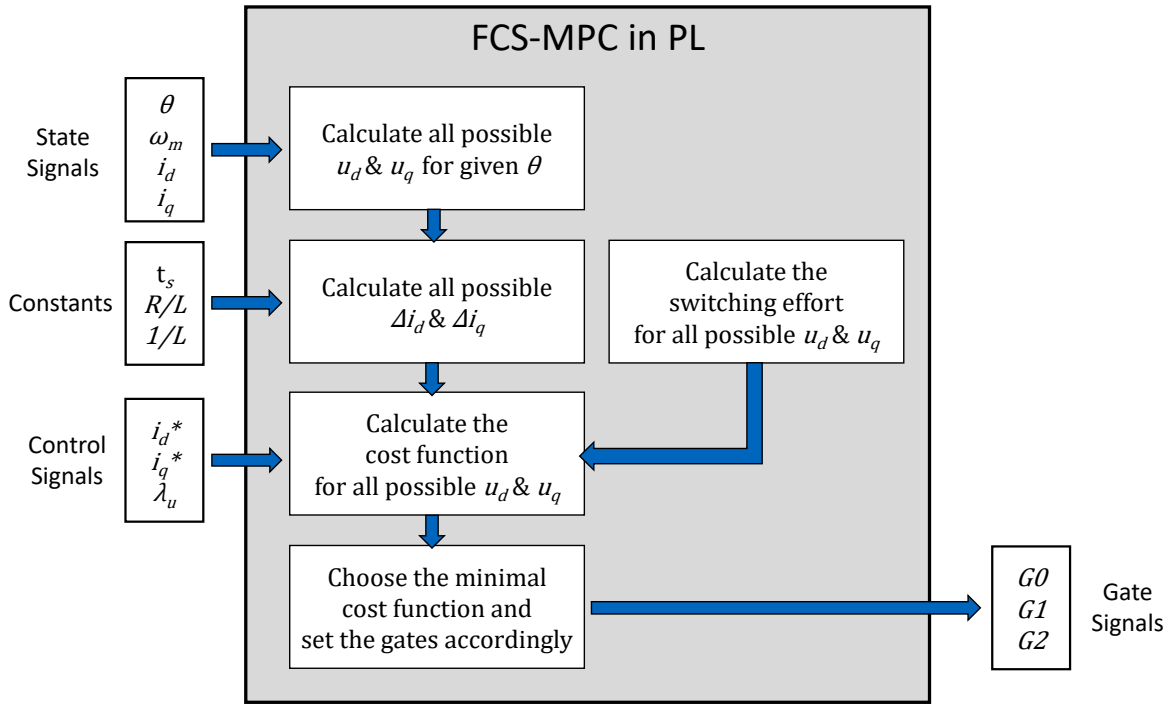


Figure 14 Simplified flowchart of the MPC algorithm.

Figure 14 shows a Flowchart of the MPC implementation with all the important algorithm steps and interfaces. On the left hand side all inputs are depicted. They grey area represents the FPGA's PL. On the right hand side the gate signals are the output of the MPC.

u_d and u_q have to be calculated within the MPC for all $k = 8$ switch positions using the Park and Clarke transformations. The Park transformation requires sine and cosine computations. Since an online calculation of trigonometric functions requires a big amount of resources I created a lookup table (LUT) for all 1000 encoder steps instead. To avoid float numbers, I additionally applied the current scaling factor. In this way the LUT could be stored as 16-bit integer with sufficient precision. For the cosine calculation I use the same sine LUT with an offset of 125 encoder steps what corresponds to an electrical angle of $\pi/2$. The resulting LUT is depicted in Figure 15.

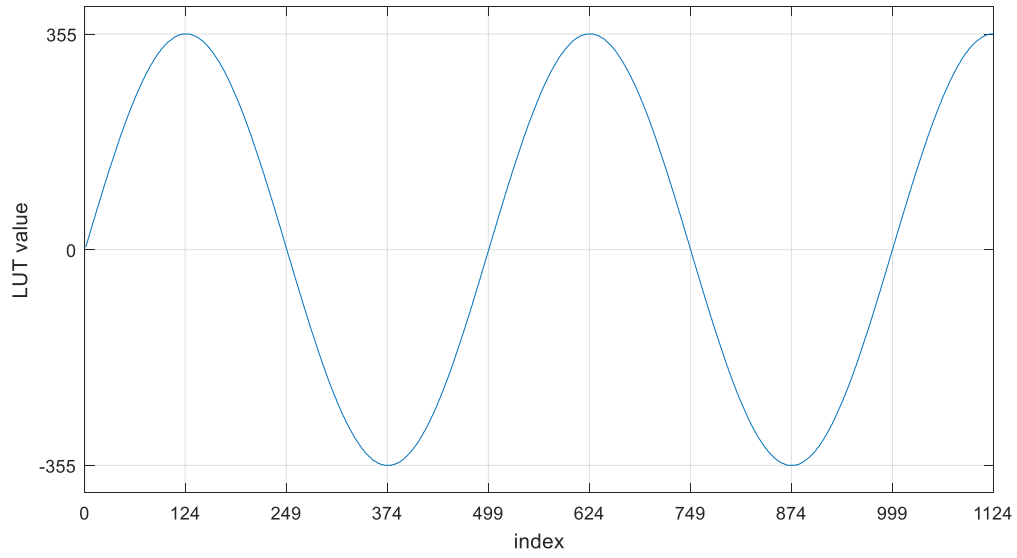
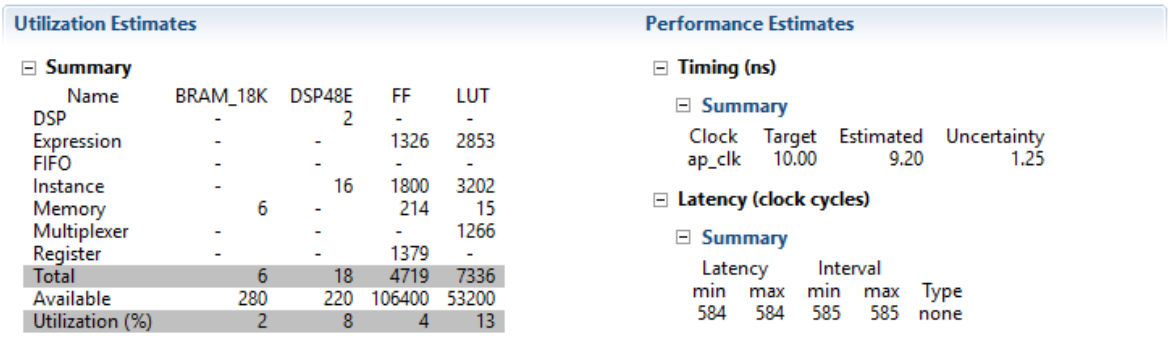


Figure 15 Visualization of the sine lookup table with sine values for the 1000 encoder steps and additional 125 values in order to also use it as a cosine LUT.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	2	-	-
Expression	-	-	1326	2853
FIFO	-	-	-	-
Instance	-	16	1800	3202
Memory	6	-	214	15
Multiplexer	-	-	-	1266
Register	-	-	1379	-
Total	6	18	4719	7336
Available	280	220	106400	53200
Utilization (%)	2	8	4	13

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.20	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
584	584	585	585	none

Figure 16 Result of the C synthesis of the FCS-MPC code. The Code can be run with $1 / (585 * 10\ ns) = 170.94\ kHz$ and uses only a fraction of the resources.

To make the code more efficient I performed calculations with integer numbers wherever possible. Also, I did not divide the current values down to their real value, but I worked with the scaled 16-bit integer ADC signals instead. I tested the code for a variety of scenarios and single precision float turned out to be accurate enough for a single prediction step in comparison to double. This way a big amount of resources can be saved.
Figure 16 shows the final C synthesis results. When analyzing the code, it becomes obvious that the efficiency could be further increased. But with 5.85µs the resulting PL latency is already low enough, and the resource utilization is only a fraction of the available resources. So, to keep the code readable I did not optimize the code further. The complete MPC code can be found in the appendix in Code Excerpt 1 andCode Excerpt 2.

The code was then exported as IP and integrated in the EDDP Vivado Project. Here the IP block was connected to all the required signals. It is free running without any trigger. Every 5.85µs its outputs are set and the inputs are read. So it is not running synchronously to the ADC and FOC sample rate of 1.6µs. This asynchronous design leads to variations in data age. For some MPC runs the input data is more current than for other runs. This is not a big problem because this 1.6µs timing variation is small compared to the sum of the other delays of 15-20µs.
For the output Gate signals the MPC IP has valid signals that indicate when the new signals are available. One of those valid signals is used as a trigger for all the following plots that have a sampling rate of 5.85µs. With the control boards switch SW0 a multiplexer can be controlled to change between FOC and MPC gate signals.

4.3 Simulation

In order to be able to test, verify and optimize the code faster I created a test bench model in Matlab Simulink. I created a continuous model of the reference PMSM in d/q dimensions and tried to tweak the parameters so that it behaves as similar as possible as the real motor. Some values like the mass moment of inertia of the dummy load had to be approximated. The noise and delay of the measured signals have also been modeled to recreate the control problem for the MPC as realistic as possible. A big advantage of the Simulation is, that all signals are accessible in arbitrary time resolution.
The MPC code is implemented as a user defined Matlab Function that calls the MPC C-function (see Appendix). Only the interface datatypes of the gates GH and GL must be changed from *ap_int<3>* to *uint8_t* since Matlab does not know this Xilinx data type. Also, in the header file the Xilinx libraries have to be commented out.

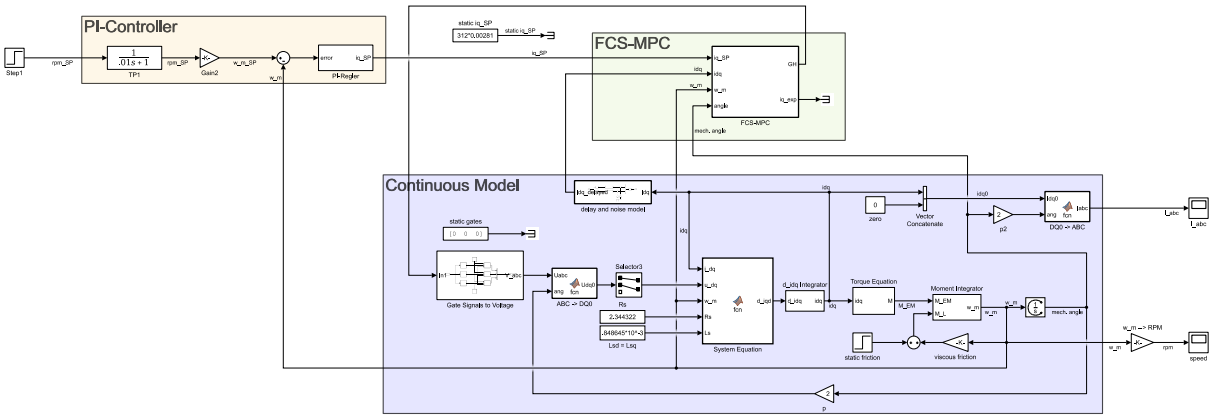


Figure 17 Test bench model in Simulink. Only the green box contains the actual MPC. The purple box models the motor's behavior with a continuous model. In the orange box a preliminary speed PI-controller is realized.

Apart from that, the identical code without modifications can be run in the simulation and also be synthesized to hardware description language by Vivado HLS.
The Simulink model could in the future also be used to design and test an outer speed PI controller, as the MPC is only fed with fixed current set points i_d^* and i_q^* at the moment. The model already contains a rudimentary PI speed controller, but it has not been tuned and tested properly.

5 Evaluation

Thanks to the above-mentioned optimizations the MPC can be executed with a frequency of over 170 kHz. Interestingly, one can clearly hear the influence of λ_u when running the MPC. With increasing λ_u the switching frequencies drop into the audible range and a high squeak becomes louder and louder. In this report I used exclusively λ_u values of 0 and 0.0079. The value of 0.0079 was chosen because in the code λ_u is subject to the squared current conversion factor. Therefore the integer value of 1000 in the code corresponds to $\lambda_u = 0.0079$ $(= 1000 \,/\, 355.311^2)$. Of course other values can be chosen. Values over 10000 ($\lambda_u > 0.07921$) do not seem reasonable for the EDPS setup.

With $\lambda_u = 0$, approximately 4000 rpm and rated current set point the performance of the MPC is similar to the FOC-PWM. The MPC's total harmonic distortion (THD) is a little lower and it also switches a little less often than the PWM at this operating point. In Figure 18,Figure 19 andFigure 20 evaluation plots for this operating point are shown. In all three cases the reference i_q has been set to the rated current of 880mA and the motor was rotating with similar speeds. The FOC-PWM was already starting to weaken the field and therefore decreasing i_d from 0 to negative values. This is why there is a slight shift between the dashed ideal current line and the real stator currents. As can been seen in Figure 18 in the current spectrum, the FOC-PWM produces a switching frequency pattern around 20 kHz. The MPC on the other hand does not produce such delimited switching patterns since there is no fixed switching frequency. But instead the whole frequency spectrum is raised in comparison to the FOC. This is clearly visible in Figure 20 with $\lambda_u = 0.0079$.
In the appendix the Figure 25,Figure 26 andFigure 27 show more detailed plots for a better visual comparison. The periodic switching frequency of the PWM leads to a clearly visible ripple on the FOC's currents. In contrast, the two MPC plots show a more uneven current course.

At lower current set points and motor speeds the MPC's switching frequency can rise up to approx. 45 kHz for $\lambda_u = 0$. For $\lambda_u = 0.0079$ the switching frequency can rise as well but there was no case observed where it exceeded 10 kHz. The Figure 22,Figure 23 andFigure 24 in the appendix illustrate the dynamic behavior for each of the three controls. In the beginning at *t = 0* the current set points i_d^* and i_q^* are set to 0, but at *t = 1ms* i_q^* is raised to rated current. In this dynamic case the MPC with $\lambda_u = 0$ switches more than twice as much as the FOC-PWM. For the MPC with $\lambda_u = 0.0079$ the switching frequency is still less than 30% of the PWM's frequency. The theoretical maximum switching frequency per device is 85.46 kHz the MPC.
If the inductances were bigger than the ones of the reference motor ($L_{line\_to\_line}$ 4.63mH [7]), the switching frequency of the MPC would be lower.

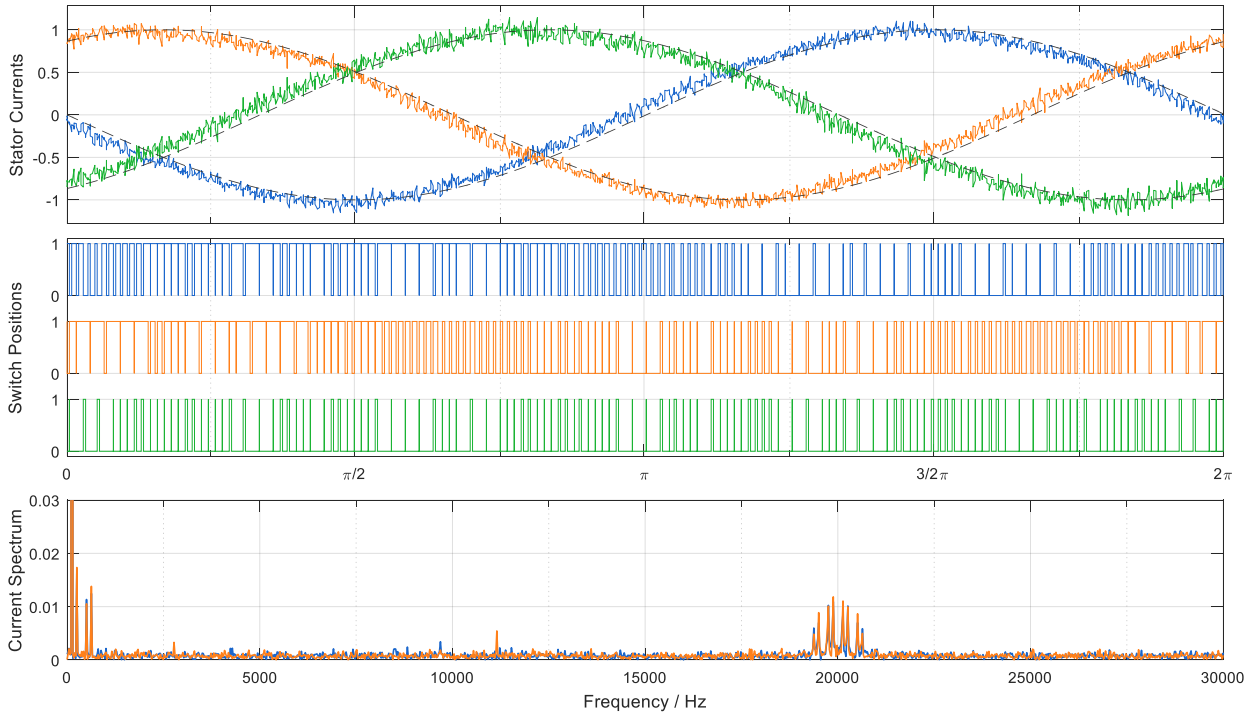


Figure 18 FOC-PWM sampled with T_s = 5.85µs, Current THD $I_{s,THD}$ = 6.32 % and the average switching frequency per device is 20 kHz

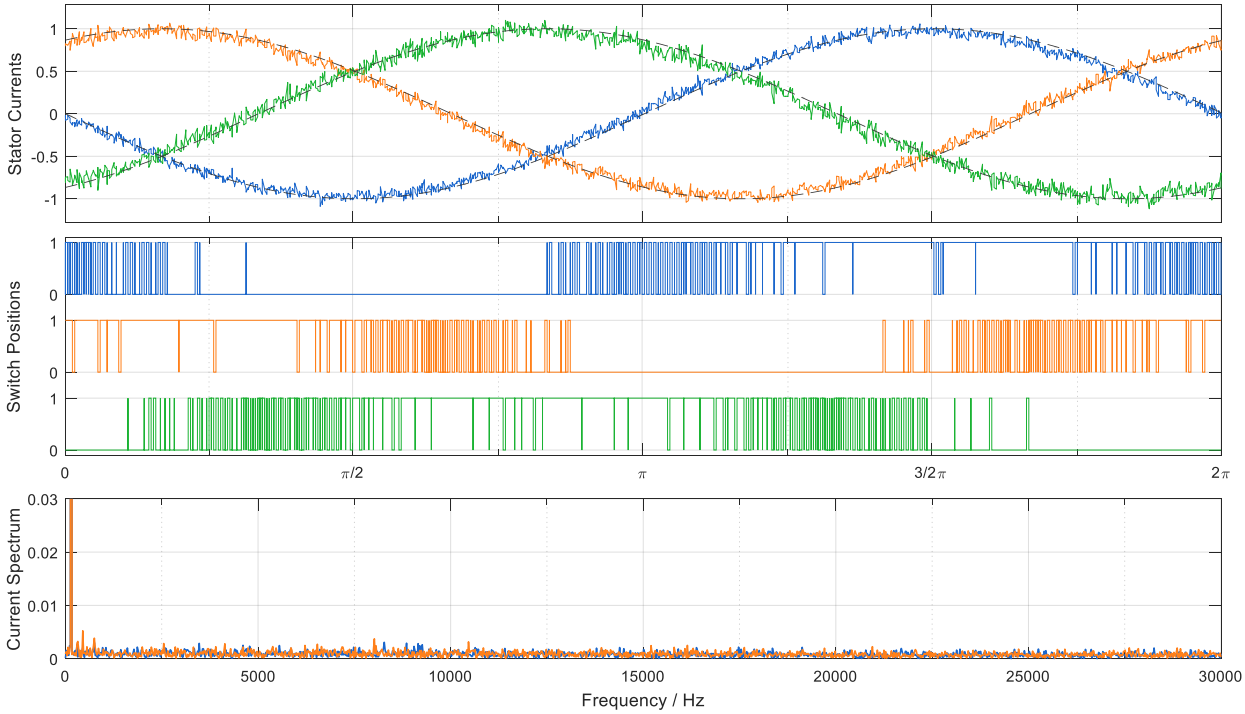


Figure 19 FCS-MPC with $\lambda_u = 0$, sampled with $T_s = 5.85\mu s$, Current THD $I_{s,THD} = 5.58$ % and the average switching frequency per device is 19.8 kHz

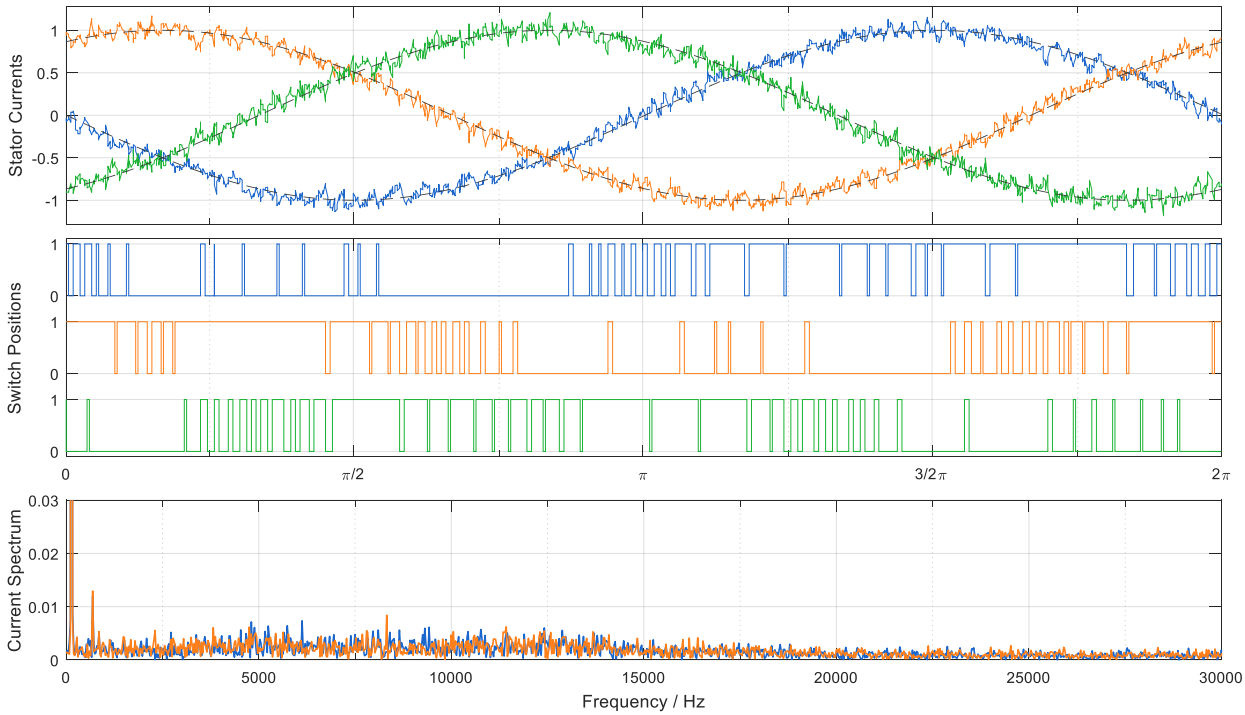


Figure 20 FCS-MPC with $\lambda_u = 0.0079$, sampled with $T_s = 5.85\mu s$, Current THD $I_{s,THD} = 8.08$ % and the average switching frequency per device is 5.67 kHz

6 Outlook

One problem of the MPC is that it is very susceptible to the relatively high noise level of the current signals. Even though the currents are being filtered before the MPC processes them, there still remain noise peaks of over 10 mA as can be seen in Figure 21. Therefore, the MPC does not perform well at very low current set points. But if the signals were filtered stronger, the signal delay would also be further increased. A simple delay compensation does not work anymore with strongly filtered signals. Ideally the MPC would model the delay and behavior of the filter. Then you could filter a lot more without sacrificing the MPC's excellent performance. This would lead to less noise induced switching and the MPC would probably clearly outperform the FOC.
Another improvement could probably be achieved by synchronizing the MPC to the ADC's sample rate. The measured data would have the same age for every MPC run and there would not be any timing variation in the current control loop.

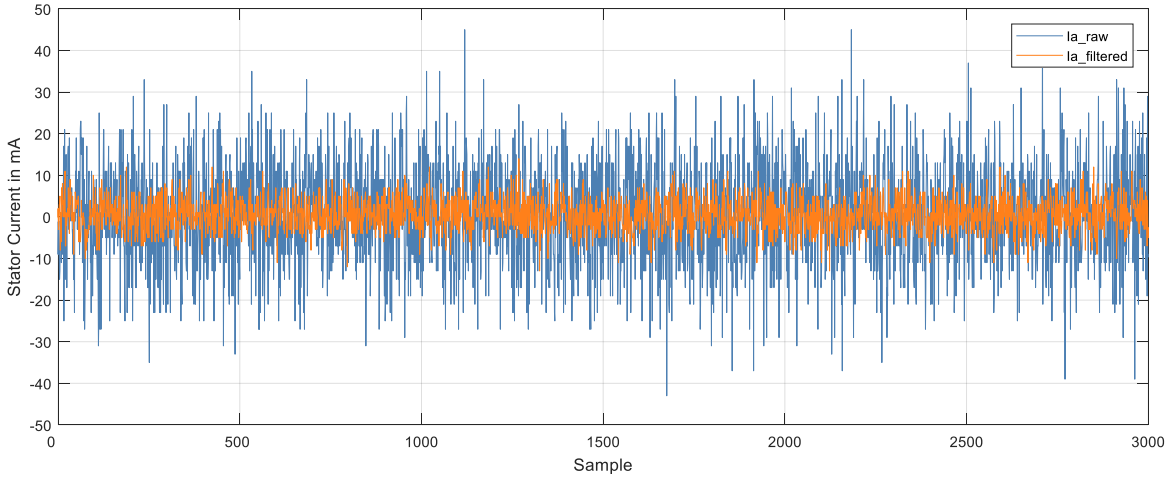


Figure 21 Recording of the noise sampled with T_s = 1.6µs. The actual stator currents are zero. The orange signal shows how the blue raw ADC signal looks like after the FOC filter IP has been applied.

7 References

[1] Xilinx Inc, "GitHub repository IIoT-SPYN," 2018. [Online]. Available: https://github.com/Xilinx/IIoT-SPYN. [Accessed 27 06 2019].

[2] Trenz Electronic GmbH, "GitHub repository IIoT-EDDP," 2017. [Online]. Available: https://github.com/Xilinx/IIoT-EDDP. [Accessed 27 06 2019].

[3] Xilinx Inc., "PYNQ: PYTHON PRODUCTIVITY FOR ZYNQ," 2018. [Online]. Available: http://www.pynq.io/. [Accessed 27 06 2019].

[4] Project Jupyter, "Project Jupyter - Home," 2019. [Online]. Available: https://jupyter.org/. [Accessed 27 06 2019].

[5] Xilinx Inc., "GitHub repository Python Productivity for ZYNQ," 2018. [Online]. Available: https://github.com/Xilinx/PYNQ. [Accessed 27 06 2019].

[6] T. Geyer, "Direct Model Predictive Control with Reference Tracking," in *Model Predictive Control of High Power Converters and Industrial Drives*, Hoboken, Wiley, 2016.

[7] Anaheim Automation Inc., "BLWR11 - Brushless DC Motors - BLWR111D-24V-10000," [Online]. Available: https://www.anaheimautomation.com/manuals/brushless/L010234%20-%20BLWR11%20Series%20Product%20Sheet.pdf. [Accessed 27 06 2019].

[8] Analog Devices Inc., "AD7403 (Rev. B) – 16-Bit, Isolated Sigma-Delta Modulator," 2014–2015. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/AD7403.pdf. [Accessed 27 06 2019].

8 Appendix

A1 List of Abbreviations and Parameters

ADC	Analog-to-digital converter. In EDDP a sigma delta modulator is used
EDDP	Electronic Drive Development Platform
EDPS	Electric Drive Power Stage
FCS-MPC	Finite Control Set Model Predictive Control
FIFO	First In, First Out Buffer
FOC	Field Oriented Control
FPGA	Field Programmable Gate Array
HLS	High Level Synthesis
ILA	Integrated Logic Analyzer (Vivado IP Core for debugging / monitoring)
PL	Programmable Logic
PMSM	Permanent-Magnet Synchronous Motor
PWM	Pulse Width Modulation
SDSoC	Software-Defines System-on-Chip
THD	Total Harmonic Distortion
VHDL	Very High Speed Integrated Circuit Hardware Description Language
λ_u	Parameter to control the switching cost (higher λu → less switching)
c_i	Current scaling factor int16 to ampere is 0.002814433
L	Inductance $L = L_d = L_q = ½ L_{line\_to\_line}$ (datasheet [7] $L_{line\_to\_line} = 0.845mH$)
GH	Inverted 3-bit gate control signal (all switches ‘1’ → GH = “000”)
GL	3-bit gate control signal (all switches ‘1’ → GL = “111”)
J	Moment of inertia of the motor and load (approximated 6 $kg*mm^2$)
p	Number of pole pairs is 2
R	Stator resistance of one phase (datasheet [7] $½ R_{line\_to\_line} = 2.315Ω$)
T_s	Sampling period of the MPC is 5.85 us
Ψ_PM	Flux linkage (measured and calculated from datasheet [7] 0.00535 Wb)
ω_m	Mechanical speed of the motor in rad/s

A2 Further Plots

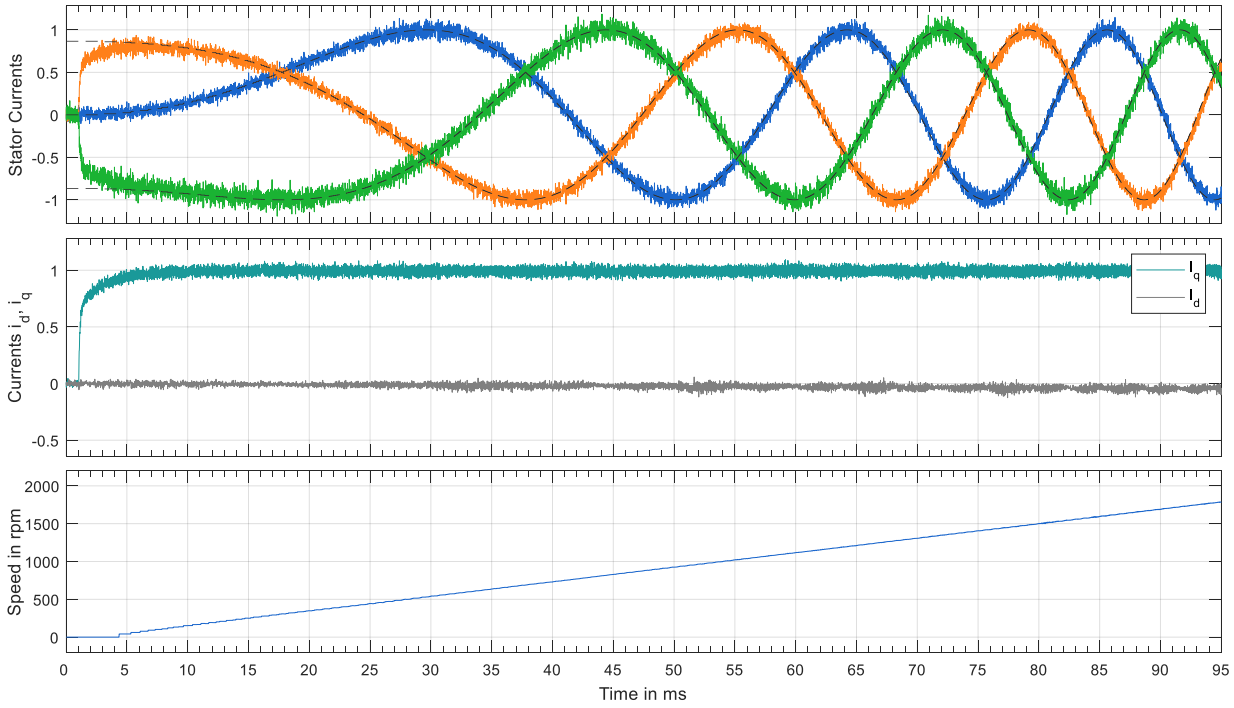


Figure 22 FOC-PWM sampled with T_s = 5.85µs. At t = 1ms the I_q set point was raised from 0 to rated current. The average switching frequency per device is 20 kHz.

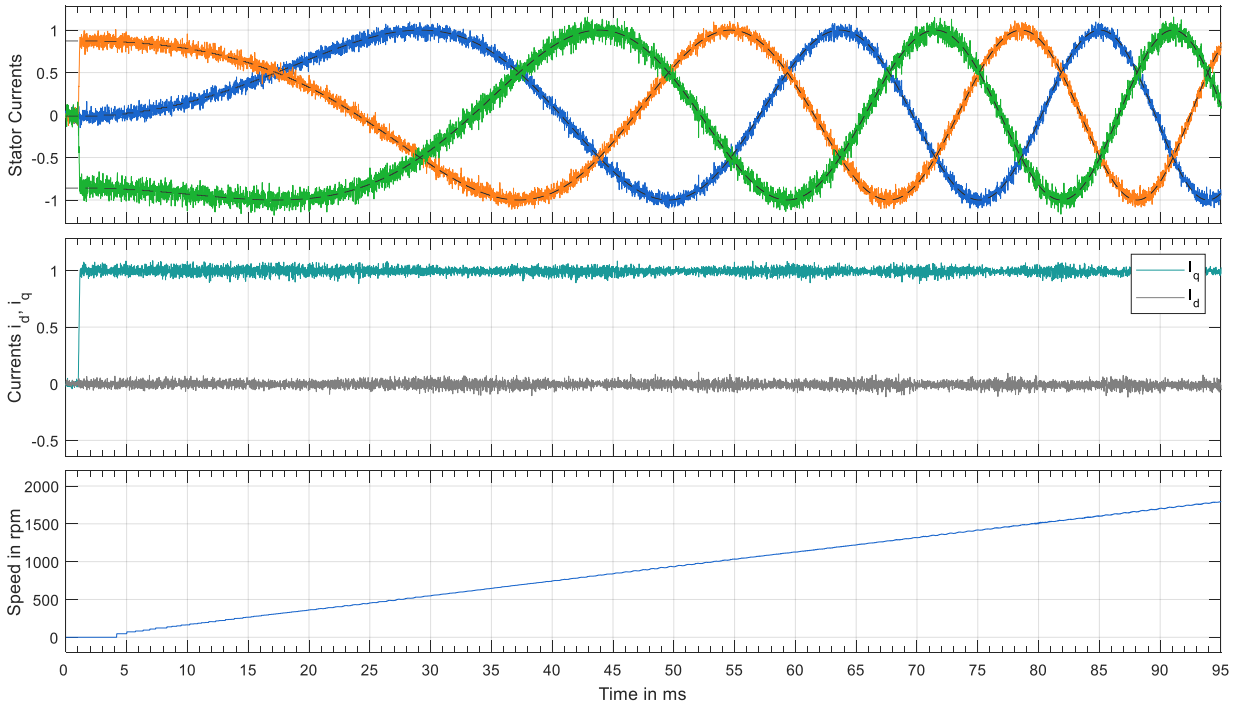


Figure 23 FCS- MPC sampled with T_s = 5.85µs and λ_u = 0. At t = 1ms the I_q set point was raised from 0 to rated current. The average switching frequency per device is 42.846 kHz.

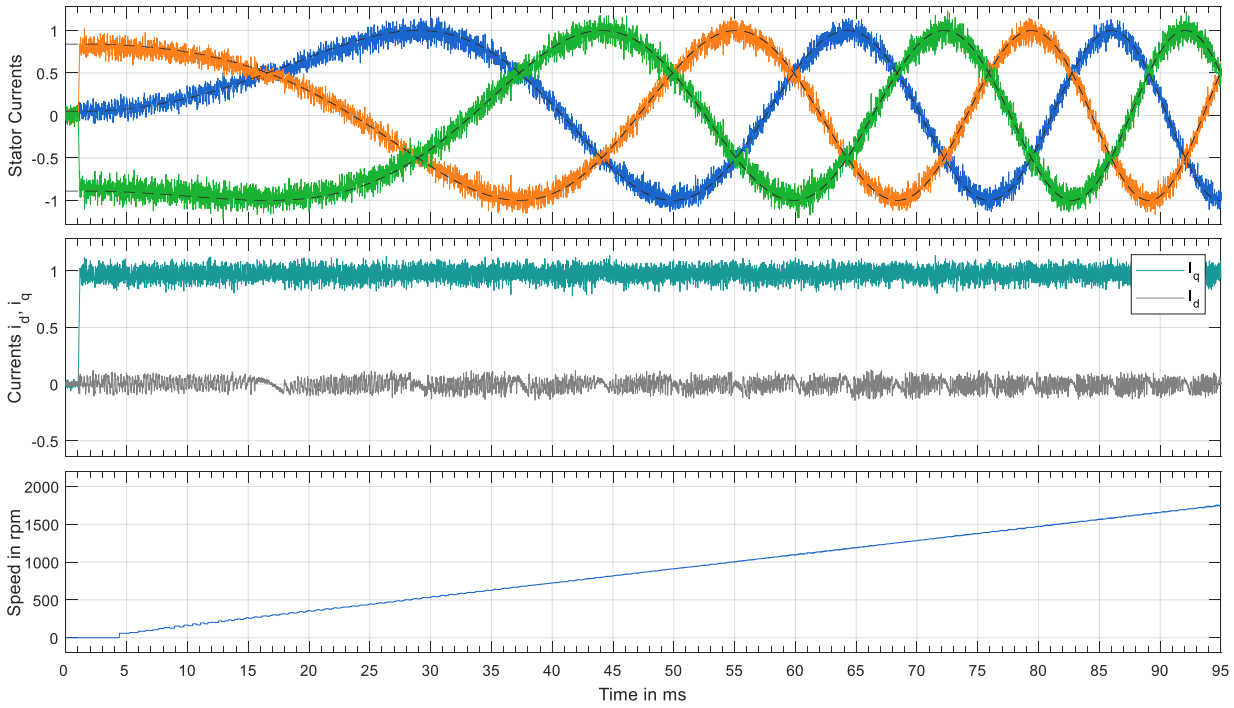


Figure 24 FCS- MPC sampled with T_s = 5.85µs and λ_u = 0.0079. At t = 1ms the I_q set point was raised from 0 to rated current. The average switching frequency per device is 5.725 kHz.

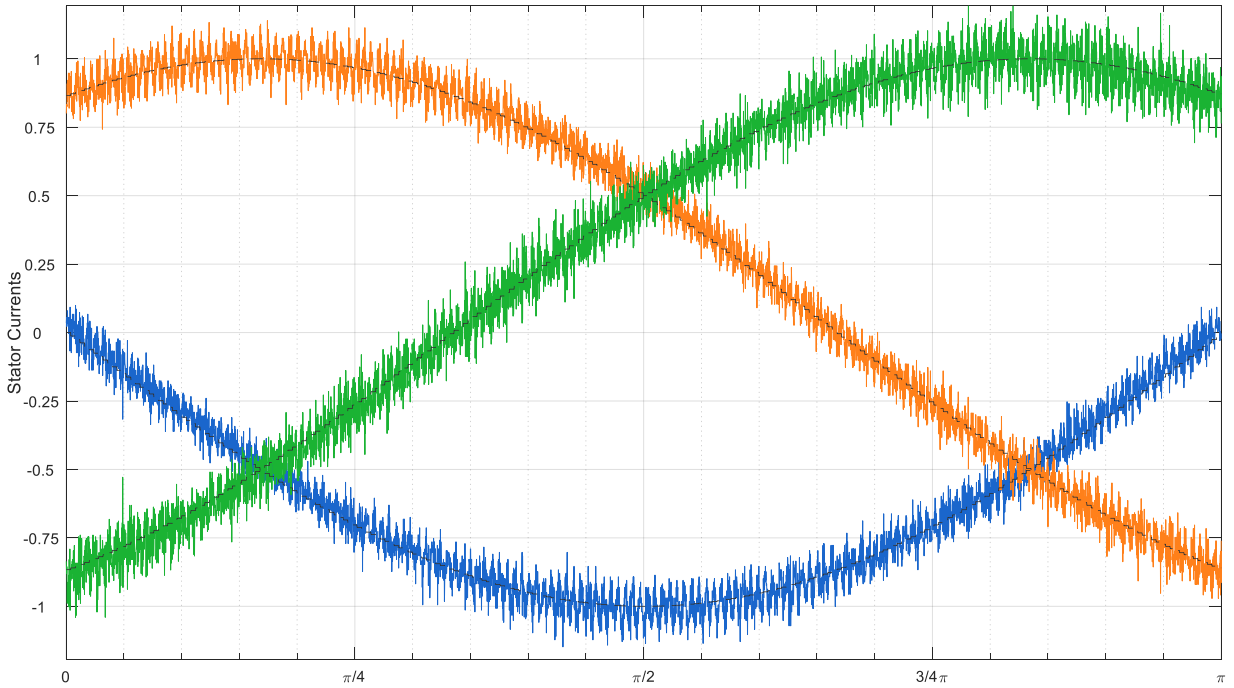


Figure 25 FOC-PWM sampled with T_s = 1.6µs. The I_q set point was set to rated current.

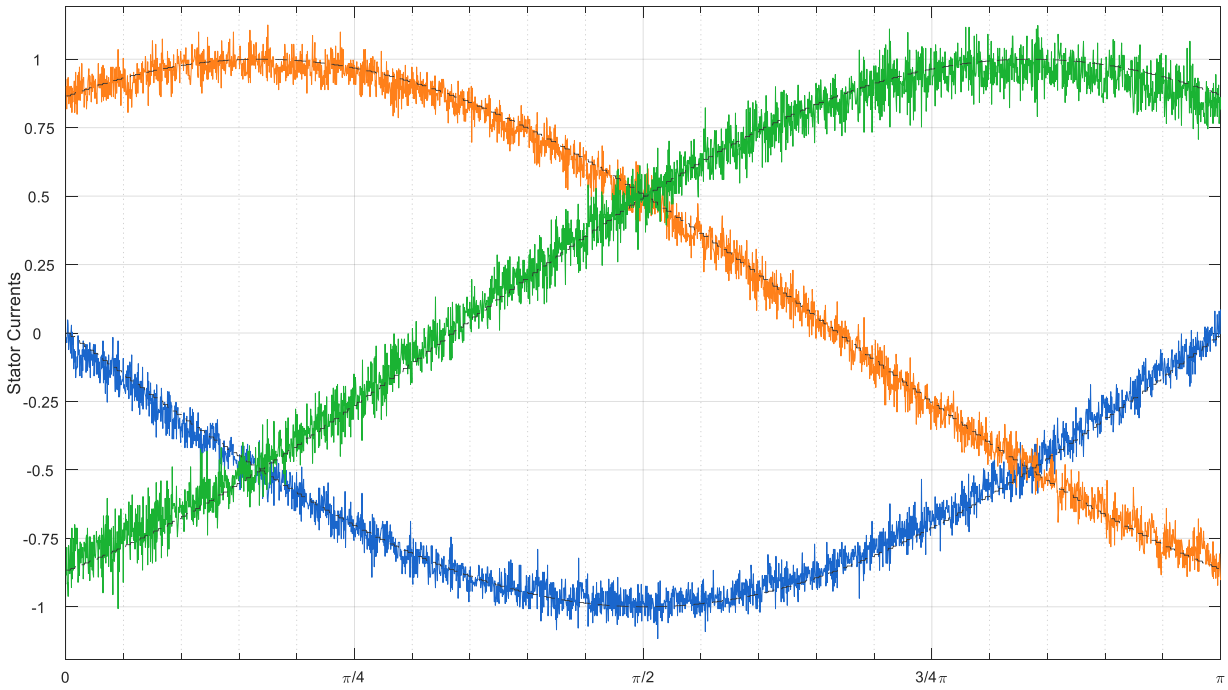


Figure 26 FCS-MPC sampled with T_s = 1.6µs and λ_u = 0. The I_q set point was set to rated current.

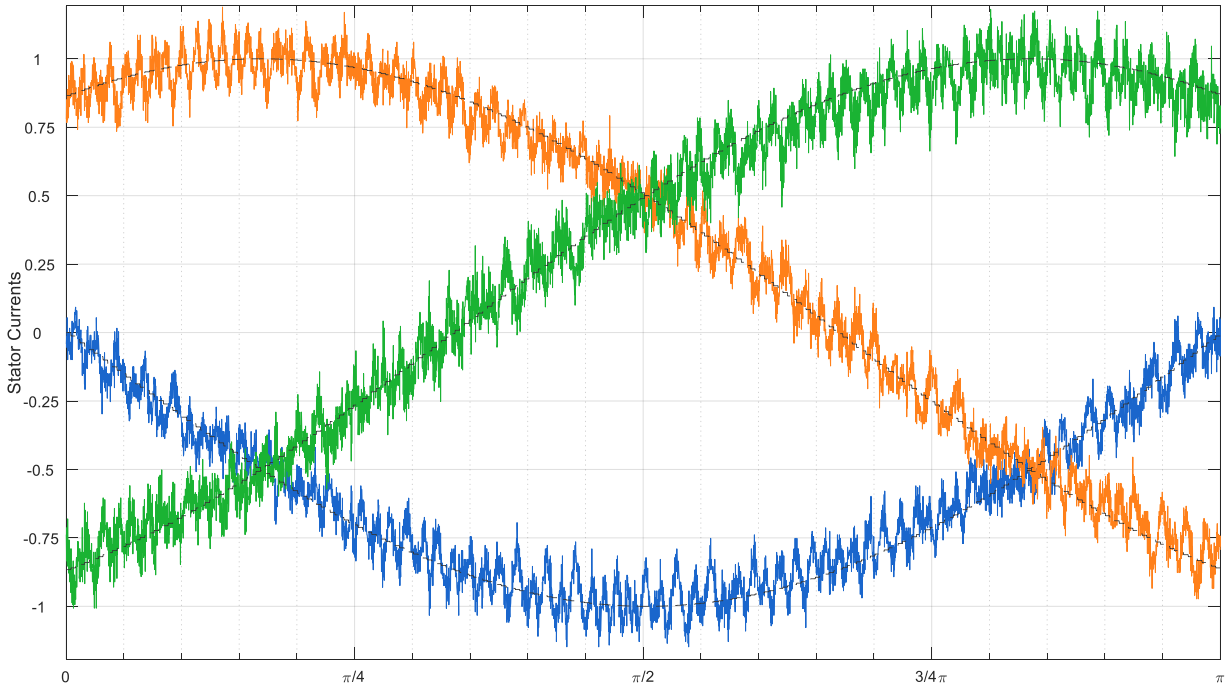


Figure 27 FCS-MPC sampled with T_s = 1.6µs and λ_u = 0.0079. The I_q set point was set to rated current.

A3 Complete FCS-MPC Source Code

```
#include <ap_cint.h>
#include <ap_int.h>
#include <stdint.h>
#include <cstdlib>
#include "sin_table.h"
#define RPM_TO_W_E_FACTOR   0.2094395102f    // w_e = 2*w_m = 2*2*pi * RPM/60 = 0.2094395102*RPM
#define ONE_OVER_SQRT3      0.57735026919f   // 1/sqrt(3) = 0.57735026919
#define INDUCTION_FACTOR    385              // w_e * Psi/L = 4*pi*RPM/60 * Psi/L/0.002814 = 469.133*RPM (Measured 385)

void FCSMPC(float   R_over_L,          // quotient of phase resistance over phase inductance [Ohm/H]
          float     one_over_L,        // inverse of the phase inductance [1/H]
          float     sampling_period,   // the sampling period of the synthesized FCSMPC Code [1]
          uint16_t  lm_over_c_i_sqr,   // lamda_u divided by the squared conversion factor c_i^2
          int16_t   angle,             // angle of the motor in encoder steps (0 ... 999) [1]
          int16_t   RPM,               // speed of the motor [rpm]
          int16_t   id_m,              // d component of the measured current (conversion factor c_i applied)
          int16_t   iq_m,              // q component of the measured current (conversion factor c_i applied)
          int16_t   id_SP,             // set point of the d component of the current (conversion c_i factor applied)
          int16_t   iq_SP,             // set point of the q component of the current (conversion c_i factor applied)
          ap_int<3> GH,                // inverted Gate signals for the high side (bit layout 0bCBA)
          ap_int<3> GL,                // inverted Gate signals for the low side (bit layout 0bCBA)
          int16_t   id_exp,            // Output for debug purpose - MPC prediction (conversion factor c_i applied)
          int16_t   iq_exp){           // Output for debug purpose - MPC prediction (conversion factor c_i applied)

          #pragma HLS INTERFACE ap_ctrl_none port=return
          #pragma HLS INTERFACE ap_vld port=GH
          #pragma HLS INTERFACE ap_vld port=GL
          #pragma HLS INTERFACE ap_none port=id_exp
          #pragma HLS INTERFACE ap_none port=iq_exp

    /* all switch positions: idx    0    1    2    3    4    5    6    7*/
    const uint8_t uk_pos[3][8] = {{ 0,   12,  0,   12,  0,   12,  0,   12},    // PHASE A
                                  { 0,   0,   12,  12,  0,   0,   12,  12},    // PHASE B
                                  { 0,   0,   0,   0,   12,  12,  12,  12}};   // PHASE C

    /* the switch position that was chosen at the last MPC run (initially 0) */
    static uint8_t uk_0 = 0;
    /* current predictions of the last three MPC runs */
    static int16_t  delta_id_pred_k_1 = 0,    // 1st delay compensation for the execution time of the MPC
                    delta_iq_pred_k_1 = 0,
                    delta_id_pred_k_2 = 0,    // 2nd delay compensation for delay in the measurement
                    delta_iq_pred_k_2 = 0,
                    delta_id_pred_k_3 = 0,    // 3rd delay compensation for delay in the measurement
                    delta_iq_pred_k_3 = 0;

    /* System Matrix of the PMSM */
    static float A_Mat[2][2];
    A_Mat[0][1] = RPM*RPM_TO_W_E_FACTOR;      // A =  [ -R/L    w_e  ]
    A_Mat[1][0] = -A_Mat[0][1];               //      [  w_e    -R/L ]
    A_Mat[0][0] =  A_Mat[1][1] = -R_over_L;   // with w_e = electrical speed of the EM in rad/sec

    /* If the EM turns faster than 2500 an encoder step has to be added (sample rate ~6us)
     * since the code is calculated for the situation 3 steps in the future */
    if (RPM > 2500)
        angle++;
    else if (RPM > 7500)
        angle += 2;
    /* sine and cosine are needed for the park transformation.
     * A LUT has been calculated offline for all encoder steps to increase performance.
     * The current scaling factor has already been applied to be able to work with integer */
    if (angle > 999)
        angle -= 1000;
    int16_t cost_over_c_i = sinus_table[angle+125];
    int16_t sint_over_c_i = sinus_table[angle];

    /* The last 3 delta_i predictions are summed up and the newest measurement is used as the base.
     * The result are the currents that are expected at t = k+3 (code started at t = k) */
    int16_t id_pred_k_3 = id_m + delta_id_pred_k_1 + delta_id_pred_k_2 + delta_id_pred_k_3;
    int16_t iq_pred_k_3 = iq_m + delta_iq_pred_k_1 + delta_iq_pred_k_2 + delta_iq_pred_k_3;
```

Code Excerpt 1 Part 1 of 2 of the FCS-MPC code

```
/*########## Calculate the Udq for all possible switch positions ##########*/
    /* only two offline calculated ualpha/beta values are needed (symmetrical voltage hexagon):
     * U_alpha/beta = [ 2/3*ua  -  ub/3    -   uc/3    ]
     *                [   0   + ub/sqrt(3) - uc/sqrt(3)]    */
    float ualpha[3], ubeta[3];
    ualpha[0] = 8;           // for switch position index 1
    ubeta[0]  = 0;           // for switch position index 1
    ualpha[1] = -4;          // for switch position index 2
    ubeta[1]  = 6.9282;      // for switch position index 2
    ualpha[2] = -ualpha[1];  // switch position 3 can be derived from 2
    ubeta[2]  =  ubeta[1];   // switch position 3 can be derived from 2

    /* three Udq values have to be calculated for the given angle:
     * U_d/q = [ cos(ang)*u_alpha + sin(ang)*i_beta ]
     *         [ cos(ang)*i_beta  - sin(ang)*u_alpha]     */
    float ud[8], uq[8];
    for (int idx = 1; idx <4; idx++){
        ud[idx] = cost_over_c_i * ualpha[idx-1] + sint_over_c_i * ubeta[idx-1];
        uq[idx] = cost_over_c_i * ubeta[idx-1]  - sint_over_c_i * ualpha[idx-1];
    }
    /* Udq 4 to 6 are just mirrored Udq 1 to 3 */
    ud[4] = -ud[3];
    uq[4] = -uq[3];
    ud[5] = -ud[2];
    uq[5] = -uq[2];
    ud[6] = -ud[1];
    uq[6] = -uq[1];
    /* ud = uq = 0 if all switch positions are identical */
    ud[0] = uq[0] = ud[7] = uq[7] = 0;

/*########## Calculate the current changes for all possible switch positions ##########*/
    int32_t iq_induction = -RPM*INDUCTION_FACTOR;    // iq induction = -w_e * Psi/L
    int32_t d_id[8], d_iq[8];                        // variables to store the di/dt values
    float delta_id[8], delta_iq[8];                  // variables to store the delta_i/t_sample
    /* Calculate the current change for all possible switch positions
     * di_dq/dt = [A00 A01]*[id] +     0     + 1/L*ud
     *            [A10 A11] [iq] + iq_induct + 1/L*uq */
    for (int idx = 0; idx < 8; idx++){
        d_id[idx] =   A_Mat[0][0]*id_pred_k_3
                    + A_Mat[0][1]*iq_pred_k_3
                    + one_over_L*ud[idx];
        d_iq[idx] =   A_Mat[1][0]*id_pred_k_3
                    + A_Mat[1][1]*iq_pred_k_3
                    + iq_induction
                    + one_over_L*uq[idx];
        /* Calculate the change of the current within one sampling period delta_i/t_sample */
        delta_id[idx] = d_id[idx]*sampling_period;
        delta_iq[idx] = d_iq[idx]*sampling_period;
    }
/*########## Calculate the switching effort for all possible switch positions ##########*/
    uint8_t sw_eff[8] = {0};
    for (int idx = 0; idx < 8; idx++)           // iterate through all switch positions
        for (int idx2 = 0; idx2 < 3; idx2++)    // iterate through all phase switches
            if (uk_pos[idx2][uk_0] != uk_pos[idx2][idx]) // if the phase switch is different from the previous...
                sw_eff[idx]++;                           // ... increase the switching effort counter for that index

/*########## Calculate the optimal switch position with the cost function ##########*/
    uint8_t uk_1_opt = 0;    // variable to find and store the best switch position index
    int32_t J[8];            // variable to store all cost function runs
    for (int idx = 0; idx < 8; idx++){
        J[idx] =    (id_pred_k_3+delta_id[idx]-id_SP)*(id_pred_k_3+delta_id[idx]-id_SP)
                  + (iq_pred_k_3+delta_iq[idx]-iq_SP)*(iq_pred_k_3+delta_iq[idx]-iq_SP)
                  +  sw_eff[idx]*lm_over_c_i_sqr;
        if (J[idx] < J[uk_1_opt])      // if the current cost function result is lower than the best so far...
            uk_1_opt = idx;            // ... save the current index as the new best index
    }

/*########## UPDATE THE STATIC SIGNALS FOR THE NEXT ROUND ##########*/
    delta_id_pred_k_1 = delta_id_pred_k_2;    // pull the previous predictions on step back
    delta_iq_pred_k_1 = delta_iq_pred_k_2;    // k+1 = k+2
    delta_id_pred_k_2 = delta_id_pred_k_3;    // k+2 = k+3
    delta_iq_pred_k_2 = delta_iq_pred_k_3;
    delta_id_pred_k_3 = delta_id[uk_1_opt];   // save the current prediction for t = k+3
    delta_iq_pred_k_3 = delta_iq[uk_1_opt];
    uk_0 = uk_1_opt;                          // save the optimal switch position as applied switch position

/*########## SET THE OUTPUTS TO THE CALCULATED VALUES        ##########*/
    /* The Gate Signals are inverted so e.g. in order to
     * turn only phase A on GH has to be 0b110 and GL 0b001*/
    *GL = uk_1_opt & 0b111;
    *GH = ~*GL;
    /* For debugging and optimization:
     * The predicted currents after two more sampling periods */
    *id_exp = id_pred_k_3;
    *iq_exp = iq_pred_k_3;
}
```

Code Excerpt 2 Part 2 of 2 of the FCS-MPC code