This article describes the detailed steps in building a model-based design (MBD) platform around an ARM®-based embedded motor control processor. It follows with examples of basic permanent magnet synchronous motor (PMSM) control algorithms initially deployed and the ease of extending the functionality to include multi-axis position control for automation systems.

System and circuit modeling has long been an important aspect of motor control system design. With the MBD approach, electrical, mechanical, and system-level models are used to evaluate design concepts before building and testing physical hardware. The latest simulation tools from MathWorks can model complete embedded control systems, including the electrical circuit and mechanical system domains. Meanwhile, embedded coding tools generate C code from control system models to enable direct deployment of control algorithms on embedded control platforms.

These tools enable a model-based design process where control algorithms can be designed and fully tested on a simulation platform before the final hardware test. The key to building a successful MBD platform is partitioning the system model and embedded software code. Once the MBD platform is tested using a known algorithm and system, new algorithms can be developed and safely tested on the simulation platform at the extreme limits of system operation.

**A Complete Design Flow**

MBD has been a discussion topic for decades, but has only in recent years evolved into a complete design flow—from model creation to complete implementation. MBD is a mathematical and visual method of addressing problems associated with designing complex embedded control systems.

Rather than using complex structures and extensive software codes, designers can use MBD to define models with advanced functional characteristics using continuous time and discrete time building blocks. These models, used with simulation tools, can lead to rapid prototyping, software testing, and hardware-in-the-loop (HIL) simulation.

Simulation enables specification discrepancies and modeling errors to be found immediately, rather than later in the design cycle. To optimize the overall code generation process, automatic code generation can be added to reduce any manual implementation steps and to help further reduce the overall time to market. All in all, the MBD approach allows a designer to expand from more classical design schemes and move directly from model creation to simulation, generation, HIL, and testing in a controlled fashion that allows changes in system settings without a complete redesign of the system.

The experimental setup used in this article is based on an ac fed closed motor control system, as depicted in Figure 1. This system represents a fully functional PMSM main input motor drive, with power factor correction, full control communication signal isolation, and optical encoder feedback. At the core of the system is the ARM Cortex®-M4 mixed-signal control processor from Analog Devices, the ADSP-CM408. This is programmed with combined tools from IAR and MathWorks to achieve complete implementation of the MBD platform.

**Modeling the AC Motor Drive System**

The target drive system is a PMSM with encoder position feedback connected to a 3-phase ac power inverter with isolated phase current feedback. The drive control algorithm is implemented on a mixed-signal application-specific circuit, consisting of an ARM Cortex®-M4 control processor with low-voltage power supply inputs and isolated phase current feedback. The system is designed to provide high performance and reliability in motor control applications.

![Figure 1. Drive system platform. (a) Block diagram of ac fed closed motor control system, and (b) prototype of the system.](image-url)
signal processor (ASSP) that includes peripherals to capture motor feedback signals and control the power inverter.

For modeling purposes, the system has three main components: the power inverter and motor (plant), the control feedback circuits, and the digital controller. The plant model uses Simulink Simscape components to simulate the power inverter electrical circuit and the motor electromechanical elements in the continuous time domain. The feedback circuit models take care of the gain and data type conversions between the controller and motor drive models.

Simulink embedded coder tools create embedded C code that accurately reflects the algorithm execution on both the simulation platform and on the embedded control processor. Successful execution of model-based design requires accurate system and circuit models and appropriate partitioning of the system model and the embedded control software. The simulation solver used is a fixed step discrete solver because of the mix of discrete and continuous time functions in the system.

The drive system hardware consists of a power board, control board, and a PMSM with encoder feedback (as seen in Figure 1b). The power board includes an input rectifier, a 3-phase inverter module, current and voltage sensors, digital and analog signal isolation circuits, and encoder signal buffers. The control board includes the motor control ASSP with a 240 MHz ARM Cortex-M4F core and dedicated motor control peripherals, including PWM timers, quadrature encoder counters, sinc filters, and embedded analog-to-digital converters (ADCs). The hardware includes motor current feedback options of using isolating current sensors with the embedded ADC or current shunts with an isolating ADC \( \Sigma \Delta \) converter and the embedded sinc filters.

The feedback signal acquisition and control algorithm execution is synchronized to the PWM switching frequency using the processor interrupt mechanism. The system simulation uses the same time step since the time constants of interest in the plant are much longer than the PWM switching period. The power inverter uses an average value model since full switching signal simulation does not provide useful control information.

The PMSM motor model, which comes from the MathWorks SimPowerSystems library, is supported by a configuration menu and even preset model parameters. The user may switch in a custom motor or inverter model depending on the focus of the design development.

The motor control (MC) algorithm model is a set of discrete time functions, which execute every time step on both the simulation and embedded platforms. Typically, the MC algorithm functions are included in a single subsystem block to simplify the code generation process. The code generator creates the C code to execute the control algorithm and data structures for the algorithm input, output, and state variables. The algorithm itself is the commonly used field oriented control (FOC) with an outer velocity loop and inner d-axis and q-axis current loops, as depicted in Figure 2.

The inverter interface and feedback paths are split into sensor signal conditioning and embedded interface blocks. The current sensor and signal conditioning models are simple gain elements since their bandwidths extend beyond the range of interest for control feedback. The position sensor model is more complex since it provides a high resolution incremental position signal and a low resolution absolute position signal.

The models for the embedded interface signals include type conversion functions since the ADCs, sinc filter, counters, and timer peripherals have 16-bit or 32-bit fixed point output data registers. The gain of each embedded interface is a function of the peripheral system clock rate, the sample rate, and the interface peripheral register settings. The model parameters must be matched to the embedded system configuration to ensure accurate simulation results.

**Software Partitioning and Code Generation**

The motor drive system performs multiple functions along with the motor control algorithm. The embedded software splits into multiple functional modules for platform flexibility and ease of development. The key code functions are system initialization, communications interface, application tasks, motor control interface, and motor control algorithm. Figure 3 describes the high level drive program flow, while Figure 4 describes the code structure.

The main program calls the initialization routines to configure the ASSP hardware and then places the processor in a continuous wait loop. All other functions are called by event driven interrupt service routines (ISRs). The ADC interrupt has the highest priority and the ADC ISR calls the motor control routine (PMSMctrl) when the motor run flag is set. The selection of the motor current feedback path is made before building the code.

The PWM trip interrupt is asynchronous; it’s only called in response to a hardware fault, and is the only function to log the fault since the hardware PWM trip function automatically shuts off the inverter PWM signals. The communications port ISR has a lower priority, processes user commands, and transmits data captured by a debug monitor function. The kernel timer ISR manages the background application tasks such as motor start and stop sequencing, debug monitor interface, and other housekeeping tasks.

![Figure 2: FOC algorithm.](image-url)
Table 1. Modeling Considerations

<table>
<thead>
<tr>
<th>Block</th>
<th>Modeling/Code Generation</th>
<th>Critical Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electromechanical System (1)*</td>
<td>Inverter: averaged functional model, very simple</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Motor: standard electromechanical DQ model</td>
<td>Stator inductance, back EMF/torque constant</td>
</tr>
<tr>
<td></td>
<td>Mech system: standard stiffness-inertia-friction model, application specific</td>
<td>Load inertia</td>
</tr>
<tr>
<td>Sensors and Interface (2)</td>
<td>Function models</td>
<td>Gains, offsets</td>
</tr>
<tr>
<td>Processor (3)</td>
<td>Peripheral function models</td>
<td>Scale factors, quantization effects</td>
</tr>
<tr>
<td>Driver Circuits (4)</td>
<td>Core algorithm: code generation from here</td>
<td>Efficient implementation, fixed-/floating-point boundaries</td>
</tr>
</tbody>
</table>

*Numbers in parentheses refer to blocks in Figure 5.
Maximum code portability and ease of maintenance is achieved by making the autocode generation nontarget specific, apart from the basic settings, such as type sizes, byte ordering, and so forth. MathWorks provides processor specific code generation modules that directly address processor peripherals and drivers. While this capability can be attractive in some circumstances, the downside is that the code becomes less portable and any changes in device drivers or peripheral configuration (for instance, in a new processor variant) will require changes to the code. Hence, in the design example presented here, the code generation is confined to the control algorithm only, with all peripherals having functional models in the Simulink model and being hand coded in the application project. This approach is emphasized in Figure 6, in which it can be seen that the code generation from the MathWorks controller model is linked to the other code and library modules in the main application project.

The Simulink model with partitioned model blocks is illustrated in Figure 7. As indicated, code is generated from the motor control algorithm part of the model. The important settings for the code generation are selected in the configuration parameters → hardware implementation window where an overall device type can be selected, and in the configuration parameters → code generation → interface window, where the standard math library is selected.

Another factor when it comes to code efficiency is the C language dialect used. Two very common dialects supported by most code generation tools and embedded workbenches are C89/C90 and C99. First and foremost, it is important that the same dialect is used across all tools. For example, if the embedded workbench is configured to build code according to C99, the automatic code generation tools must build codes according to the C99 standard also. Failing to do so can result in a significant drop in code performance and in the worst case, unintended functioning of the code.

Another important factor is fixed point vs. floating-point type representation. Fixed point is supported by both coding dialects, so for this case the choice of dialect does not matter, as long as the same dialect is used across all tools. However, if floating-point types are used, the choice of C dialect becomes much more important.

C89/C90 does not distinguish between single precision floating point and double precision floating point. If the code is going to run on a processor that supports double precision, this may be acceptable, but for a processor that only supports single precision, such as an ARM Cortex-M4, the distinction makes a big difference. With this in mind, it is crucial to make sure the autocode generation tool, as well as the embedded workbench, is set up to use C99 dialect.

Simulink provides toolboxes such as Simscape and SimMechanics that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm. For the algorithm itself, there are some useful blocks, such as Park transformations and sine cosine CORDIC approximation blocks that can be used to easily model the electromechanical system once physical parameters are known. Even if physical parameters are not fully characterized, predefined models of components such as motors can be loaded with approximately matched specifications to enable initial design of the motor control algorithm.

The autocode interface is defined by an initialization function call, and one or more time step function calls, which must be called from within the main application code at the appropriate time steps. In this example, there are two time steps functions—the main control algorithm, which is called at the PWM rate of 10 kHz, and a speed measurement function, which is called at a 1 kHz rate. The autogenerated code modules are integrated into the main project as illustrated in Figure 8.
As indicated, the code is organized in a modular fashion that makes inte-
gration of application specific functions, such as networking and protection,
quite straightforward. High priority tasks, such as the motor control algorithm,
are called from the ISRs of Figure 3. Application level tasks are called as
scheduled tasks from a basic scheduler kernel. The MC interface routines
are included in the motor control and measure code blocks, where the latter
includes all of the current feedback signal handling codes. The ADI monitor
code includes the debug monitor function used for system tests, which cap-
tures application and control algorithm signal data while the motor is running.
The data is transmitted over a serial link to a PC for display and analysis.

System Test and Debug
Having determined the critical parameters of Table 1 by measurement, calcu-
lation, and data sheets, the appropriate controller gains for velocity and cur-
rent loops can be determined using the Simulink model. This can be achieved
utilizing either standard PID tuning approaches [2] or the tuning tools provided
by MathWorks, such as the PID tuner tool. This process is described in more
detail in Reference 3.

The performance of the current loop for both modeled and experimental
operation is shown in Figure 9 and Figure 10. The experimental data is only
sampled every 5 ms in this plot, so there is some aliasing, but the overall
trend is clear enough.

The performance of the model-based autogenerated code can be determined
by examining the code execution timeline over a PWM period. This can be
done using I/O pins and a scope, or more easily, using the ITM event feature
in the IAR Embedded Workbench C-SPY Debugger. The sequence of events
within the PWM cycle is illustrated in the timeline of Figure 11.

A PWM synchronization pulse occurs at the start of every new PWM cycle,
and this is connected in hardware to the ADC timer, which controls the
sampling of each ADC channel. In this case, motor currents are sampled
immediately after the PWM sync pulse and direct memory accessed (DMAed)
to memory, after which the algorithm is executed and updated
values for PWM duty cycle are generated. As illustrated in Figure 11, the
execution of the model-based autogenerated code consumes less than 10%
of the PWM cycle, allowing plenty of overhead for additional background
tasks. Historical concerns about the efficiency of autogenerated codes are
no longer a significant factor.
In terms of code footprint, the relative size of the algorithm autocode is shown in Table 2, where it can be seen that the autogenerated code occupies just over 10 kb of memory, which equates to almost 15% of the total footprint. The available SRAM on the ADSP-CM408 of 384 kB clearly accommodates this memory requirement with no difficulty, allowing the program to be run from SRAM, at the fastest rate possible, with more than enough headroom available for more complex algorithms and additional monitoring or user interface features.

<table>
<thead>
<tr>
<th>Modules</th>
<th>Code (Bytes)</th>
<th>Data (Bytes)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motor control peripherals (PWM, encoder)</td>
<td>1584</td>
<td>94</td>
<td>2.3%</td>
</tr>
<tr>
<td>Measurement peripherals (ADC, sinc, serial monitoring)</td>
<td>2224</td>
<td>2691</td>
<td>6.7%</td>
</tr>
<tr>
<td>PFC controller configuration and monitoring</td>
<td>3032</td>
<td>2149</td>
<td>7.1%</td>
</tr>
<tr>
<td>User interface peripherals (UART, LCD)</td>
<td>1484</td>
<td>5055</td>
<td>9%</td>
</tr>
<tr>
<td>System (kernel, state machine, startup)</td>
<td>1192</td>
<td>1364</td>
<td>3.5%</td>
</tr>
<tr>
<td>Algorithm autocode</td>
<td>5656</td>
<td>5022</td>
<td>14.7%</td>
</tr>
<tr>
<td>Drivers, libraries</td>
<td>21,596</td>
<td>19,737</td>
<td>56.7%</td>
</tr>
<tr>
<td>Total</td>
<td>36,768</td>
<td>36,112</td>
<td>72,880 (100%)</td>
</tr>
</tbody>
</table>

**New Application Development**

The premise of the SW discussed in this article has been a system consisting of two main components. The first is a model-based component that implements control algorithms. Though the model is developed with the embedded target in mind, the code obtained from the autogeneration tool is essentially generic. Second, there is a handwritten SW component that ties the generic algorithm code to the embedded target, handles scheduling, and assigns processor resources. In terms of reuse of models and scalability, this system partitioning has several benefits.

In this article development of control for a single motor (single axis) has been discussed. Now, imagine that the drive specification calls for control of two motors (dual axis) by the same processor. It goes without saying this is a radical change of the system but the strength of working with generic models is becoming evident. The single-axis model already developed did not make any assumptions regarding processor peripherals—it is a general control algorithm for a PM motor. To create a model that is capable of controlling single/dual axes is therefore only a matter of creating a second instance of the single axis model.

Naturally, the handwritten code requires tailoring to support single/dual axis, but assuming the processor has the right set of peripherals and the computational resources to control dual axes, the modifications to the handwritten code are straightforward. Regardless of controlling single or dual axes, the main tasks of the handwritten code are to assign values to the model’s input, write the model’s output to processor peripherals, and schedule when the model is to be executed. Therefore, going from single to dual axes is mostly a matter of assigning/configuring peripherals and schedule execution of the algorithm of the added axis. The process is seamless and enabled by the fact the model is generic.

The benefits of using a model-based design may be limited if only a one control system is developed. However, in most cases, product development means multiple product variants and for these cases reuse of models becomes attractive—not only because of reduced development time but also because of the increased quality that comes from using trusted models. Over time, an algorithm developer will create a library of models and, if implemented correctly, these models can be reused across products. And because the models are generic they can run on the processors available today, as well as the ones that will be available tomorrow.

In addition to meeting potential requirements for product variants or control of multiple axes, developers can sometimes provide different controller modes. A typical example is an application that offers torque control, velocity control, and position control modes. Development of a position control algorithm can be layered on the fundamental building blocks of the current and speed control algorithms.

In most applications, a position control loop is applied as an outer loop around the inner velocity and current loops. A basic position controller requires only a proportional gain term. An integral term is not generally required since any steady state error in the position loop will result in a nonzero speed reference. Provided that the inner current and velocity loops are well tuned, these can be regarded as ideal unity-gain blocks, and the task of tuning the position loop becomes a straightforward task.

In addition to the outer proportional control loop, it can be important to include a position reference profiler in order that the load follows defined periods and rates of acceleration and deceleration. This is important for minimizing mechanical stress in many systems. In this application example, a constant acceleration, constant speed, and constant deceleration profile is applied to the position reference change. This is illustrated in Figure 12, in which the position reference, profiled position reference, and corresponding ideal velocity profile are indicated. The extent to which the actual velocity follows this profile is dependent on the dynamic response of the velocity controller.

![Figure 12. Profiled position reference.](image_url)

All of this functionality—position loop gain, position profiler, as well as ancillary functions, such as home positioning and end-stop detection, is implemented as additional blocks in the model-based part of the code. The only additional handwritten code change required is configuration of I/O to accommodate home position and end-stop signals.

**Conclusion**

Model-based design can be a powerful tool in accelerating embedded development for motor drive manufacturers. If set up and configured in a generic manner, it can significantly ease the burden of handwritten code development and maintenance. It can also accelerate time to market, as code development can proceed initially in the absence of hardware, if reasonably accurate models of the key system components are available.

These features have been shown in the context of a PMSM drive operated under FOC with extensions to multiple axis and position control. A partitioning approach to the software modules and the model-based components has
been detailed that optimizes the value provided by the model-based solution. The advantages of the model in terms of optimizing the velocity controller parameters, and the compactness and efficiency of the code generation, have also been illustrated from the experimental data.

References


About the Author

Dara O’Sullivan is a system applications engineer for the motor and power control group at Analog Devices. His area of expertise is power conversion and control in ac motor control applications. O’Sullivan received his B.E, M.Eng.Sc., and Ph.D. from the University College Cork, Ireland. He can be reached via email at dara.osullivan@analog.com.

Jens Sorensen is a system applications engineer at Analog Devices, where he works with motor control solutions for industrial applications. His main interests are control algorithms, power electronics, and control processors. While currently focusing on industrial applications, Sorensen developed motor control and power electronics for appliance and automotive applications earlier in his career. He can be reached via email at jens.sorensen@gmail.com.

Aengus Murray is the motor and power control applications manager for the automation, energy, and sensors unit at Analog Devices. He is responsible for the complete Analog Devices signal chain offering for industrial motor and power control. Murray holds a bachelor and doctorate degree in electrical engineering from University College Dublin in Ireland. He has over 30 years of experience in the power electronics industry and has also worked with International Rectifier, Kollmorgen Industrial Drives, and Dublin City University. He can be reached via email at aengus.murray@analog.com.

Online Support Community

Engage with the Analog Devices technology experts in our online support community. Ask your tough design questions, browse FAQs, or join a conversation.

ez.analog.com