2011-01-2267

# On-Target Rapid Prototyping: A Practical Approach for Bootstrapping Production ECU Software Development

**Tom Erkkinen**
MathWorks, Inc.

**Seshadri Shekar**
Automotive Infotronics Private Limited

**Mohamed Ziaudeen**
Automotive Infotronics Private Limited

**Shobhit Shanker**
MathWorks, Inc.

Copyright © 2011 MathWorks, Inc

## ABSTRACT

Rapid control prototyping (RCP) is a widely used technique for verifying a controller's functional behavior. Typically, RCP uses a target processor with ample processing power and memory, which makes the technique attractive for engineers exploring new concepts. However, a large gap often exists between the RCP target and the production ECU in terms of the available code generation technology, the supporting tool chain, and I/O hardware. Consequently, significant work is required when migrating a controller from RCP to production. Furthermore, due to cost constraints, RCP systems are difficult to deploy in large numbers for fleet testing or preproduction trials.

In response to the challenges associated with RCP, automotive engineers are moving towards a technique called on-target rapid prototyping (OTRP). With OTRP, the code is generated, cross-compiled, and downloaded either to the ECU used in production or a development version of it with additional memory and instrumentation support. OTRP enables engineers to use the same code generator, supporting tool chain, and ECU hardware during development, simplifying the migration to production. In addition, due to the relatively low cost of development ECUs, OTRP systems can be deployed in large quantities.

This paper provides an introduction to Model-Based Design and OTRP, a step-by-step approach for getting started with OTRP using a new algorithm export technique, and considerations for moving from OTRP to production. An application example is provided to illustrate how OTRP has been effectively used for a production-intent ECU program, which includes a novel external mode implementation on a resource constrained fixed-point embedded system.

## INTRODUCTION TO MODEL-BASED DESIGN

A model represents a dynamic system whose response is a function of its inputs, state, and time. Historically, system engineers have used block diagrams as shown in Figure 1 to model plant environments and physical systems as well as to design ECU algorithms.

In recent years, graphical modeling environments consisting of block diagrams and state machines have been used to analyze, simulate, prototype, specify, and deploy software algorithms in production ECUs. Model-Based Design refers to the use of models and modeling environments as the basis for ECU development.

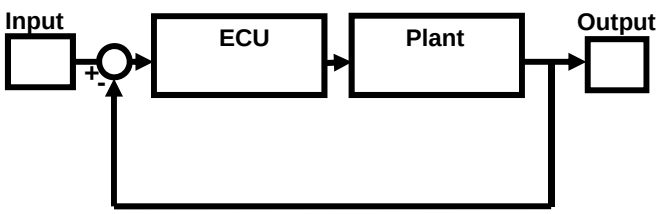


Figure 1: Feedback controller model.

Automotive systems developed using Model-Based Design include:

- Engine and transmission ECUs
- Hybrid, battery, and green vehicle systems
- ABS and chassis control systems

- Climate control and body electronics
- Instrument clusters and displays
- Radio receivers and audio signal processing systems

Used throughout the system development life cycle, Model-Based Design enables continuous verification and validation of requirements, designs, and implementations. This approach affords significant advantages for formal software processes as well as for any project on which risk management, error prevention, or early error detection is priorities.

Model-Based Design comprises the following main activities:

- Modeling and simulation
- Rapid prototyping
- Embedded deployment
- In-the-loop testing
- Integral activities

MODELING AND SIMULATION – A block diagram model of a dynamic system is represented schematically as a collection of blocks interconnected by lines that represent signals. The signals are the inputs, outputs, and states of the blocks.

Blocks and lines can be real or virtual. Virtual blocks or lines have no effect on the simulation results but aid in constructing or understanding diagrams. Blocks and subsystems, whether they are real or virtual, can be stored in custom libraries to facilitate reuse and abstraction. Large models can be structured using model referencing, wherein a top level model invokes lower level models in a way that reduces memory consumption and enables faster simulation and model update times.

Simulation can be accomplished in two ways. One way is to use an in-memory representation of the model and execute the simulation in an interpretive mode. The second way is to generate code from the model and execute the code using a technique known as simulation through code generation. Interpretive simulation provides users with more control of the execution environment and greater interaction capabilities, but it can be slower for large models. Simulation through code generation provides less user interaction but more speed. For this reason, it is also known as simulation acceleration. Models using model referencing can simulate in normal (interpretive) or accelerated mode.

RAPID PROTOTYPING - In bypass rapid prototyping, code is generated from the controller or algorithm model. The code is then cross-compiled and downloaded to a high-speed (often floating-point) rapid-prototyping computer where it executes in real time. I/O is typically managed by a memory pod or emulation device that is connected to both the rapid prototyping computer and an existing ECU, perhaps still residing in a vehicle. Other I/O options include communication via buses, such as CAN, and may require custom signal processing and power electronics. The controller parameters are tweaked “on-the-fly” during test drives or in the lab with the actual plant (e.g., engine) and allow insertion of new code to bypass existing ECU code. When a set of parameter values is identified that enables performance requirements to be met, the new algorithm is deemed *feasible*. See Figure 2.

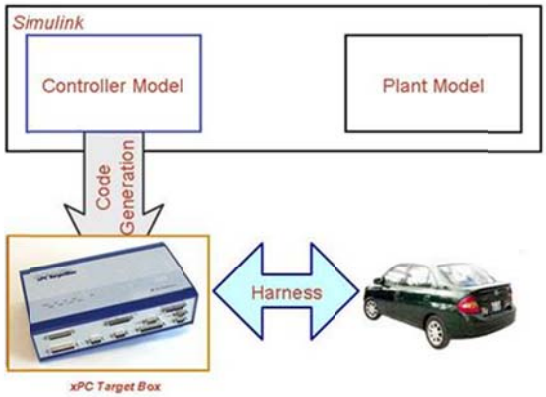


Figure 2: Bypass rapid prototyping.

In OTRP, as with bypass rapid prototyping, code is generated just for the controller portion of the model. However, in OTRP the cross-compiled code is not deployed to a rapid prototyping computer, but rather to the embedded microprocessor or ECU used in production, or perhaps to a close approximation of it configured with a little more memory and I/O. OTRP often uses an integer processor and thus needs a more detailed, fixed-point model, as opposed to the floating-point processors and models used for bypass rapid prototyping. I/O is managed via standard ECU devices.

The host computer then optionally interfaces directly with the ECU hardware, perhaps residing in fleet vehicles, for parameter tuning using external mode or a calibration tool. Controller parameters are then tweaked “on-the-fly.” When performance requirements are met, the new algorithm has been shown to be both feasible and *practical*; that is, it will work in a production, resource-constrained environment. See Figure 3.

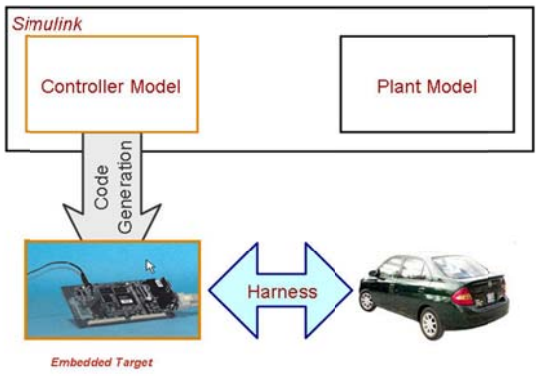


Figure 3: On-target rapid prototyping.

Table 1 compares traditional bypass prototyping with OTRP.

| | Traditional RP | On-Target RP |
|---|---|---|
| *Purpose* | Useful for testing new ideas and green-field research | Useful for refinement and calibration of designs during development |
| *Execution Hardware* | Uses PC or non-target HW | Uses ECU or near-production HW |
| *Code Efficiency, I/O latency* | Less emphasis on code efficiency and I/O latency | More emphasis on code efficiency and I/O latency |
| *Programs* | Works well for new vehicle programs | Works well for delta changes to existing programs |
| *Engineers* | Typically done by systems engineers in R&D or advanced production | Typically done by systems and software engineers in production |
| *Cost and Convenience* | May require custom real-time simulators and hardware, or may be done with inexpensive "off-the-shelf" PC hardware and I/O cards | May use existing hardware, thus more convenient and less expensive, particularly for deploying in large numbers |

Table 1: Traditional vs. On-target rapid prototyping.

EMBEDDED DEPLOYMENT - After rapid prototyping, the controller model is often converted to a detailed, executable software specification. There are a number of factors to consider here, such as function and file partition, startup and shutdown procedures, diagnostics, and built-in test routines. The model is constrained and elaborated to perform properly on embedded system hardware.

Embedded code is then generated for the detailed controller model and downloaded to the actual embedded microprocessor or ECU as part of the production software build. No simulation activity is associated with this step. The key here is to ensure that the final build has fully integrated the automatically generated code with existing legacy code, I/O drivers, and real-time operating system (RTOS) software.

IN-THE-LOOP TESTING - Simulation of models is one of the first verification and validation (V&V) steps. Testing models requires a more rigorous approach than the ad-hoc simulation runs that are often used in early algorithm development. Model testing demands a systematic approach to the creation and execution of test cases. Special blocks, such as signal builders and assertions, facilitate this type of test procedure. Structural coverage analysis for the model and the code helps assess test completeness. After model testing, there are several ways to test the generated code.

Software-in-the-loop (SIL) testing involves executing the production code for the controller within the modeling environment in non-real-time with the plant model. The code executes on the same host that is used by the modeling environment. A code wrapper generated with the code provides the interface between the simulation and generated code. See Figure 4.

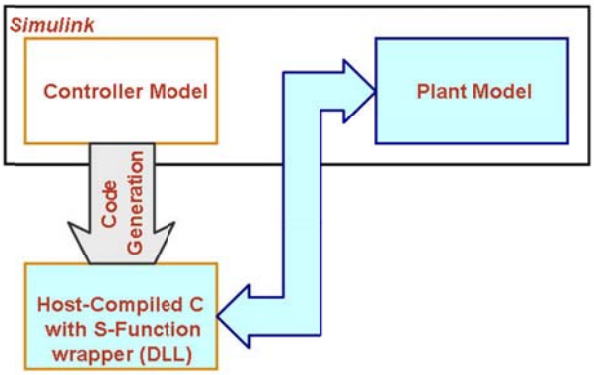


Figure 4: Software-in-the-loop testing.

Processor-in-the-loop (PIL) testing is similar to SIL in that it too executes the production code for the controller. However the code executes on the actual embedded processor or an instruction set simulator, and thus, verifies the code on the actual target. A CAN bus or serial devices are used to pass data between the production code executing on the processor and a plant model in the modeling environment. As with SIL, PIL is used for non-real-time testing of the generated code. See Figures 5.

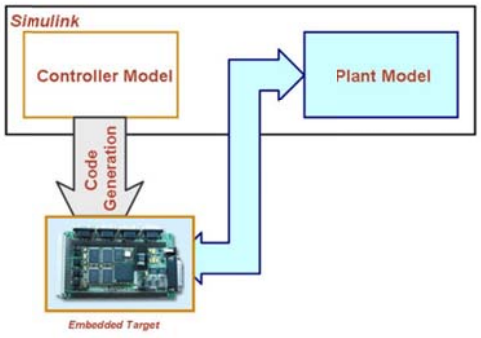


Figure 5: Processor-in-the-loop testing.

For hardware-in-the-loop (HIL) testing, the code is generated for the plant model. It runs on a highly deterministic, real-time computer. Sophisticated signal conditioning and power electronics are needed to properly stimulate the ECU inputs (sensors) and receive the ECU outputs (actuator commands). Whereas rapid prototyping is often a development or design activity, HIL serves as a final lab test phase before final system integration and field tests commence. See Figure 6.

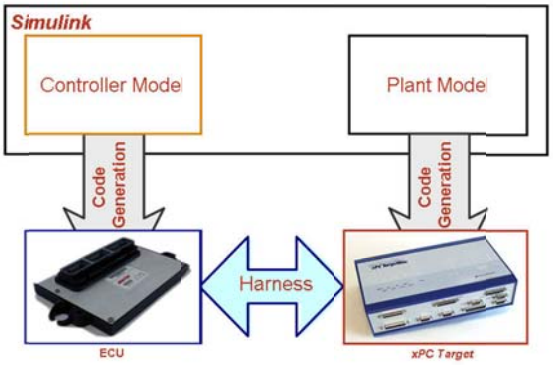


Figure 6: Hardware-in-the-loop testing.

INTEGRAL ACTIVITIES – In Model-Based Design, a number of integral activities span the entire development cycle such as documentation. Documentation, like code, can be automatically generated from models. Documentation can be produced in a template form, to which engineers add the content of each documentation section. Requirements traceability is accomplished using interfaces between blocks in the model and existing requirements management sources. The code generated from the model can also be traced back to the blocks, enabling auditors to trace high-level requirements all the way to the code, and trace code back to the requirements. As with requirements management, source control for a model is accomplished outside the modeling environment using existing source control products. Interfaces between the modeling environment and source control tool are available and enable developers to automatically check in and check out models, as well as to document changes.

## STEP-BY-STEP OTRP APPROACH

There are two approaches to code generation for embedded deployment. The first approach, algorithm export, generates code for the functions and then integrates them into the overall hand-written application. The second approach, full executable generation, uses the model to fully generate the entire application.

The algorithm export approach is better suited to suppliers who need to port their models to multiple processors and thus require more flexibility. The full executable model requires that device driver blocks be included with the model, even though the drivers are not simulated. The full executable approach may be used by OEMs who do not need algorithm portability because they are locked down to one processor architecture for several years. Figure 7 illustrates the algorithm export approach.

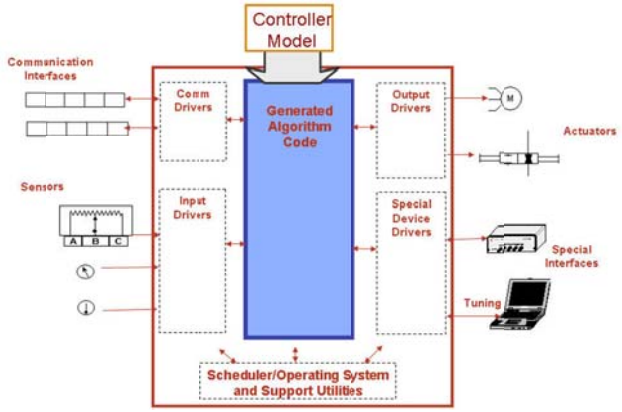


Figure 7. Algorithm export code generation.

The basic steps for performing OTRP are described below. They begin with deciding if a full executable or algorithm export technique should be used.

With either approach it is important to establish model guidelines and code generation settings that enhance design clarity and code generation efficiency. The Simulink Model Advisor and the Code Generation Advisor from MathWorks help in these areas. It is also important to establish a baseline model and simulation in floating point then convert to fixed-point if needed.Tool automation for this conversion is quite useful here.

In summary, the initial Model-Based Design steps that apply to both OTRP approaches discussed below are as follows:

1. Establish model guidelines using Simulink Model Advisor.

2. Establish code generation settings using Code Generation Advisor.

3. Create the floating point algorithm design using modeling tools. Simulink and Stateflow.

4. Convert to fixed-point data, if required, using fixed-point automation tools.

5. Compare fixed-point and floating-point results.

FULL EXECUTABLE APPROACH - With full executable integration, target-specific blocks are added to the generic algorithm model created in the previous phase. The target-specific blocks can represent device drivers or target-optimized functions. Target-specific code cannot execute on the host processor, so these blocks typically specify a default value or pass-through behavior used for simulation mode. To improve model portability, developers should keep the algorithm component in a separate subsystem or model from the target-specific framework blocks.

In Simulink, S-functions are employed to create device driver blocks. S-functions can be created by hand or

automatically generated using the Legacy Code Tool in Simulink. The S-functions need to be in-lined to optimize the generated code, which then calls the target-specific code without a code wrapper.

Code replacement libraries are employed to create the processor-optimized code. Typically, developers use a code replacement library to map basic operators and math functions to more optimized versions. For example, with TFL it is easy to generate code with a pragma or hardware instruction that enables automatic saturation on overflow protection. Replacements can also be used to create a highly optimized trigonometry function.

In addition to S-functions and code replacements, developers need to create a custom main file and automate the build process to invoke the cross-compiler tool chain. The compilation, download, and execution can be fully automated. These build customizations are done using source file templates, template make files, and hook APIs that help control the build process.

Finally, a System Target File can be created to enable the fully customized build solution described above. A more detailed discussion of these topics is available in the Simulink Coder documentation [1].

The key steps can be summarized as follows:

1. Create a System Target File (STF) baseline target without drivers.

2. Add device driver blocks using S-functions.

3. Add code optimizations using Target Function Libraries.

4. Select the STF to generate and compile code.

5. Download and run the code on the target processor.

6. Tune parameters using external mode or a third-party calibration tool (optional).

Tuning parameters using external mode requires a communication interface between the host and target. Once this is established, external mode APIs can be used to create a program for interactive communication. A basic example that uses TCP/IP is provided with Simulink, and other options are available.

For example, to use CAN, developers would need host CAN support, perhaps using Vehicle Network Toolbox, which they would use to communicate with the target's CAN support. A CAN Calibration Protocol (CCP) block could be developed from scratch or ported from an existing example. Then during code generation, developers can select the option for creating ASAP2 files to define the data and memory locations.

See [2] for an example full executable OTRP approach using in-house real-time operating system integration.

ALGORITHM EXPORT APPROACH - As with the full executable approach, developers should establish model guidelines [3] with code generation settings, and create the floating- or fixed-point design. However, device driver blocks are not going to be created since these are in the hand-coded framework software. The framework will also call the generated algorithm code. The key to a successful call, or integration, is to establish the appropriate call interface and reference but not redefine the interface data.

Embedded Coder provides a number of options for controlling the function signature of the generated code. By default, it generates the initialize, step, and terminate functions as void-void functions with global data. Separate data structures are created for each category of data in a Simulink model. Variables created for Simulink inports, outports, parameters, and states are each placed in separate data structures. Storage for each of the structures is allocated by the generated code.

Developers can change this default behavior and pass pointers to each of the structures as arguments to the initialize, step, and terminate functions. In this case, the generated code will no longer rely on global data and can be reused. The developer is responsible for declaring storage for each of the data structures.

Developers can also explicitly control the prototype of the initialize, step, and terminate functions. In addition to the function name, developers can specify the argument names, passing mechanisms (by reference or by value), and qualifier. The dialog box for controlling a function prototype is shown in Figure 8. This example is for a Simulink model with four inports and one outport. The default void-void prototype was changed to pass the inputs in several ways, while the Simulink outport is set to be a return value from the function.

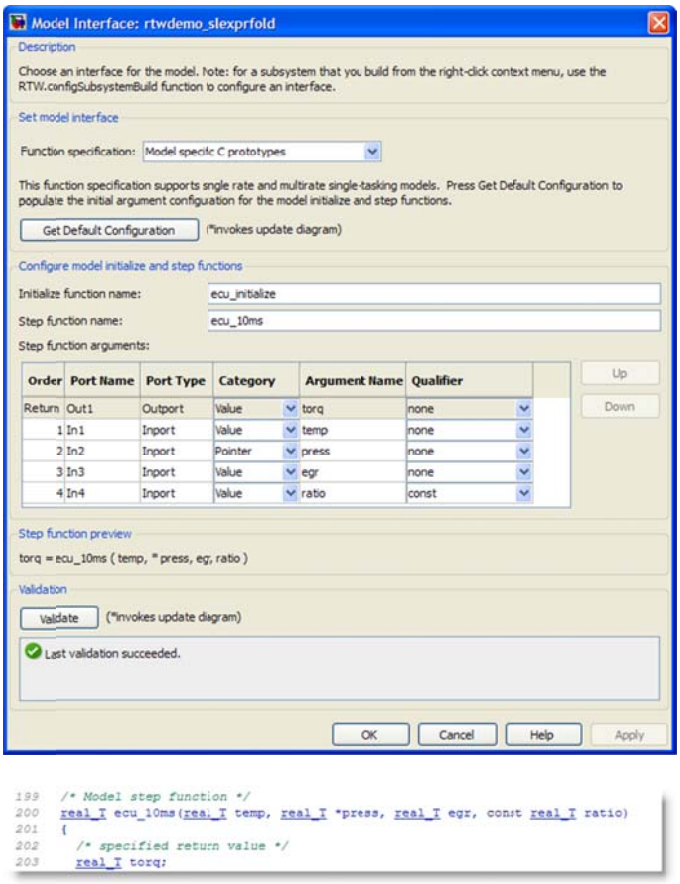


Figure 8. Function Prototype Control and Generated Code.

The model data can be controlled using Simulink Data Objects, MPT Data Objects, and Custom Storage Classes (CSCs). Once a model signal or parameter is associated with a data object, it is straightforward to assign it a CSC, such as a global variable or get/set access method. An imported extern CSC is useful during algorithm export since it can reference an existing data item (such as a calibration parameter defined outside the model in a separate data dictionary) and use it in the model and generated code without redefining it.

The key steps used to export an algorithm for embedded system deployment are as follows:

1. Generate code with an appropriate call interface.
2. Establish import and export data interfaces.
3. Create files and functions as required for integration with the external framework code.
4. Use the existing scheduler and production build process to invoke the generated algorithm code.
5. Download and run the code on the target processor.
6. Tune parameters using a third-party calibration tool (optional).

## CONSIDERATIONS IN MOVING FROM OTRP TO PRODUCTION CODE GENERATION

With an OTRP framework established, it is a natural next step to consider deploying in production. The activities that may need to be refined include code optimization, code verification, certification, and standards compliance.

Code replacement libraries, discussed previously, are optional for OTRP but are crucial for production because code needs to be highly optimized to reduce per unit costs. This enables a direct mapping between the ANSI-C code normally output by the code generator and the target-specific code supported by a particular ECU processor.

Once the code is generated, it is important to verify its behavior. PIL testing, as described earlier, is an effective verification mechanism. PIL is especially important when target-optimized code is used since it is not possible to test the target code on the host computer during simulation. MathWorks offers a variety of PIL solutions, including PIL APIs that enable developers to quickly create their own target-based test bench.

See [4] for a description of TFL and PIL testing.

AUTOSAR (AUTomotive Open System ARchitecture) represents a special case of external framework code. This emerging automotive standard is well supported by Simulink. When working with AUTOSAR, developers can eliminate many of the steps described above and instead select the AUTOSAR system target file provided by Embedded Coder. They would then use a runtime environment (RTE) generator to import the XML description produced during code generation for target integration.

Another emerging trend for high integrity production ECUs is ISO 26262 certification. Refer to [5], which describes the use of Simulink in IEC 61058 and ISO 26262 applications.

## CASE STUDY – AIPL

Automotive Infotronics Private Limited (AIPL) is a joint venture between Ashok Leyland Limited and Continental AG. It designs, develops, and adapts electronics products and services for the transportation sector. The charter of this organization is to offer products to developing market OEMS meeting a better price/performance point compared to existing products in the market. The company develops electronic components and software for units such as instrument cluster applications, body control electronics and various other control units mostly for both commercial vehicle applications [6].

BODY CONTROL UNIT/MULTIPLEXER AND INSTRUMENT CLUSTER DEVELOPMENT

The first two flagship products of AIPL are the body Control Unit/Multiplex (BCU/MUX) and Instrument Cluster (IC) for a major Indian OEM. The software for these products was completely developed in-house. The application software was fully developed with MathWorks products (MATLAB, Simulink, Stateflow, and Embedded Coder) and the device driver layer was developed with the CodeWarrior® IDE from Freescale® for the BCU and with the SOFTUNE® IDE of Fujitsu® for the IC. The models were verified for algorithmic correctness through model-in-the-loop (MIL) simulation. Using the Simulink Fixed Point product, the model was converted to a fixed-point model representation, SIL testing was used to verify target word length restrictions and test for overflow conditions. To save time and reduce the potential for build errors, MATLAB scripts were written to facilitate automatic code generation of the production code from the application models and to compile the device driver files and application files. These scripts are also responsible for linking the object code and flashing it to the target processor.

MODEL AND MODELING ARCHITECTURE

The BCU (see Fig. 9), MUX (see Fig. 10), and IC (see Fig. 11) work together in an integrated mode. The BCU processes analog, digital and frequency inputs to derive values that will drive the IC as well as digital signals to directly drive vehicle loads.

The modules exchange their computed results with each other and view commands using a modular approach using Simulink Subsystems. Using non-virtual subsystems to model the entire system application, AIPL configured the code generator (Embedded Coder) to generate code that interfaced smoothly with the hand-coded scheduler.

In this process, for the CAN Calibration Protocol (CCP) to work, the calibration parameters must be grouped and placed in a predetermined section of memory. This was achieved using #pragma directives for the target compiler. To automate grouping of the parameters during the code generation, AIPL engineers defined the parameters that needed calibration as a custom storage class (CSC). This caused the code generator to group all the parameters of this storage class type under a single directive. The name of the directive is target dependent.

Figure 9. Body Control Unit (BCU)

Figure 10. Multiplexer (MUX)

Figure 11. Instrument Cluster (IC)

A key feature of this architecture is that it enables on-line calibration of the target. For this purpose, the CAN Calibration Protocol (CCP) is implemented through Simulink external mode and Vehicle Network Toolbox. For the BCU, the tunable parameters include module thresholds, timings, vehicle parameters, and sensor tolerances. For the IC running critical, vehicle-specific algorithms this on-line tuning proved particularly helpful for the OEM as many vehicle specific parameters needed to be tuned. AIPL provided a MATLAB-based

GUI (see Figure 12) to enable the on-line parameter calibration.

The ability to customize the Simulink model environment to process user inputs has provided many design capabilities that would otherwise be very difficult to implement. In summary, the MathWorks tools provided AIPL with a single environment for algorithm modeling, in-the-loop simulation, verification and validation, code generation, build and deployment, and calibration GUI development. Combined with the ability to interface with a running target, these capabilities would typically require several different tools, with their associated costs and integration issues.

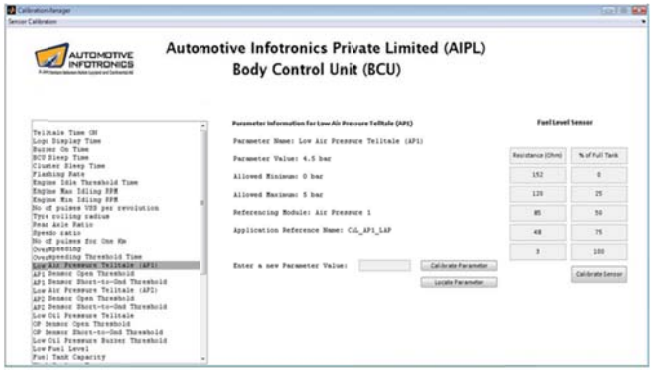


Figure 12. MATLAB-based GUI for calibrating the BCU via CCP.

TESTING AND VALIDATION

After development, the AIPL products underwent an exhaustive testing procedure for validation against the customer requirements. Following these tests, the products underwent field trials in the real vehicle.

PROJECT RESULTS AND CONCLUSION

For the BCU/MUX and IC, Model-Based Design reduced development time of the application software by up to 40%. Moreover, compared to hand-coding, the model-based approach made it easier to handle requirements changes and resolve defects. Using MathWorks tools, AIPL engineers specified the system requirement in the form of model, and relied on code generation for the implementation. This significantly shortened development time on these products, for which time-to-market was critical.

## CONCLUSION

Automatic code generation with Model-Based Design offers embedded system developers a number of advanced options for prototyping, deploying, and verifying production software. Understanding the potential applications of code generation is important, because simply applying the technology is not going to improve production processes. Embedded system developers must establish a production workflow that leverages code generation technologies and yet fits within well-established software engineering principals, such as reducing complexity and establishing proper procedures for verification and validation as well as calibration.

The case study described how AIPL used Model-Based Design to develop a Body Control Unit/Multiplex and Instrument Cluster and reduce development time by 40%.

## REFERENCES

[1] Developing Custom Targets, Simulink Coder User Guide, MathWorks, 2011, www.mathworks.com/access/helpdesk/help/toolbox/rtw.

[2] Automatic Code Generation – Technology Adoption Lessons Learned from Commercial Vehicle Case Studies, T. Erkkinen MathWorks, S. Breiner John Deere, SAE Commercial Vehicle paper 08AE-22, 2007, www.mathworks.com/mason/tag/proxy.html?dataid=9939&fileid=44540.

[3] Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow - Version 2.1, MathWorks Automotive Advisory Board (MAAB), 2009, www.mathworks.com/automotive/standards/maab.html.

[4] Fixed-Point ECU Code Optimization and Verification with Model-Based Design, T. Erkkinen MathWorks, SAE Congress paper 2009-01-0269, 2009.

[5] Qualifying Software Tools According to ISO 26262, M. Conrad and P. Munier MathWorks, F. Rauch TÜV SÜD, Model-Based Development of Embedded Systems, 2010.

[6] Ashok Leyland and Siemens VDO in Infotronics JV, Press Release, July 2007. www.siemens.co.in/en/news_press/index/news_archive/jul_16_2007.htm