

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

Thomas Erkinen
MathWorks Inc.

Anupam Gangopadhyay
Engineering Automotive Infotronics Private Limited

Shobhit Shanker
MathWorks Inc.

Seshadri Shekar and Ziaudeen Mohamed
Automotive Infotronics Pvt, Ltd.

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.

CITATION: Erkinen, T., Gangopadhyay, A., Shanker, S., Shekar, S. et al., "On-Target Rapid Prototyping: A Practical Approach for Bootstrapping Production ECU Software Development," *SAE Int. J. Passeng. Cars - Electron. Electr. Syst.* 5(1):2012, doi:10.4271/2011-01-2267.

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.

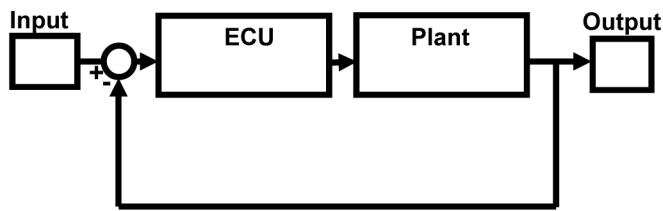


Figure 1. Feedback controller model.

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](#).

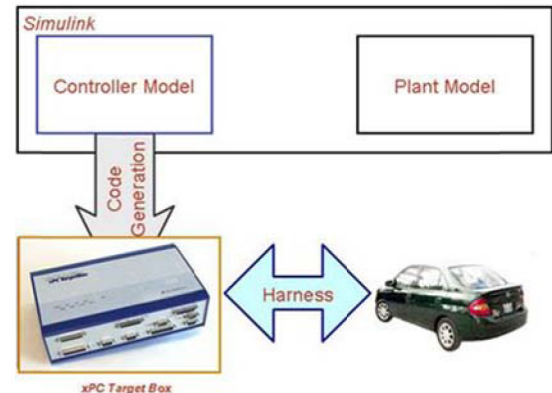


Figure 2. Bypass rapid prototyping.

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.

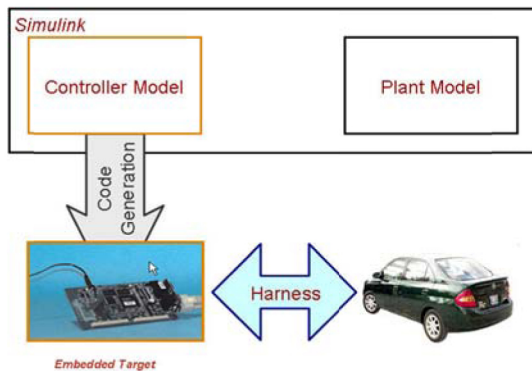


Figure 3. On-target rapid prototyping.

Table 1 compares traditional bypass prototyping with OTRP.

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

	Traditional RP	On-Target RP
<i>Purpose</i>	Useful for testing new ideas and green-field research	Useful for refinement and calibration of designs during development
<i>Execution Hardware</i>	Uses PC or non-target HW	Uses ECU or near-production HW
<i>Code Efficiency, I/O latency</i>	Less emphasis on code efficiency and I/O latency	More emphasis on code efficiency and I/O latency
<i>Programs</i>	Works well for new vehicle programs	Works well for delta changes to existing programs
<i>Engineers</i>	Typically done by systems engineers in R&D or advanced production	Typically done by systems and software engineers in production
<i>Cost and Convenience</i>	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

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.

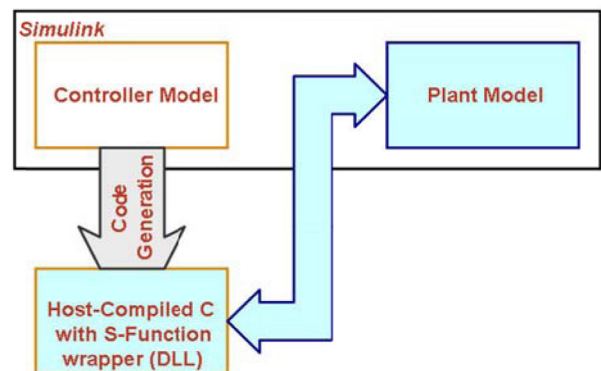


Figure 4. Software-in-the-loop testing.

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.