2008-01-0744

Fixed-Point ECU Development with Model-Based Design

Tom Erkkinen
The MathWorks, Inc.

ABSTRACT

When developing production software for fixed-point Engine Control Units (ECUs), it is important to consider the transition from floating-point to fixed-point algorithms. Systems engineers frequently design algorithms in floating-point math, usually double precision. This represents the ideal algorithm behavior without much concern for its final realization in production software and hardware. Software engineers and suppliers in mass production environments, however, are concerned with production realities and often need to convert these algorithms to fixed-point math for their integer-only hardware.

A key task is to design scale factors that maximize code efficiency by minimizing the bytes used, while also minimizing quantization effects such that the fixed-point algorithms match the floating-point results within an acceptable numerical margin. This floating- to fixed-point conversion task is tedious, labor intensive, error-prone, and often requires multiple iterations between system and software engineers.

Model-Based Design simplifies fixed-point development by providing tools and workflows that help the conversion process. System engineers doing on-target rapid prototyping for fixed-point ECUs often benefit from automated scaling and workflow assistance to support their initial fixed-point design. Production software engineers benefit from automated scaling as well, but they also require fine grain control over fixed data specification in their modeling environment to work with accumulator word sizes and target-specific optimizations. In addition to providing automated scaling and fine grain data modeling features, Model-Based Design capabilities for fixed-point verification and validation continue to evolve. One example is bit-accurate, fixed-point simulation with automated comparison to embedded software results using processor-in-the-loop testing.

This paper presents Model-Based Design capabilities and tools that support development and verification of fixed-point ECU software used in mass production vehicles.

INTRODUCTION

Model-Based Design provides executable specifications, automatic code generation, and automated verification and validation tools. These technologies can be used to accelerate software development for any embedded system. Additional engineering effort is needed to produce the optimized designs and efficient code needed to satisfy the resource constraints of embedded microprocessors used in mass production. Thus, it is important to have processes and guidelines that yield optimal fixed-point code.

This paper presents recently developed technologies and industry best practices for developing fixed-point code using Model-Based Design.

The topics presented are:

- Developing the system model
- Preparing model and data for conversion
- Floating- to fixed-point conversion
- Generating optimized code
- Performing verification and validation

DEVELOPING THE SYSTEM MODEL

A case study based on a well-known fault tolerant fuel system demonstration model (shown in Figure 1) will help illustrate the development and verification of fixed-point ECU software using Model-Based Design. Because this model has been described in previous literature [1], it will not be detailed here. However an overview is provided to explain the model structure and purpose.

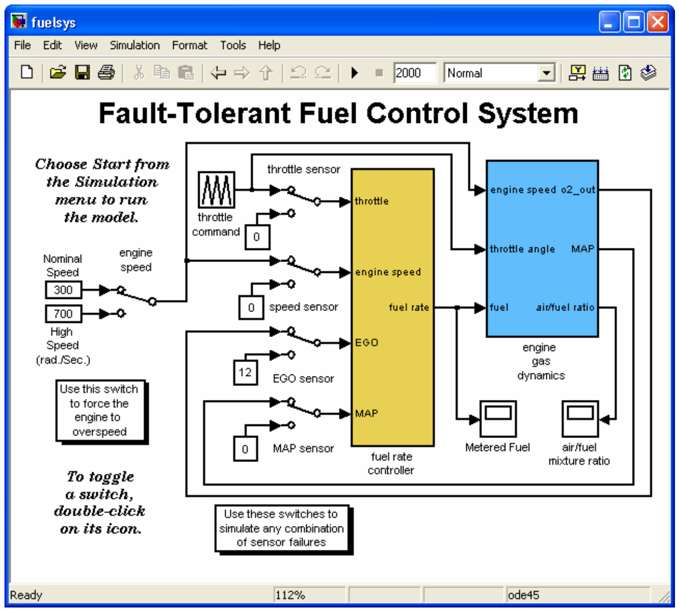


Figure 1: System model.

The fuel control system model has three main components: a controller model for the ECU, a plant model for the engine gas dynamics, and several sensors. The engine model consists of air-fuel intake dynamics comprising both the throttle body and intake manifold. The engine model has two inputs, engine speed and throttle angle; and three outputs, sensed oxygen, manifold pressure (MAP), and air-fuel ratio.

The plant model is developed using continuous time blocks. A 10 millisecond (10 ms) sample rate is required for the fuel rate control system algorithm.

A closed-loop simulation can be performed using the plant and controller, plus input stimulus and output scopes. The controller must be tolerant to sensor failure faults. Faults can be inserted using manual switches. By toggling the input signals to nominal or fault values, model developers can examine the effect of the fault on controller performance and assess the controller's tolerance to faults.

The control system consists of a state machine and block diagrams. The state machine detects faults and establishes a fueling mode. Block diagrams model the sensor correction and fault redundancy logic, intake airflow estimation and correction logic, and a fuel rate calculation that concludes with a software limit on the fuel rate output command. The control system model is shown in Figure 2.

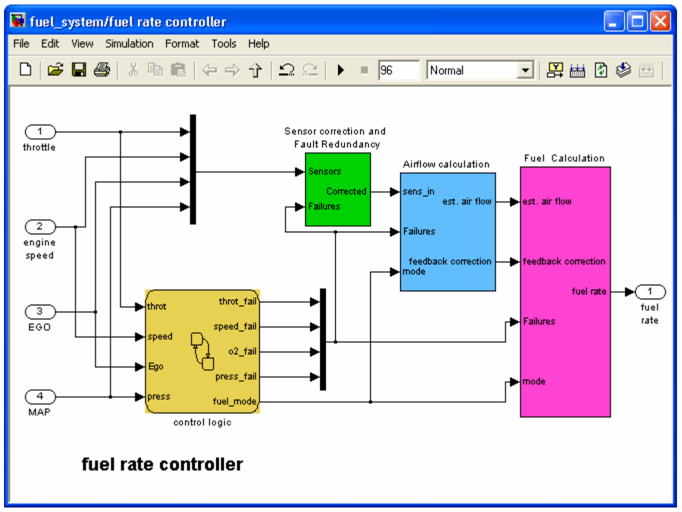


Figure 2: Control system model.

The control logic diagram has three states for each input sensor: warm up, normal, and failure. The criteria for transitioning between states are established via threshold parameters for each sensor. A sensor failure counter is used to track the total number of sensor failures.

This counter is then used to determine a fueling mode. A low fueling mode is used if there are no sensor failures; a rich fueling mode is used for one sensor failure; and a disabled mode is used for two or more failures. Figure 3 shows the main portions of the state machine.

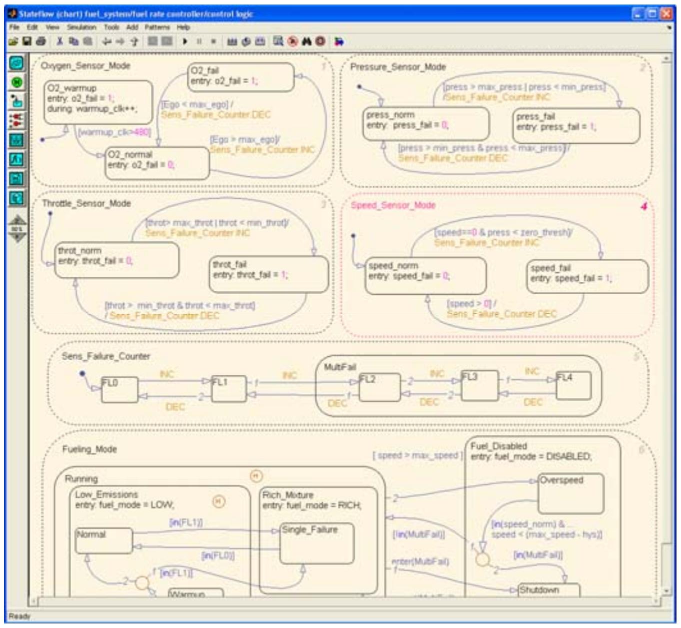


Figure 3: Control logic state machine.

The intake airflow estimation and correction logic shown in Figure 4 includes feedforward and feedback control algorithms. Table lookups provide pumping constants and rates.

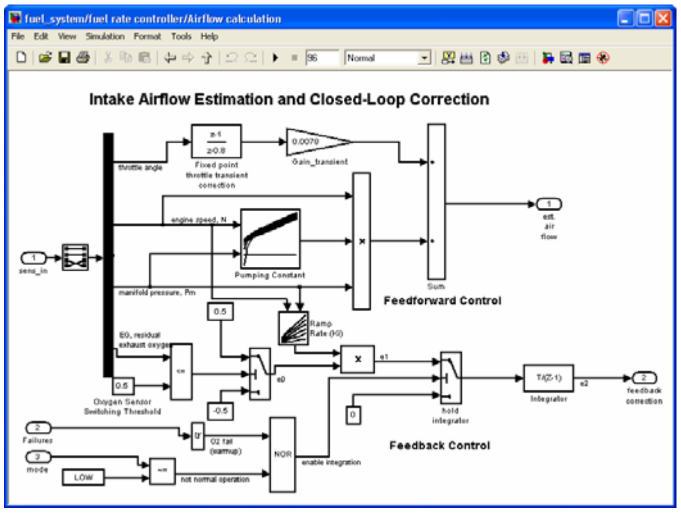


Figure 4: Closed-loop block diagram.

The system model is then simulated and time response results are collected for metered fuel and air/fuel ratio as shown in figure 5.

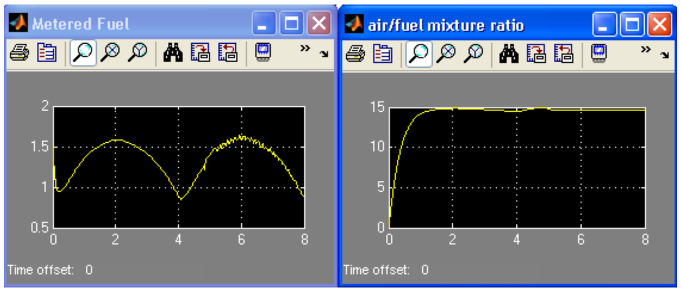


Figure 5: System response.

This fuel system model is typical of most models initially developed using Model-Based Design. These models address the behavioral or functional requirements but are not necessarily well suited for implementation on an embedded system for a variety of reasons, such as:

- Continuous time representation of blocks is used instead of discrete.
- Simulation results are calculated using double precision real data types instead of integer or fixed-point types.
- Interfaces between components such as the controller, plant, and sensors are not well specified or locked down.

Some organizations try to address these issues by forcing the system developer to consider implementation aspects during initial algorithm design. In others, system designers are free to explore and optimize behavioral designs using their preferred modeling style.

PREPARING MODEL AND DATA FOR CONVERSION

After a floating-point behavioral model is developed, it must be prepared for implementation on a fixed-point embedded microcontroller.

Some preparation tasks are needed even if the code is to be deployed code on a floating point processor. For example, the embedded system will run in discrete time, so continuous time blocks used for the embedded algorithm should be replaced by discrete blocks. It is possible to automate this process using conversion utilities. Rate transition blocks can also be used to convert the continuous-time signals to discrete-time signals sampled at 10ms, as required by the controller. Note that the effect of sampling on system performance and stability must also be analyzed.

Preparing a model for conversion to fixed-point math requires several steps outlined in the following sections.

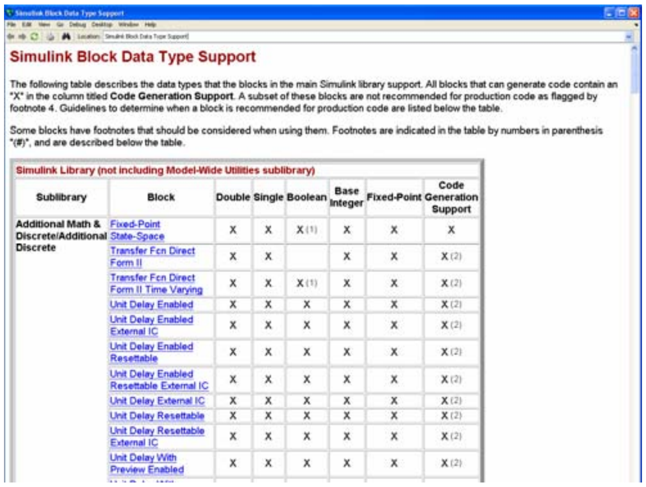
CREATE INITIAL REFERENCE DATA

Before beginning any model conversion task, create reference signal data for your floating point, behavioral model. These results can be used later for equivalence comparisons with the fixed-point model and generated code. The results shown in Figure 5, for example, will be used for equivalence testing of the fuel system controller.

REPLACE UNSUPPORTED BLOCKS

Identify and replace blocks that do not support fixed-point types. This includes replacing continuous-time with discrete-time blocks. Start by reviewing a list of data types supported by each block, as shown in Figure 6.

For models with Embedded MATLAB® functions, choose those that support fixed-point. There are hundreds of functions and blocks that support fixed-point implementation, including all the functions an engineer would typically use in embedded algorithm design.



Simulink Block Data Type Support

The following table describes the data types that the blocks in the main Simulink library support. All blocks that can generate code contain an "X" in the column titled **Code Generation Support**. A subset of these blocks are not recommended for production code as flagged by footnote 4. Guidelines to determine when a block is recommended for production code are listed below the table.

Some blocks have footnotes that should be considered when using them. Footnotes are indicated in the table by numbers in parenthesis "(#)", and are described below the table.

Simulink Library (not including Model-Wide Utilities sublibrary)

Sublibrary	Block	Double	Single	Boolean	Base Integer	Fixed-Point	Code Generation Support
Additional Math & Discrete/Additional Discrete	Fixed-Point State-Space	X	X	X (1)	X	X	X
	Transfer Fcn Direct Form II	X	X		X	X	X (2)
	Transfer Fcn Direct Form II Time Varying	X	X	X (1)	X	X	X (2)
	Unit Delay Enabled	X	X	X	X	X	X (2)
	Unit Delay Enabled External IC	X	X	X	X	X	X (2)
	Unit Delay Enabled Resettable	X	X	X	X	X	X (2)
	Unit Delay Enabled Resettable External IC	X	X	X	X	X	X (2)
	Unit Delay External IC	X	X	X	X	X	X (2)
	Unit Delay Resettable	X	X	X	X	X	X (2)
	Unit Delay Resettable External IC	X	X	X	X	X	X (2)
	Unit Delay With Preview Enabled	X	X	X	X	X	X (2)

Figure 6: Block data type support table.

SET UP SIGNAL LOGGING

Logging signals of interest during simulation is important because logged signals are used for analysis and comparison in other tasks. Model inputs and outputs are commonly logged (as was done with the initial reference data) but it may be helpful to log other signals to help with conversion to fixed-point.

Engineers no longer need to specifically add blocks or name signals to log data. It is now possible to log unnamed signals or log all data from a selected portion of the model subsystem hierarchy as shown in Figure 7.

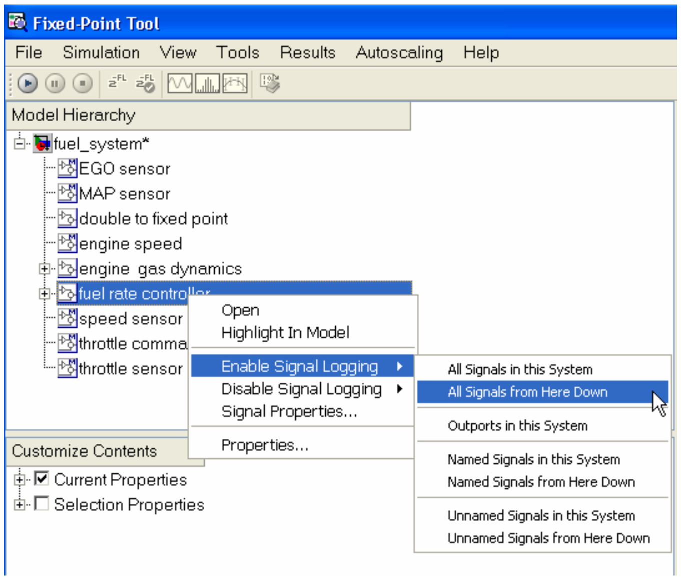


Figure 7: Logging fixed-point data.

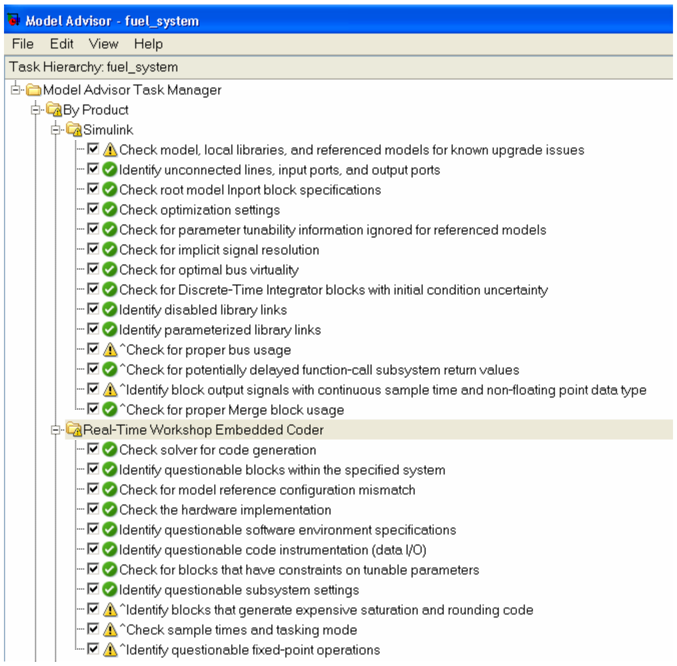
SPECIFY TARGET HARDWARE CHARACTERISTICS

The model simulation behavior and code generation outputs are determined by target hardware characteristics. Specifying the correct word lengths for char, int, long, and other attributes unique to a particular embedded microprocessor is needed to avoid producing incorrect results during simulation or code generation.

CHECK MODEL SUITABILITY FOR PRODUCTION

Automated model checks should be used to inspect the model's suitability for production code deployment as shown in Figure 8. These checks cover a broad range of topics such as model upgrades and library links.

Some checks, such as "identify questionable fixed-point operations" and "check hardware implementation", are crucial for fixed-point development. Additional checks can be run, including checks based on the updated MAAB guidelines [2] or safety related standards such as IEC 61508 [3].



Model Advisor - fuel_system
File Edit View Help
Task Hierarchy: fuel_system
Model Advisor Task Manager
By Product
Simulink

- Check model, local libraries, and referenced models for known upgrade issues
- Identify unconnected lines, input ports, and output ports
- Check root model Inport block specifications
- Check optimization settings
- Check for parameter tunability information ignored for referenced models
- Check for implicit signal resolution
- Check for optimal bus virtuality
- Check for Discrete-Time Integrator blocks with initial condition uncertainty
- Identify disabled library links
- Identify parameterized library links
- ^Check for proper bus usage
- ^Check for potentially delayed function-call subsystem return values
- ^Identify block output signals with continuous sample time and non-floating point data type
- ^Check for proper Merge block usage

Real-Time Workshop Embedded Coder

- Check solver for code generation
- Identify questionable blocks within the specified system
- Check for model reference configuration mismatch
- Check the hardware implementation
- Identify questionable software environment specifications
- Identify questionable code instrumentation (data I/O)
- Check for blocks that have constraints on tunable parameters
- Identify questionable subsystem settings
- ^Identify blocks that generate expensive saturation and rounding code
- ^Check sample times and tasking mode
- ^Identify questionable fixed-point operations

Figure 8: Model checks can determine a model's suitability for production.

CREATE SIMULATION REFERENCE DATA

If the model has changed or additional signals are logged, create new reference signal data for the floating-point model. These results will be used for automated scaling and equivalence comparisons with the fixed-point design and automatically generated code.

PREPARE FOR DATA TYPING AND SCALING

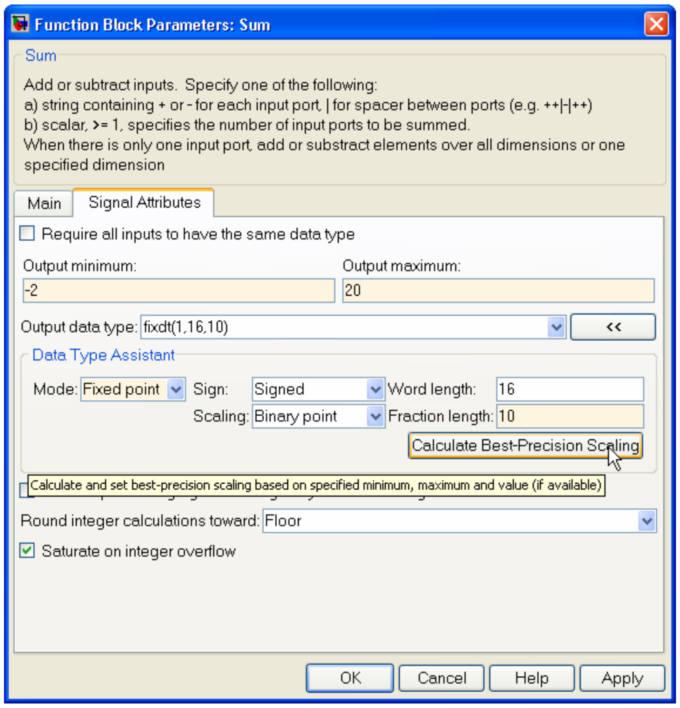
Some models are easier to convert than others. Differences in the level of effort required are often related to how blocks in the model were configured for data type inheritance and other propagation settings. During the initial design, heavy use of data propagation speeds prototyping and allows for fast analysis and design iterations. As the project moves closer to production, however, it may be helpful to do less automated propagation and more fine grain data design for the individual data types and scaling options.

To make models easier to convert:

- Remove output data type inheritance, especially in cases that lead to data type propagation conflicts.
- Relax input data type settings or constraints that might lead to data type propagation errors.
- Ensure state charts have strong data typing with Simulink®.
- Specify block minimum and maximum values for block output and parameter minimum and maximum values, if known.

FLOATING- TO FIXED-POINT CONVERSION

It is possible to examine each block and manually scale it according to the specified range and desired precision. The new data type assistant feature within the Block Parameter dialog box facilitates this approach by calculating appropriate fixed-point scaled data types based on specified minimum and maximum values. Figure 9 shows this dialog box for a Sum block.



Function Block Parameters: Sum

Sum

Add or subtract inputs. Specify one of the following:
a) string containing + or - for each input port, | for spacer between ports (e.g. ++|-|++)
b) scalar, >= 1, specifies the number of input ports to be summed.
When there is only one input port, add or substract elements over all dimensions or one specified dimension

Main | Signal Attributes

☐ Require all inputs to have the same data type

Output minimum: -2

Output maximum: 20

Output data type: fixdt(1,16,10)

Data Type Assistant

Mode: Fixed point

Sign: Signed

Word length: 16

Scaling: Binary point

Fraction length: 10

Calculate Best-Precision Scaling

Calculate and set best-precision scaling based on specified minimum, maximum and value (if available)

Round integer calculations toward: Floor

☑ Saturate on integer overflow

OK | Cancel | Help | Apply

Figure 9: Data Type Assistant for a Sum block.

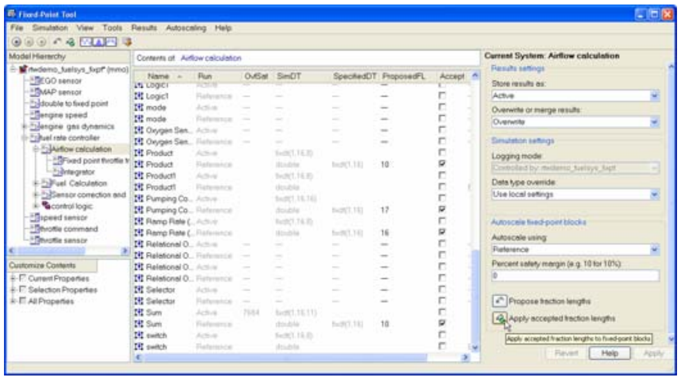
During the scaling process, in addition to the input and output signals, engineers should consider the intermediate calculations that use an accumulator. In previous releases, blocks used the user-specified output data type to perform all operations. In some cases, this behavior could cause precision loss and saturation during intermediate operations. In recent releases, the addition, subtraction, and summation blocks use an ideal accumulator data type based on the hardware characteristics when performing intermediate operations. Consequently, these blocks now produce more precise results and code is generated with less saturation checks.

Another approach to manual scaling uses the Fixed-Point Tool and its automatic scaling feature to convert from floating- to fixed-point. It is possible to lock down each block and prevent it from being modified by the auto scaling tool. This allows you to use automatic scaling in conjunction with individually scaled blocks. The autoscale function computes the scaling information based on the individually scaled blocks and reference data prepared for the floating-point model. Engineers can then accept or reject the proposed scaling for each signal.

It is also possible to perform data type override and compare double precision results to the scaled fixed-point results. This allows a project to use one model for both floating- and fixed-point design. Another technique to target one model for floating- or fixed-point designs is to substitute different data dictionaries as previously described [4].

In addition to comparison plots, the Fixed-Point Tool also records the number of overflows and saturations that occurred. Figure 10 shows the tool and its proposed scaling for the Fuel System model.

The Sum block is shown to have reached its overflow saturation check 7664 times during the simulation. Hence, the automatic scaler proposes to change the fraction length from 11 to 10 bits to provide the extra range needed with maximum precision.



Contents of: Airflow calculation

Name	Run	OvfSat	SimDT	SpecifiedDT	ProposedFL	Accept
Logic1	Reference	—	—	—	—	☐
mode	Active	—	—	—	—	☐
mode	Reference	—	—	—	—	☐
Oxygen Sen...	Active	—	—	—	—	☐
Oxygen Sen...	Reference	—	—	—	—	☐
Product	Active		fixdt(1,16,8)			☐
Product	Reference		double	fixdt(1,16)	10	☑
Product1	Active		fixdt(1,16,8)			☐
Product1	Reference		double			☐
Pumping Co...	Active		fixdt(1,16,16)			☐
Pumping Co...	Reference		double	fixdt(1,16)	17	☑
Ramp Rate (...	Active		fixdt(1,16,8)			☐
Ramp Rate (...	Reference		double	fixdt(1,16)	16	☑
Relational O...	Active	—	—	—	—	☐
Relational O...	Reference	—	—	—	—	☐
Relational O...	Active	—	—	—	—	☐
Relational O...	Reference	—	—	—	—	☐
Selector	Active	—	—	—	—	☐
Selector	Reference	—	—	—	—	☐
Sum	Active	7664	fixdt(1,16,11)			☐
Sum	Reference		double	fixdt(1,16)	10	☑
switch	Active		fixdt(1,16,8)			☐
switch	Reference		double			☐

Current System: Airflow calculation

Results settings

Store results as: Active

Overwrite or merge results: Overwrite

Simulation settings

Logging mode: Controlled by: rtwdemo_fuelsys_fxpt

Data type override: Use local settings

Autoscale fixed-point blocks

Autoscale using: Reference

Percent safety margin (e.g. 10 for 10%): 0

Propose fraction lengths

Apply accepted fraction lengths

Apply accepted fraction lengths to fixed-point blocks

Revert | Help | Apply

Figure 10: Fixed Point Tool for automatic scaling.

GENERATING OPTIMIZED CODE

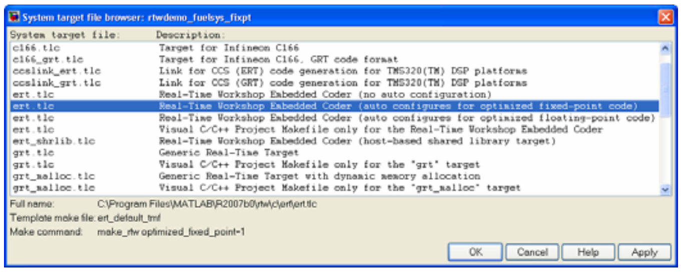
Prior to generating code, it is advisable to disable the signal logging used during the fixed point conversion. This will avoid declaring extra signal memory in the generated code.

Model Advisor checks for Real-Time Workshop® Embedded Coder software should also be run. This will check many code efficiency improvement areas such as:

Identifying blocks that generate expensive saturation and rounding code.
Identifying questionable fixed-point operations.
Inspecting lookup tables to ensure they are properly spaced for maximizing code efficiency.

Producing code from an optimized design is a straightforward step of selecting a deployment target and generating code. The production code targeting options range from the default ANSI/ISO C, to target optimized algorithm code, to a highly customized deployment target that includes calls to device drivers. It is also possible to target middleware and abstraction layers such as AUTOSAR™.

For ANSI/ISO C, an Embedded Real-Time Target (ERT) option exists that is optimized for fixed-point code as shown in Figure 11.


System target file browser: rtwdemo_fuelsys_fixpt

System target file:	Description:
c166.tlc	Target for Infineon C166
c166_grt.tlc	Target for Infineon C166, GRT code format
ccslink_ert.tlc	Link for CCS (ERT) code generation for TMS320(TM) DSP platforms
ccslink_grt.tlc	Link for CCS (GRT) code generation for TMS320(TM) DSP platforms
ert.tlc	Real-Time Workshop Embedded Coder (no auto configuration)
ert.tlc	Real-Time Workshop Embedded Coder (auto configures for optimized fixed-point code)
ert.tlc	Real-Time Workshop Embedded Coder (auto configures for optimized floating-point code)
ert.tlc	Visual C/C++ Project Makefile only for the Real-Time Workshop Embedded Coder
ert_shrlib.tlc	Real-Time Workshop Embedded Coder (host-based shared library target)
grt.tlc	Generic Real-Time Target
grt.tlc	Visual C/C++ Project Makefile only for the "grt" target
grt_malloc.tlc	Generic Real-Time Target with dynamic memory allocation
grt_malloc.tlc	Visual C/C++ Project Makefile only for the "grt_malloc" target

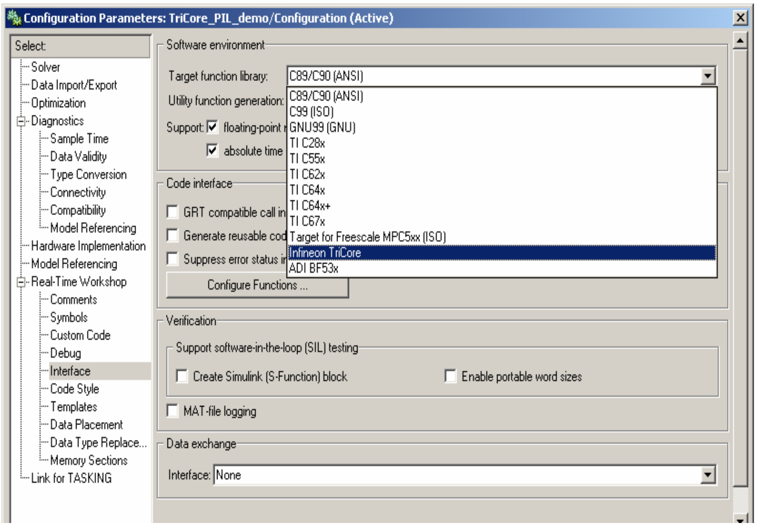
Full name: C:\Program Files\MATLAB\R2007b0\rtw\c\ert\ert.tlc
Template make file: ert_default_tmf
Make command: make_rtw optimized_fixed_point=1

Figure 11: ANSI-C optimized fixed-point target using Real-Time Workshop® Embedded Coder™.

Other than word sizes and other target characteristic settings, this code is portable and can be deployed on any target with the specified word sizes.

For target optimized code, a number of options exist. The first is to have the generated code call an existing C function at the appropriate point within the algorithm. One method A second option is to use Legacy Code Tool in Simulink.

Another newer option is to replace generated code math functions and operators with target-specific versions. This is done using Target Function Libraries (TFL). TFL requires the end user to create a table mapping default functions and operators to their target specific equivalents. The TFL is then available as a code generation option. Figure 12 shows the Infineon® TriCore® TFL as an example.

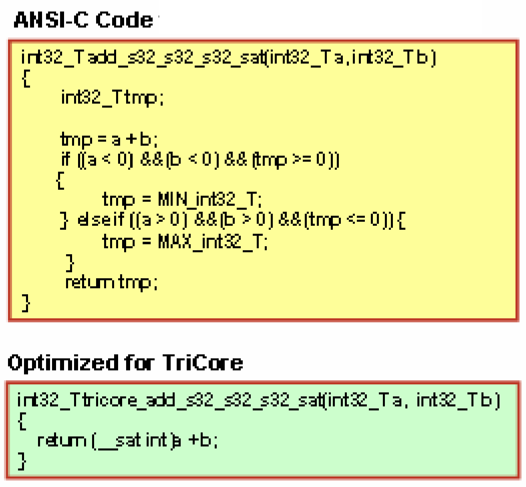

Configuration Parameters: TriCore_PIL_demo/Configuration (Active)

Software environment
Target function library: C89/C90 (ANSI)
Utility function generation:
Support: ☑ floating-point
☑ absolute time
Code interface
☐ GRT compatible call in
☐ Generate reusable cod
☐ Suppress error status i
Verification
Support software-in-the-loop (SIL) testing
☐ Create Simulink (S-Function) block
☐ Enable portable word sizes
☐ MAT-file logging
Data exchange
Interface: None

Figure 12: Selecting optimized Target Function Libraries.

Once a TFL is selected, the generated code will incorporate the replacement items seamlessly. See Figure 13 for a comparison of ANSI-C and TriCore optimized code for fixed-point add of 32-bit integers with saturation logic. Not only is the code smaller, the execution time also decreases dramatically.

In one example, the optimized code ran faster than ANSI-C code by a factor of 17. Of course, this code is no longer portable. However, it is easy to change the TFL choice in your model to target another device.


ANSI-C Code

```
int32_T add_s32_s32_s32_sat(int32_T a, int32_T b)
{
    int32_T tmp;

    tmp = a + b;
    if ((a < 0) && (b < 0) && (tmp >= 0))
    {
        tmp = MIN_int32_T;
    } else if ((a > 0) && (b > 0) && (tmp <= 0)) {
        tmp = MAX_int32_T;
    }
    return tmp;
}
```

Optimized for TriCore

```
int32_T tricore_add_s32_s32_s32_sat(int32_T a, int32_T b)
{
   return (__sat int)a + b;
}
```

Figure 13: ANSI-C and TriCore optimized fixed-point code using TFL

The final deployment option involves building a complete application involving algorithms, device drivers, and operating system software. Several options exist for this process: using The MathWorks™ target products, using third-party commercial target products, or building a custom target with documented procedures and APIs.

Commercially available compiler tool chain and processor target support packages [6] should be reviewed before building a custom target.

PERFORMING VERIFICATION AND VALIDATION

The reference data collected from the floating-point behavioral model can be reused for equivalence testing throughout the development process. It is first used to compare the results of fixed-point design to the original floating-point model. The comparison is done frequently and it is important to note that code generation is not needed during this step since bit-accurate fixed-point simulation is provided by Simulink® Fixed Point™.

The next use may be in software-in-the-loop (SIL) testing. This occurs on the host computer and makes it easy for the generated code to run with the original plant model or test harness. The generated code may include legacy code. Target-specific code, however, cannot be tested since it cannot execute on the host computer.

Testing code on the target processor is done using processor-in-the-loop (PIL) testing. PIL testing co-simulates the code (in this case actual object code) on an Instruction Set Simulator or embedded hardware with the original plant model or test harness in Simulink.

The MathWorks offers a variety of link products that automate PIL testing using commercial Integrated Development Environments (IDEs). It is possible to run PIL testing on processors supported by these IDEs.

Model verification using analysis instead of simulation can also be done using Simulink® Design Verifier™ software, which automatically generates test cases based on structural coverage criteria and enables formal proofs and analysis.

Code verification through analysis instead of simulation can also be done using Polyspace™ products, which formally analyze code to identify common code defects such as fixed-point overflow, divide-by-zero, and array out of bounds checks.

CONCLUSION

Model-Based Design with automatic code generation is increasingly important advantageous in automotive software development. Advances in fixed-point tools are occurring rapidly. Software engineers need to stay current on the technology and understand how to best apply it for mass production environments. This paper provided an update on these topics.

REFERENCES

1. "A Seamless Implementation of Model-Based Design Applied to a New Fuel Control Feature for an Existing Engine ECU", by Thomas Erkkinen, The MathWorks, and Koos Zwaanenburg, ETAS, SAE Technical Paper No. 2006-01-0612, April 2006.
2. "Controller Style Guidelines for Production Intent Using MATLAB®, Simulink® and Stateflow® - Version 2," MathWorks Automotive Advisory Board (MAAB), dated July 2007, www.mathworks.com/industries/auto/maab.html
3. IEC 61508-3:1998. International Standard IEC 61508 Functional safety of electrical/electronic/ programmable electronic safety-related systems – Part 3: Software requirements. 1st edition,1998
4. "Multi-Target Modeling for Embedded Software Development for Automotive Applications", by Grantley Hodge, et al, Visteon Corp, SAE Technical Paper No. 2004-01-0269, March 2004, www.visteon.com/whitepapers/2004_01_0269.pdf
5. "Fixed-Point Modeling and Code Generation Tips", by Vinod Reddy, Siva Nadarajah and George Beals, MATLAB Central, dated 2008 www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=7197&objectType=file
6. Real-Time Workshop® Embedded Coder Supported Hardware, by The MathWorks, http://www.mathworks.com/products/rtwembedded/supportedio.html

CONTACT

Tom Erkkinen, Embedded Applications Manager, The MathWorks, Inc.

Tom leads a MathWorks initiative to foster industry adoption of production code generation technologies. Before joining The MathWorks, he worked at Lockheed-Martin and NASA developing a variety of control algorithms and real-time software. Tom holds a B.S. degree in Aerospace Engineering from Boston University and an M.S. degree in Mechanical Engineering from Santa Clara University.

tom.erkkinen@mathworks.com

POST SAE UPDATE (May 2008): Release 2008a of Simulink Fixed Point includes the Fixed Point Advisor that automates many of the float- to fixed-point conversion tasks described herein.