Converting Models from Floating Point to Fixed Point for Production Code Generation

By Bill Chou and Tom Erkkinen

An essential step in embedded software development, floating- to fixed-point conversion can be tedious, labor-intensive, and error-prone. System engineers frequently design algorithms in floating-point math, usually double-precision. This format represents the ideal algorithm behavior but takes little account of the algorithm's final realization in production software and hardware. Software engineers and suppliers in mass production environments often need to convert these algorithms to fixed-point math for their integer-only hardware. As a result, multiple iterations between system and software engineers are often required.

Products Used

- Simulink®
- Stateflow®
- Fixed-Point Toolbox™
- Simulink Fixed Point™
- Simulink Verification and Validation™
- Real-Time Workshop®
- Real-Time Workshop Embedded Coder™
- Stateflow Coder™

Using a fault-tolerant fuel system model as an example, this article describes tools and a workflow for converting models from floating point to fixed point for production code generation. Topics covered include:

- Preparing the model data
- Analyzing, refining, and optimizing the fixed-point scaling
- Generating optimized code
- Verifying and validating the code

The approach described here uses two tools in Simulink Fixed Point™: Fixed-Point Advisor and Fixed-Point Tool.

Preparing the Model And Data For Conversion

The fault-tolerant fuel system model contains three main components: an ECU controller, a plant for the engine gas dynamics, and several sensors. We concentrate on the ECU controller for fixed-point modeling, conversion, and code generation.

Some preparation tasks are required even if the code generated from the ECU is to be deployed on a floating-point embedded micro-processor. Fixed-Point Advisor is an interactive tool designed to facilitate model preparation (Figure 1).

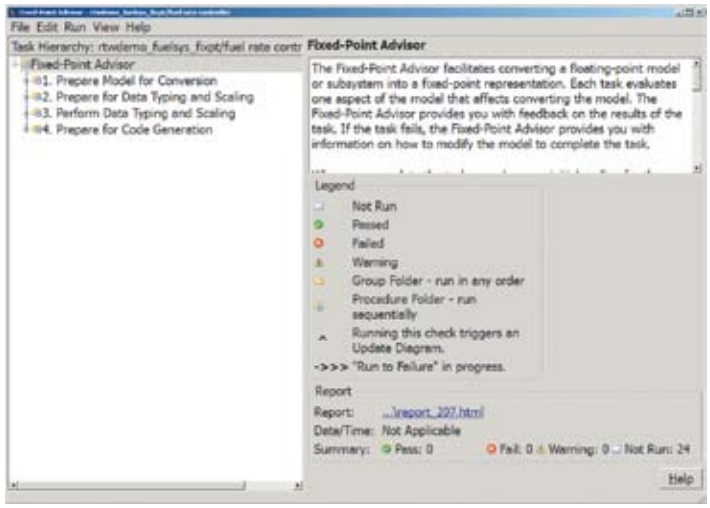

Figure 1. Model preparation using Fixed-Point Advisor.

Sublibrary	Block	Double	Single	Boolean	Base Integer	Fixed-Point	Code Generation Support
Simulink/Logic and Bit Operations	Bit Clear			X (7)	X	X	X
	Bit Set			X (7)	X	X	X
	Bitwise Operator			X (7)	X	X	X
	Combinatorial Logic	X		X			X
	Compare To Constant	X	X	X	X	X	X
	Compare To Zero	X	X	X	X	X	X
	Detect Change	X	X	X	X	X	X (2)
	Detect Decrease	X	X	X	X	X	X (2)
	Detect Increase	X	X	X	X	X	X (2)
	Detect Fall Negative	X	X	X	X	X	X (2)
	Detect Fall Nonpositive	X	X	X	X	X	X (2)
	Detect Rise Nonnegative	X	X	X	X	X	X (2)
	Detect Rise Positive	X	X	X	X	X	X (2)
	Extract Bits	X	X	X	X	X	X
	Interval Test	X	X	X	X	X	X
	Interval Test Dynamic	X	X	X	X	X	X
	Logical Operator	X	X	X	X	X	X
	Relational Operator	X	X	X	X	X	X
	Shift Arithmetic	X	X		X	X	X

Figure 2. Block data type support table in Simulink.

Using Fixed-Point Advisor, we will complete the following steps:

- Replace unsupported blocks
- Set up signal logging and create initial reference data
- Specify target hardware characteristics
- Prepare for and perform data typing and scaling
- Check model suitability for production code deployment

Replacing Unsupported Blocks

We begin by replacing blocks that do not support fixed-point data types—including replacing continuous-time with discrete-time blocks. We can use rate transition blocks to convert the continuous-time signals to discrete-time signals sampled at 10ms, as required by the controller. At this stage, we must also analyze the effect of sampling and quantization on system performance and stability. Simulink® provides a list of data types supported by each block (Figure 2). Most blocks support fixed-point data types. Stateflow® fully supports fixed point.

If your model includes Embedded MATLAB™ functions, you can choose from hundreds of functions that support fixed point, including those typically used in embedded algorithm design.

Setting Up Signal Logging And Creating Initial Reference Data

We log signals of interest from simulation for use in equivalence comparisons with the fixed-point model and for code generation. Typically, input and output signals are logged. Fixed-Point Advisor provides a list of additional signals that it might be helpful to log.

To make signal logging easier, we can log all signals or select signals, including unnamed ones, from portions of the model subsystem hierarchy (Figure 3). Once we have configured the signals to be logged, we create and store the reference data using the floating-point model.

Specifying Target Hardware Characteristics

Model simulation behavior and code generation outputs are determined by the characteristics of the target hardware. Model Advisor reminds us to specify the correct word lengths for `char`, `int`, `long`, and other attributes unique to a particular embedded microprocessor so as to avoid producing incorrect results from simulation or code generation.

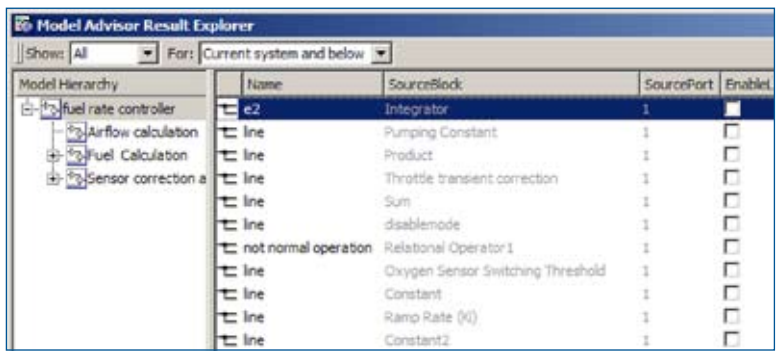

Figure 3. Logging fixed-point data, including unnamed signals.

Preparing For Data Typing And Scaling

Data type inheritance and other propagation settings affect the time needed to convert the model from floating to fixed point. During the initial design phase, engineers frequently use inherited data type propagation to speed up prototyping and to quickly iterate several designs. As the project approaches production, they fine-tune and specify individual data types and scaling to optimize fixed-point results.

Fixed-Point Advisor facilitates this workflow by automating the following steps:

- Removing output data type inheritance to help avoid data type propagation conflicts
- Relaxing input data type settings or constraints that might lead to data-type propagation errors
- Verifying that state charts have strong data typing with Simulink

We must specify design minimum and maximum values for inport blocks. We can also specify these values for other block outputs and parameters.

Performing Initial Data Typing And Scaling

Using Fixed-Point Advisor, we specify initial data typing and scaling for blocks.

Based on our initial input and direction, Fixed-Point Advisor proposes data typing and initial scaling for inport blocks, constants, blocks that do not fall into either category, parameters, and blocks that use intermediate data types, such as the Sum and Product blocks. Fixed-Point Advisor uses either the design or simulation minimum and maximum from the floating-point data to propose the initial fixed-point scaling. The tool reports scaling conflicts and suggests ways to resolve them. It then checks for numerical errors and analyzes the logged signals.

An initially scaled fixed-point model is then produced, together with plots comparing the floating-to-fixed-point model results.

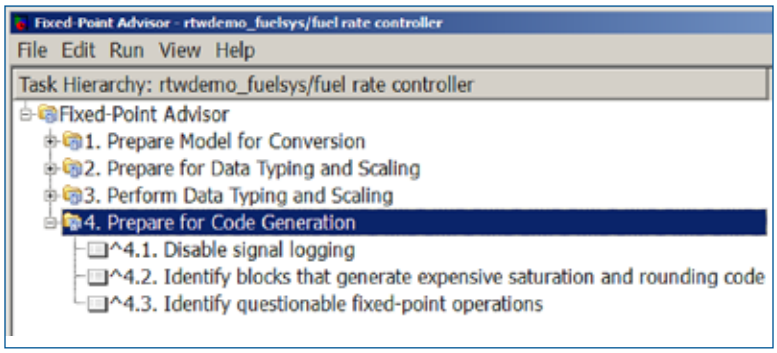

Figure 4. Identifying issues that could lead to the generation of inefficient code.

Checking the Model's Suitability For Production Code Deployment

Using Fixed-Point Advisor, we run final checks to determine the model's suitability for production code deployment (Figure 4). These include:

- Disabling signal logging to avoid declaring extra signal memory in the generated code
- Identifying blocks that generate expensive saturation and rounding code
- Identifying questionable fixed-point operations—for example, ensuring that lookup tables are spaced so as to maximize code efficiency

Analyzing, Refining, and Optimizing the Fixed-Point Scaling

Using the Fixed-Point Tool's automatic scaling function, we analyze, refine, and optimize scaling for relevant blocks in the model that we initially scaled using Fixed-Point Advisor.

We use the data type override feature to collect the dynamic range of signals in double precision. The Fixed-Point Tool uses this information to propose a more suitable fixed-point scaling for each block, based on the number of available bits. Individual blocks can be locked down to prevent them from being modified by the tool. We can then use automatic scaling with individually scaled blocks and accept or reject the proposed scaling for each signal.

With the Fixed-Point Tool we can use one model for both the floating- and fixed-point designs, reducing the need to maintain separate models during design iterations.

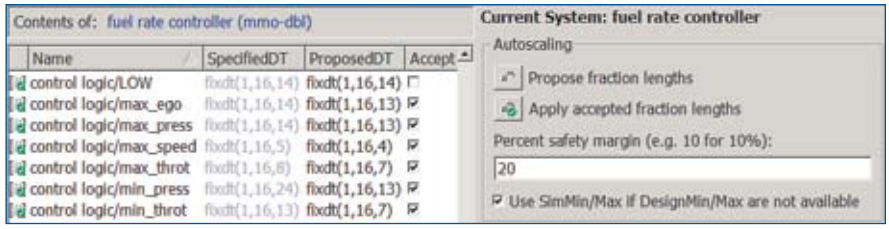

Figure 5. Automated scaling using the Fixed-Point Tool.

Comparison plots provide a quick and easy way to analyze and contrast the behavior of the fixed- and floating-point designs. The Fixed-Point Tool records the number of overflows and saturations that occurred in the simulation. Figure 5 shows the tool and its proposed scaling for the Fuel System model. Further analysis showed the output of the Sum block saturated during simulation. The automatic scaler proposed changing the fraction length from 11 to 10 bits, increasing the output dynamic range to avoid saturation and retain maximum precision. Once we have made this change, the results match closely (Figure 6), and we are ready for code generation.

Generating Optimized Code

Before generating code for the system, we run checks in Simulink using Model Advisor. Some checks, such as "identify questionable fixed-point operations" and "check hardware implementation," are crucial for fixed-point development. Additional model standards checks are available in Simulink Verification and Validation™, including checks based on the MAAB guidelines and safety-related standards such as IEC 61508 and DO-178B.

To generate code from an optimized design, we must first select a deployment target. Options range from the default ANSI/ISO C/C++ to a target with processor-optimized code. We could also target middleware or abstraction layers, such as AUTOSAR.

For ANSI/ISO C code generation, we select the Embedded Real-Time Target (ERT) option in Real-Time Workshop Embedded Coder™, which is optimized for fixed-point code (Figure 7). 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.

There are several capabilities for generating target-optimized code. The first is to have the generated code call an existing C function at the appropriate point within the algorithm, typically by using the Legacy Code Tool in Simulink.

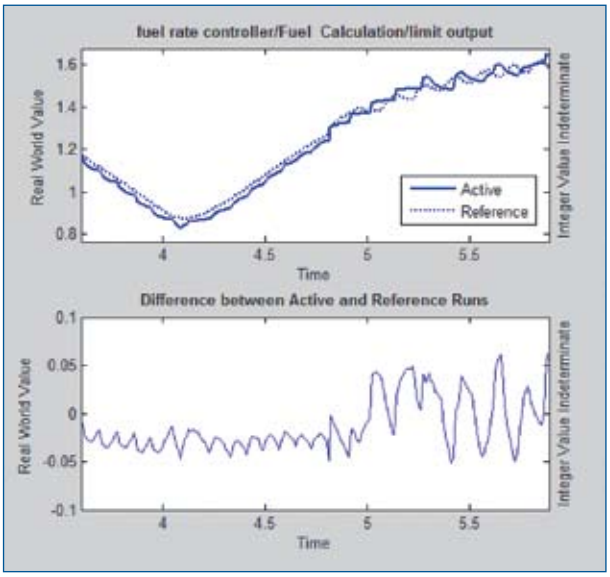

Figure 6. Comparison with original floating-point results.

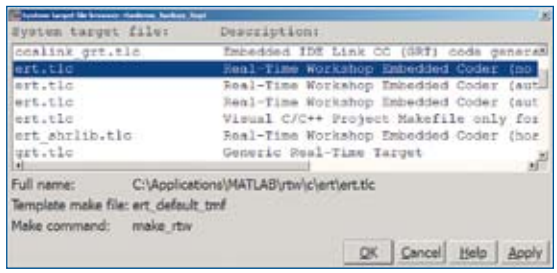

Figure 7. ANSI-C code optimized for a fixed-point target using Real-Time Workshop Embedded Coder.

A second capability is to automatically replace generated math functions, math operators, and memory-allocation functions such as `memcpy` with target-specific versions. This is done using Target Function Libraries (TFL), tables mapping default functions and operators to their target-specific equivalents. Several pre-built TFLs are available (Figure 8). In addition, end users can create customized tables for their own targets. The TFL is then available as a code-generation setting.

Once a TFL is selected, the generated code incorporates the replacement items. One advantage of this option is that you can quickly generate optimized code for several different targets from the same model simply by changing the TFL. Figure 9 compares ANSI-C- and TriCore® optimized code for fixed-point addition of 32-bit integers with saturation logic. The code is smaller and the execution time dramatically reduced—by a factor of 17.

Verifying and Validating the Production Code

The reference data collected from the floating-point behavioral model can be reused for equivalence testing throughout the development process. We first use the data to compare the results of the initial fixed-point design to the original floating-point model. We do not need to generate code just to compare the model results because bit-accurate fixed-point simulation is supported by Simulink Fixed Point.

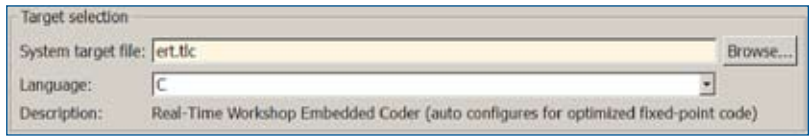

Figure 8. Selecting optimized Target Function Libraries.

```
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;
      } elseif ((a > 0) && (b > 0) && (tmp <= 0))
      {
        tmp = MAX_int32_T;
      }
      return tmp;
}
```

```
int32_Tadd_s32_s32_s32_sat(int32_T a, int32_T b)
{
      return (_sat int)a+b;
}
```

Figure 9. Fixed-point code optimized for ANSI-C (top) and TriCore (bottom) using TFL.

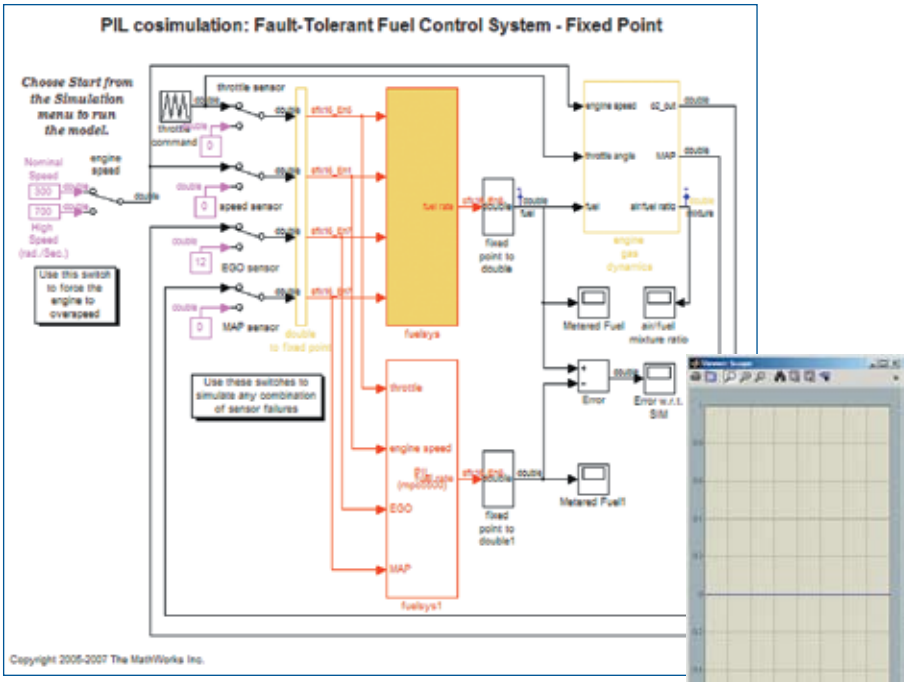

Figure 10. PIL testing. The scope (right) shows no difference between the signal output from the model and the generated object code using the PIL block.

We test the code on the target processor using processor-in-the-loop (PIL) testing (Figure 10). PIL cosimulates the object code on an embedded hardware or instruction set simulator with the original plant model or test harness in Simulink. MathWorks link products, such as Embedded IDE Link™ MU (for Green Hills® MULTI®), automate PIL testing using third-party integrated development environments (IDEs). It is possible to run PIL testing on processors supported by these IDEs, such as Freescale™ MPC 5500.

Another way to verify the code is to use Polyspace™ products, which formally analyze code to identify defects such as fixed-point overflow, division-by-zero, and array out-of-bounds.

Bit-accurate fixed-point simulation helps you model designs within the Simulink environment. Tools provided by Simulink Fixed Point let you automate time-consuming parts of the fixed-point conversion workflow and explore designs to further refine the fixed-point performance. Real-Time Workshop Embedded Coder provides an automated path to production code deployment. ■

Resources

VISIT
www.mathworks.com

TECHNICAL SUPPORT
www.mathworks.com/support

ONLINE USER COMMUNITY
www.mathworks.com/matlabcentral

DEMOS
www.mathworks.com/demos

TRAINING SERVICES
www.mathworks.com/training

THIRD-PARTY PRODUCTS AND SERVICES
www.mathworks.com/connections

Worldwide CONTACTS
www.mathworks.com/contact

E-MAIL
info@mathworks.com

80368V00 11/08

For More Information

- Using Diagnostics to Improve Your Simulink Model
  www.mathworks.com/diagnostics
- Fixed-Point Modeling and Code Generation Tips for Simulink 7 (R2008a)
  www.mathworks.com/simulink7-tips