Tips for Fixed-Point Modeling and Code Generation

By Siva Nadarajah (Senior Consultant, The MathWorks)
and George Beals (Senior Technical Writer, The MathWorks)

Introduction

This article provides tips on performing fixed-point modeling and generating code from such modeling.

Fixed-point technology itself brings with it unique difficulties and challenges. These tips are intended to help you:

- Reduce development time and therefore reduced cost
- Reduce ROM, RAM, and execution time requirements of the generated code

Fixed-Point Summary

Fixed-point representation allows you to express a real number as an integer by specifying its word size (in bits) and location of the binary point, as desired. See the example below, which represents the base-10 number +6.5 in fixed-point notation as an eight-bit data type.

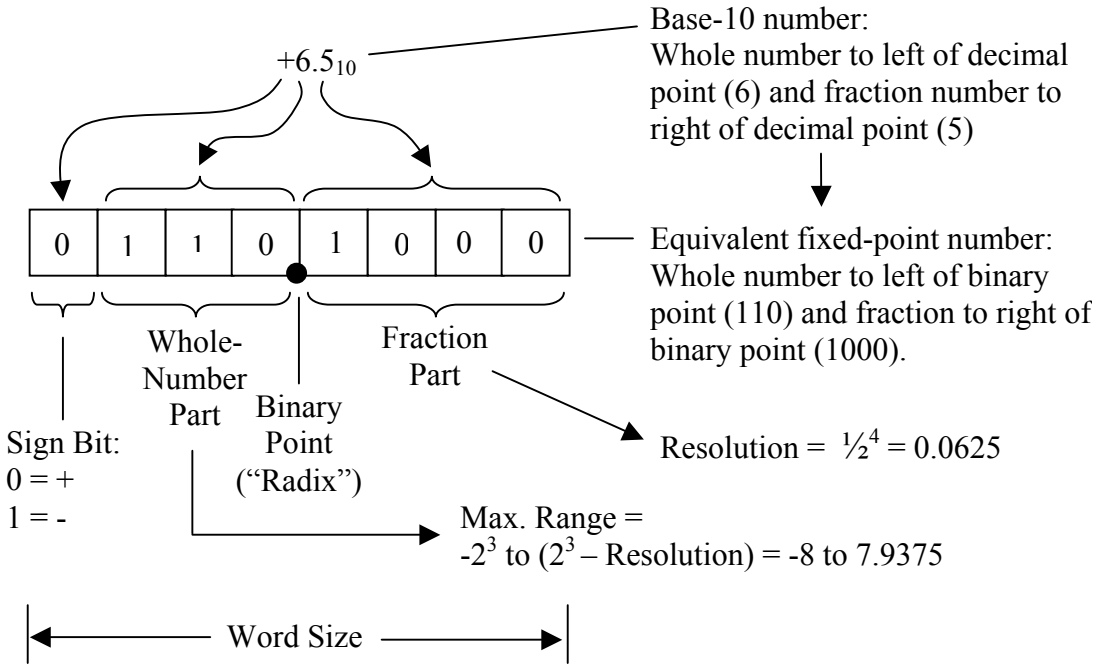

Figure 1: Conversion of base-10 to fixed-point.

Converting the whole-number part of the decimal to fixed-point is more straightforward than converting the fraction part. Here is how to convert an entire decimal number to its equivalent fixed-point number:

- The whole-number part of the fixed-point number = the binary equivalent of the decimal whole number. In our example, the 6 converts to 110.

- The fraction-part of the fixed-point number = the binary equivalent of (the decimal fraction divided by the resolution). And, resolution is $1/2^E$, where E is the number of bits to the right of the binary point. In our example, resolution is $1/2^4 = 0.0625$. Therefore, the fraction-part of the fixed-point number = the binary equivalent of 0.5/0.0625 = 1000.

The fraction determines the *resolution*, which is the smallest non-zero value that the fixed-point number can represent. (In MathWorks documentation, resolution is called *precision*.) The whole number determines the maximum *range*, namely -2^x to (2^x – resolution), where x is the number of bits to the left of the binary point (minus any sign bit). It is important to note that changing the location of the binary point in a fixed-point number causes a tradeoff between range and resolution.

Additional terms in fixed-point contexts are *scaling*, *bias*, and *slope*. The slope and bias together make up the scaling of a fixed-point number. The location of the binary point changes scaling. Think of the familiar y = mx+b, where m is slope and b is bias. The terminology used for this in the MathWorks Fixed-Point Blockset documentation is V = S*Q + B, where Q is "quantized fixed-point value" or "stored integer," V is "real-world" (that is, base-10) value, S is the user-specified slope, and B is the user-specified bias.

Related Dialog Boxes and Documentation

The two figures shown next describe the Simulink and Stateflow dialogs that allow you to specify fixed-point settings. For information on how to navigate to these dialogs, and for additional information on MathWorks fixed-point usage, see the documentation that accompanies the MathWorks products you are using.

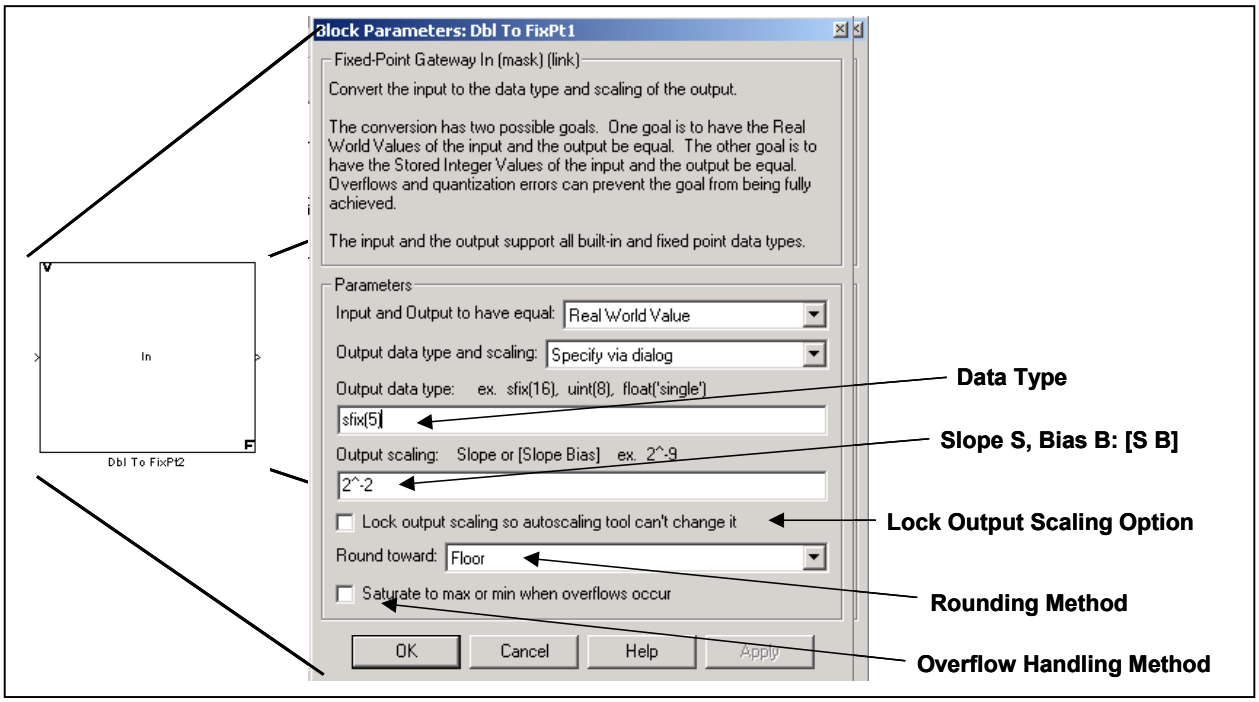

Figure 2: Simulink dialog for fixed-point settings.

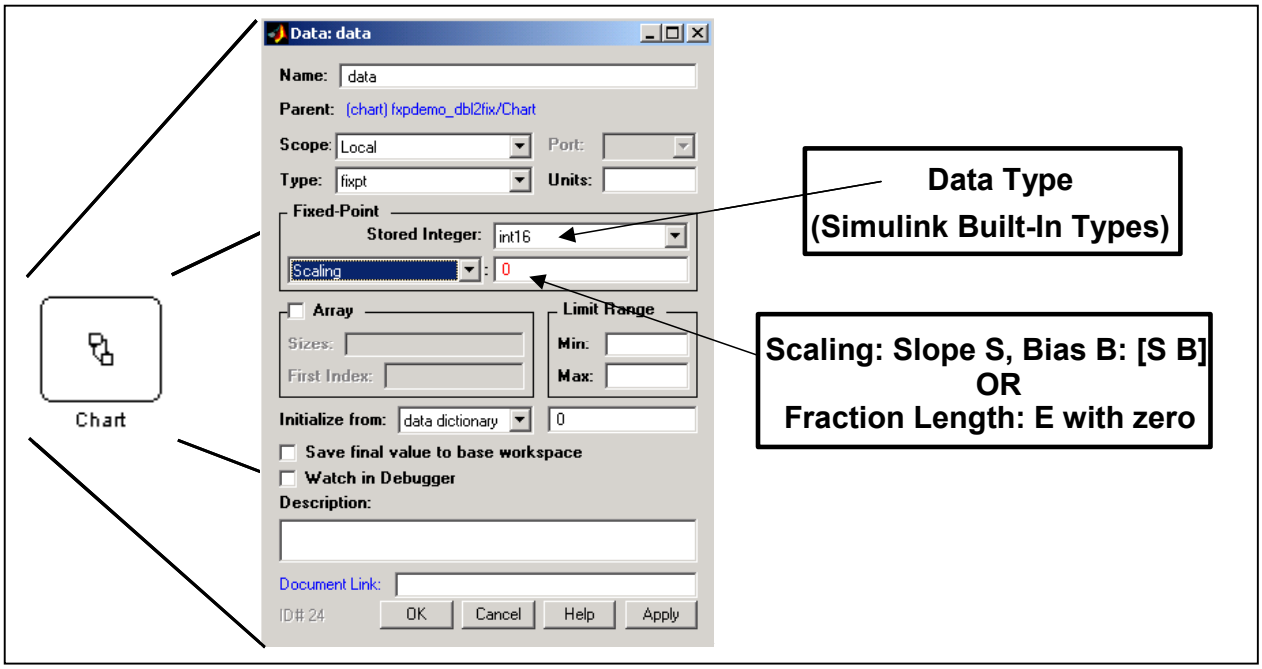

Figure 3: Stateflow dialog for fixed-point settings.

Problems

For the most part, using fixed-point numbers in modeling has more advantages than using floating-point, as seen in the comparison below:

Consideration	Fixed Point	Floating Point
RAM and ROM consumption	Small	Large
Execution time	Fixed-point is faster than floating-point.	
Word size and scaling	Flexible	Inflexible
Development time	Long	Short
Susceptibility to errors	Fixed-point is more prone to errors than floating-point.	
Availability and cost of hardware	Available in every microcontroller, and inexpensive	Available only in high-level (that is, feature-rich) microcontrollers, and expensive

However, there are some problems with using fixed-point numbers.

- You must design fixed-point slope S and bias B so that Q relates to V with reasonable range and resolution for the application. This is illustrated in the following graphs.

Note: The fixed-point number represented in the following graphs is sfix5_En2 data type, range of –4 to 3.75, and a resolution of 0.25.

On this and the following graphs, the top (sine wave) waveform is V. The other waveform is Q. The y-axis is the value of Q or V. The x-axis is time.

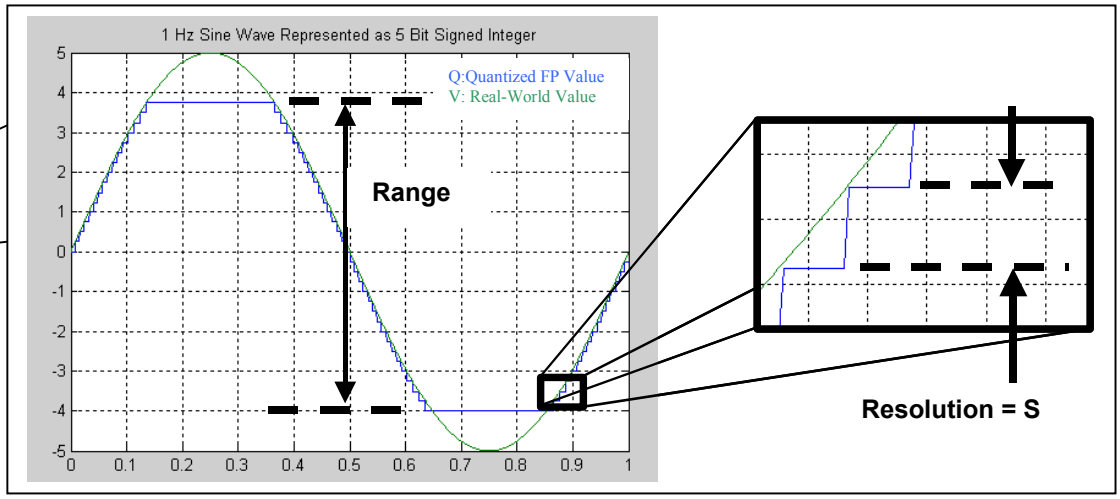

Figure 4: Example of range and resolution.

Note: The example above shows a five-bit integer for illustration purposes only. A more typical example is a 16- or 32-bit integer.

- You must adjust bias B to account for fixed biases between Q and V, which affects the range.

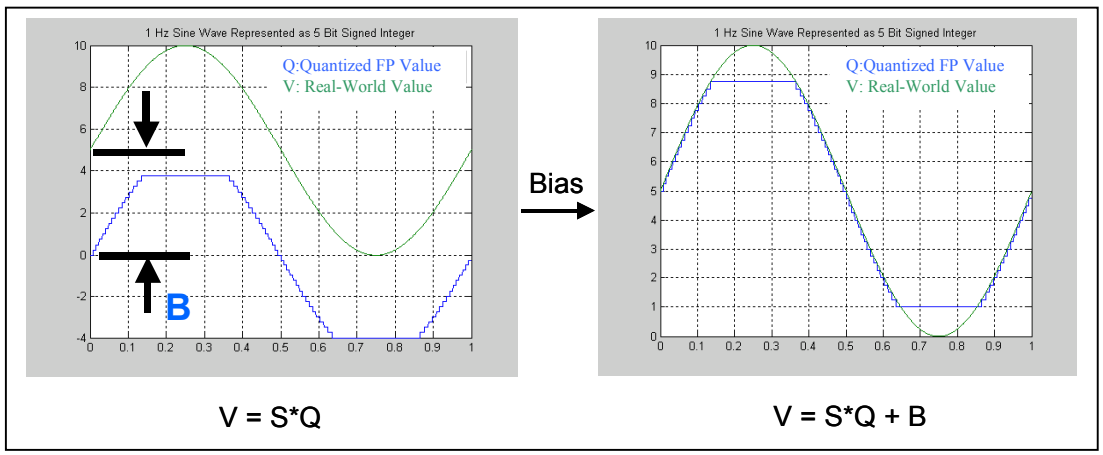

Figure 5: Adjusting bias.

- You must choose a rounding method. When the target microcontroller converts a fixed-point number from a higher resolution to a lower resolution, there are additional bits to deal with. If it throws them away, the conversion can yield imprecise results. This is what occurs when you select the "floor" rounding method. However, floor is the simplest, most common, and most efficient of the rounding methods. Alternatively, you can select a rounding method to preserve the extra bits for more accuracy, namely, the more complex but less efficient "zero," "nearest," and "ceiling" methods.

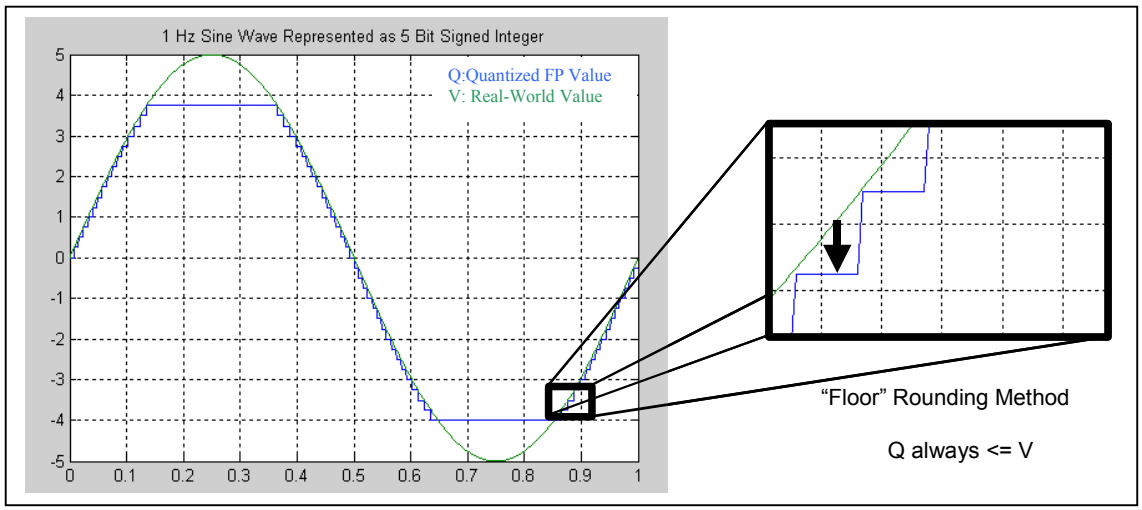

Figure 6: Example of floor rounding method.

Below is a comparison of results from using different rounding methods for a resolution change from 0.125 to 0.5:

Value of Q with Resolution of 0.125	Value of Q with Resolution of 0.5 after Indicated Rounding Method*			
	Floor (- inf)	Ceiling (+ inf)	Zero (0)	Nearest
2.125	2.0	2.5	2.0	2.0
-2.125	-2.5`	-2.0	-2.0	-2.0
1.875	1.5	2.0	1.5	2.0
-1.875	-2.0	-1.5	-1.5	-2.0

* Terms in parentheses (-infinity, +infinity, and 0) indicate that towards which the rounding is taking place.

- You must choose an overflow handling method. “Wrap” and “saturate” are available. Overflow occurs when the number to be stored as a result of a mathematical operation exceeds the number of bits the resulting data type can accommodate.

Note: Use saturate only if the algorithm requires it, because this method increases ROM consumption and execution time.

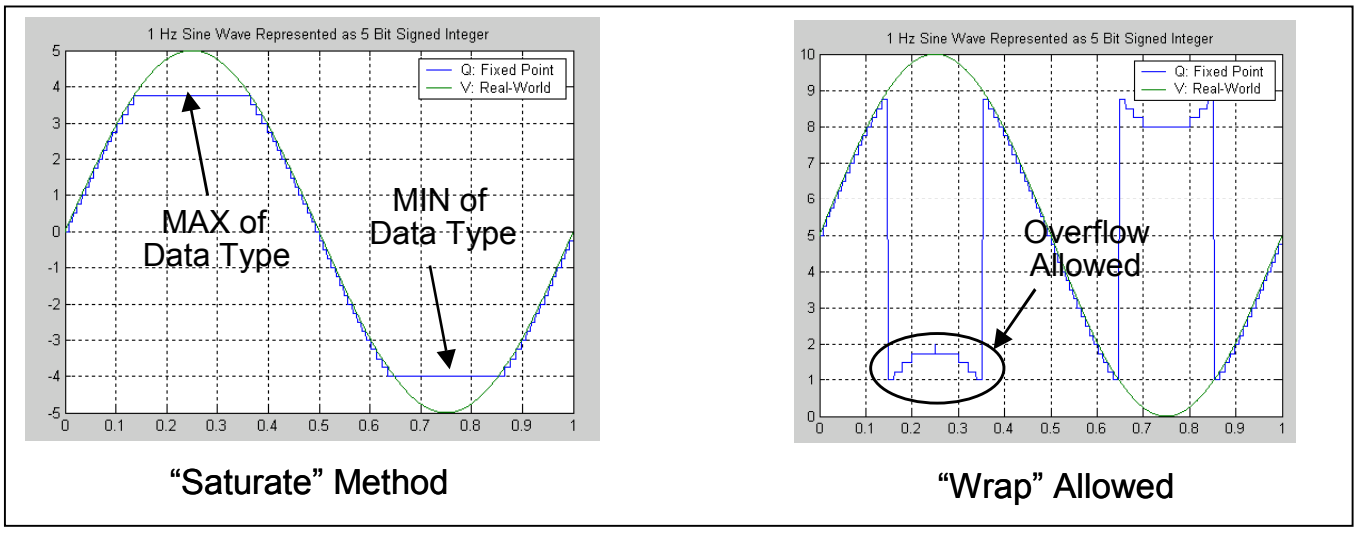

Figure 7: Comparison of handling methods.

For each block output in the model, you must specify the word size and binary-point location. Also, depending on the kind of mathematical calculation performed on two or more fixed-point numbers, the result may require greater word size. First, you will need to simulate the model, looking for outputs that best approximate values obtained from a floating-point simulation of the same model. An inappropriate tradeoff between range and resolution can cause an inaccurate result or unnecessary code size. Or worse, an incorrect tradeoff can cause an erroneous result, due to underflow or overflow.

General Solution

The table below describes the steps involved in a general solution to fixed-point modeling. This is a typical model-based design and development cycle often employed in various industries, such as manufacturing Implementing the recommended tips where indicated in this general solution will considerably improve your process.

Steps in the General Solution	**Recommended Tip** (See explanations in *Tips for Optimizing the General Solution*)
I. Develop a floating-point model of the control algorithm using blocks from the Simulink block libraries or Fixed-Point Blockset.	Tips 4, 6, and 16
II. Validate that the algorithms are correct in the floating-point model by performing simulation or rapid prototyping.	
III. Convert the floating-point model to fixed-point.	
A. If the model contains continuous blocks, replace each with the equivalent discrete block.	Tip 4
B. Establish the fixed-point representation.	
1. Convert the blocks to fixed-point manually, or automatically using `fixpt_convert`. In R13, many blocks support fixed-point intrinsically.	Tip 7
2. Specify fixed-point data types and scaling for Simulink blocks.	Tips 2, 3, 9, and 13
3. If model contains Stateflow charts with fixed-point data, use Stateflow Explorer to specify fixed-point data types and scaling (`sfexplr`).	
4. To bring data into the simulation or to export it from the simulation, add gateway blocks to inports and outports.	
C. Choose integer sizes.	
D. If you are unsure what the ranges and resolutions of the model's outputs should be, use the Autoscale Blocks button on the Fixed-Point Settings dialog box.. If you want to override all the Simulink and Stateflow data types as a double data type during simulation, use the Data Type Override field in the Fixed-Point Settings dialog box.	Tip 10 Tip 8
E. Run simulation and inspect results by comparing them to floating-point results.	Tip 14
F. Fine-tune the scaling and data type as needed for each block, and select the saturation and rounding method. Repeat steps D and E until the fixed-point design is acceptable.	Tip 11
IV. Validate the fixed-point design by performing simulation or rapid prototyping to ensure that the model's controller performance is within acceptable limits when compared with the floating-point simulation in step II above. If results are not within acceptable limits, return to step III.	
V. Generate the code.	
A. Review integer sizes based on the target microcontroller.	Tip 10
B. Set code generation optimization switches manually or automatically with `ecmodelprep`.	
C. Decide if generated code functions, data types, and packaging are acceptable.	Tips 12 and 15
D. Decide on using fixed-point math functions versus macros.	Tip 1
E. If functions are selected, minimize the number of fixed-point types and scaling combinations to reduce the number of functions generated.	Tips 1 and 2
F. If you do change type and scaling combination settings, then you have changed the fixed-point design. Return to step III D or IV, as necessary.	Tip 3
G. Inspect the generated code to determine additional optimization opportunities. For example, optimize blocks with cumbersome and unnecessary math operations. For this step, use the HTML Code Generation Report with links between the code and the model. For information on the report, see Real-Time Workshop Embedded Coder documentation.	Tips 5, 6, 11, 14, 15, 17, and 18
H. Compile and link code to check RAM and ROM size. See the Real-Time Workshop Embedded Coder documentation for a tutorial on how to quickly compile the generated code with MATLAB dependencies.	
VI. Deploy and validate the code on the target (possibly preceded by running it on an evaluation board).	
A. Validate the software execution results against the fixed-point design in step IV.	
B. Check execution timing against timing requirements.	

Tips for Optimizing the General Solution

The optimum solution considers both the design and implementation needs of the development process, and does so throughout the process. A design perceived as "stellar" during the modeling stage is not satisfactory if, for example, the implemented code consumes too much ROM or executes too slowly. The collection of tips below condenses years of experience in performing fixed-point modeling and implementing its generated code in embedded microcontrollers. These tips take into account both stages of the process. Knowing these tips, and incorporating them at the strategic locations identified in the preceding table, will optimize your overall solution.

Note: The tips apply to MATLAB Release 13 and later. In the code examples used herein, the comments feature of Real-Time Workshop Embedded Coder was turned off for space considerations. All code is generated for a 16-bit target controller.

All .mdl and .m files referenced in this document are located in fixpt_tips.zip. Click **here** *to download this folder.*

Tip 1. Choosing a macro or function for math operations to reduce ROM size

Usually, we recommend using macros for math operations. This is the default. However, if the model has a large number of fixed-point operations that require special attention, consider using math functions instead of macros. The purpose of this is so that the function can be called multiple times, thus reducing ROM consumption. However, this may increase execution time. Examples of fixed-point operations needing special attention are saturation, rounding, and operations with operands that are larger than target integer size.

You can use one of two methods: type a function option into a field on the Simulation Parameters dialog box, or choose "function" from a menu.

Type a function option:

1. Navigate to the Simulation Parameters dialog. (With the model open: Simulation > Simulation Parameters > Real-Time Workshop >. In the System target file field, browse to ert.tlc.)
2. As shown below, add the `-aFixPtUtilType="function"` option in the System target file field.

Note: FunctionVsMacro.mdl is an example model for this tip.

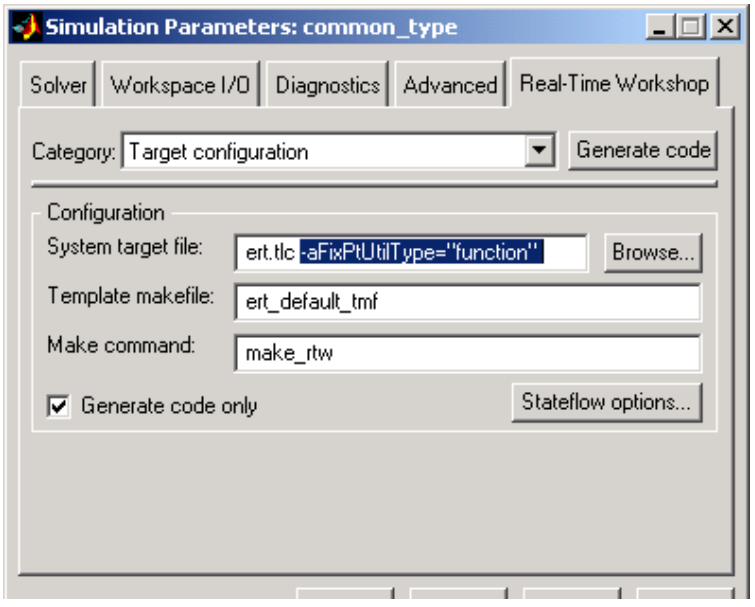

Figure 8: "System target file" field.

Choose "function" from a menu:

1. Locate and open fixpt_code_gen_target_installation.txt in the fixpt.zip folder.

2. By following the instructions in fixpt_code_gen_target_installation.txt, install the files located in fixpt_code_gen_target.zip.

3. Restart MATLAB.

4. Navigate to the Simulation Parameters dialog. (With the model open: Simulation > Simulation Parameters > Real-Time Workshop.

5. Using the Browse button beside the System target file field, select *fixpt.tlc*.

6. Select *Fixed Point options* in the Category field.

7. Select *function* in the Select macro or function field.

Tip 2. Using common fixed-point data type and scaling for ROM reduction

Each algorithm requires certain data types and scalings for blocks in the fixed-point model. This requires the developer to use a variety of fixed-point number formats, making tradeoffs between word size and binary-point location in each. Keeping this variety of fixed-point numbers to a minimum, so that the different blocks use common fixed-point formats where possible, will reduce ROM. This is true whether you choose a macro or a function for fixed-point math operations. (See Tip 1.) However, choosing functions instead of macros results in significantly less ROM. The ROM is reduced, in either case, because Real-Time Workshop Embedded Coder generates a unique function in the code for each mathematical operation having operands of unique format. The unique format consists of a combination of type sign (signed or unsigned) and scaling. In general, having a common type sign provides more efficiency than having common scaling. When different blocks use common fixed-point data types and scaling, the generated mathematical functions are reused in the code.

For example, for the model below, the data type is a signed fixed-point 16-bit, and scaling is $1/2^7$.

SpeedType = sfix(16);
SpeedScaling = [2^-7 0];

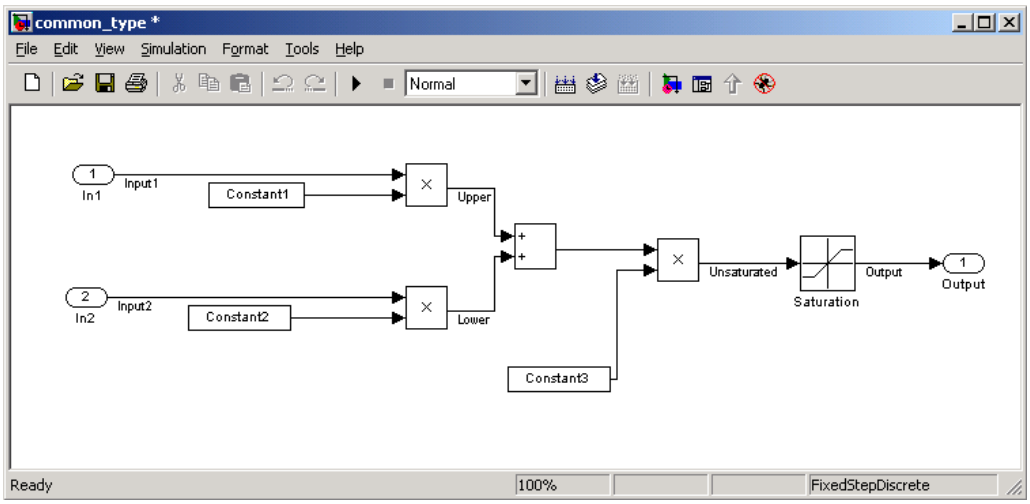

Figure 9: Model common_type.mdl.

Now the developer can set the data type and scaling of related blocks to common settings. For example, on the Block Parameters dialog shown below, the output data type and scaling of the multiplication and addition blocks in this model are set to SpeedType and SpeedScaling, respectively.

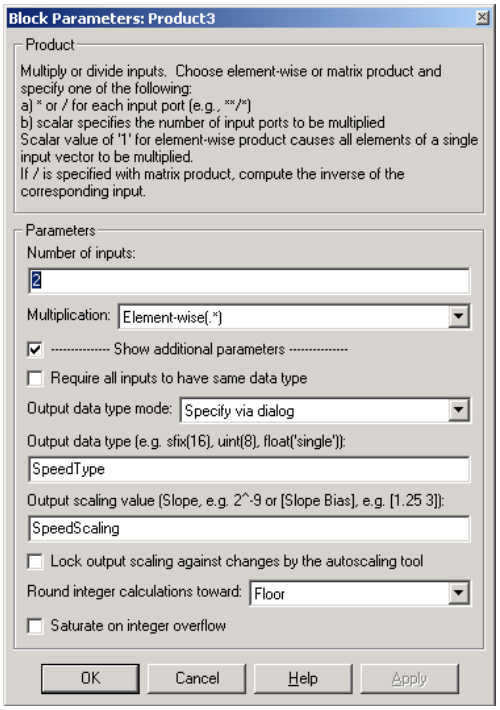

Figure 10: Example settings for output data type and scaling.

The code that results is shown in Figure 11. The code reuses the multiplication routine `MUL_S16_S16_S16_SR7()` because of common type usage.

```
int16_T Upper;
int16_T Lower;
int16_T Unsaturated;
int16_T Output;

const int16_T Constant1 = 320;
const int16_T Constant2 = 192;
const int16_T Constant3 = 3264;

void MUL_S16_S16_S16_SR7( int16_T *C, int16_T A, int16_T B)
{
  *C = ((int16_T)ASRl(7,(((long)(A)) * ((long)(B)))));
}

void common_type_step(void)
{

  MUL_S16_S16_S16_SR7(&(Upper),Input1,Constant1);

  MUL_S16_S16_S16_SR7(&(Lower),Input2,Constant2);

  MUL_S16_S16_S16_SR7(&(Unsaturated),(Upper + Lower),Constant3);

  {
    Output = rt_SATURATE(Unsaturated,(-640),2048);
  }
}
```

Figure 11: The same function is used in code due to common data type and scaling.

To achieve ROM reduction:

1. Open the model.
2. Determine the variety of data types and scaling that the algorithm needs.
3. According to the algorithm's needs, set the common data type and scaling for the appropriate blocks.
4. Generate code.

Tip 3. Restricting data type sizes to reduce ROM and execution time

Restricting fixed-point data type sizes so that they are equal to or less than the integer size of the target microcontroller reduces ROM and execution time. Thus, restricting data type sizes results in fewer mathematical instructions in the target microcontroller. Otherwise, more mathematical instructions will be required.

For example, performing a multiplication or division operation with 32-bit operands on a 16-bit microcontroller adds overhead in the code. Notice the code labeled "Added Code" shown in Figure 13 for the model below.

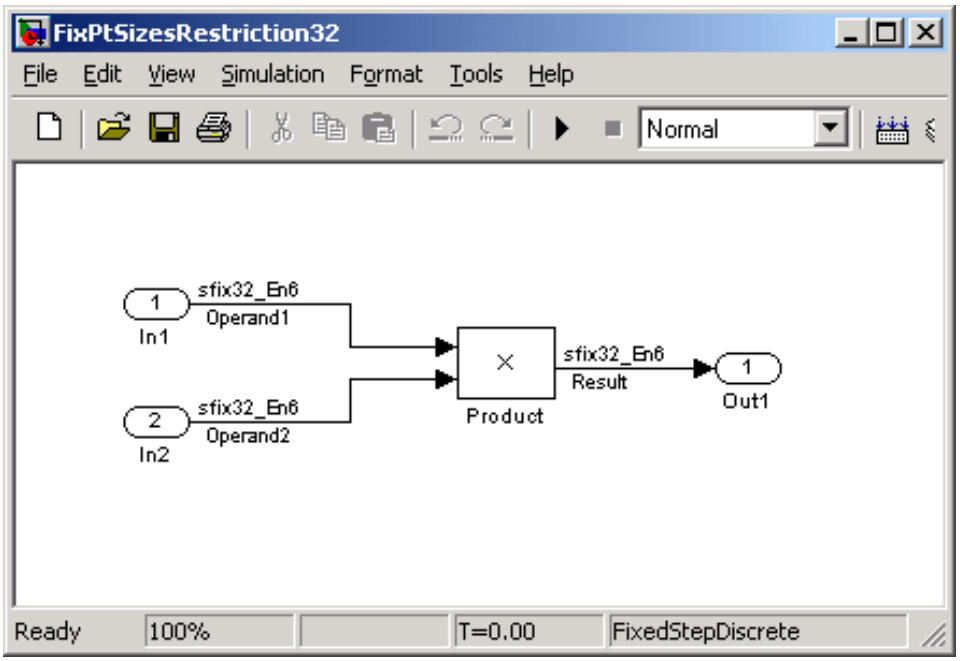

Figure 12: Model FixPtSizesRestriction32.mdl.

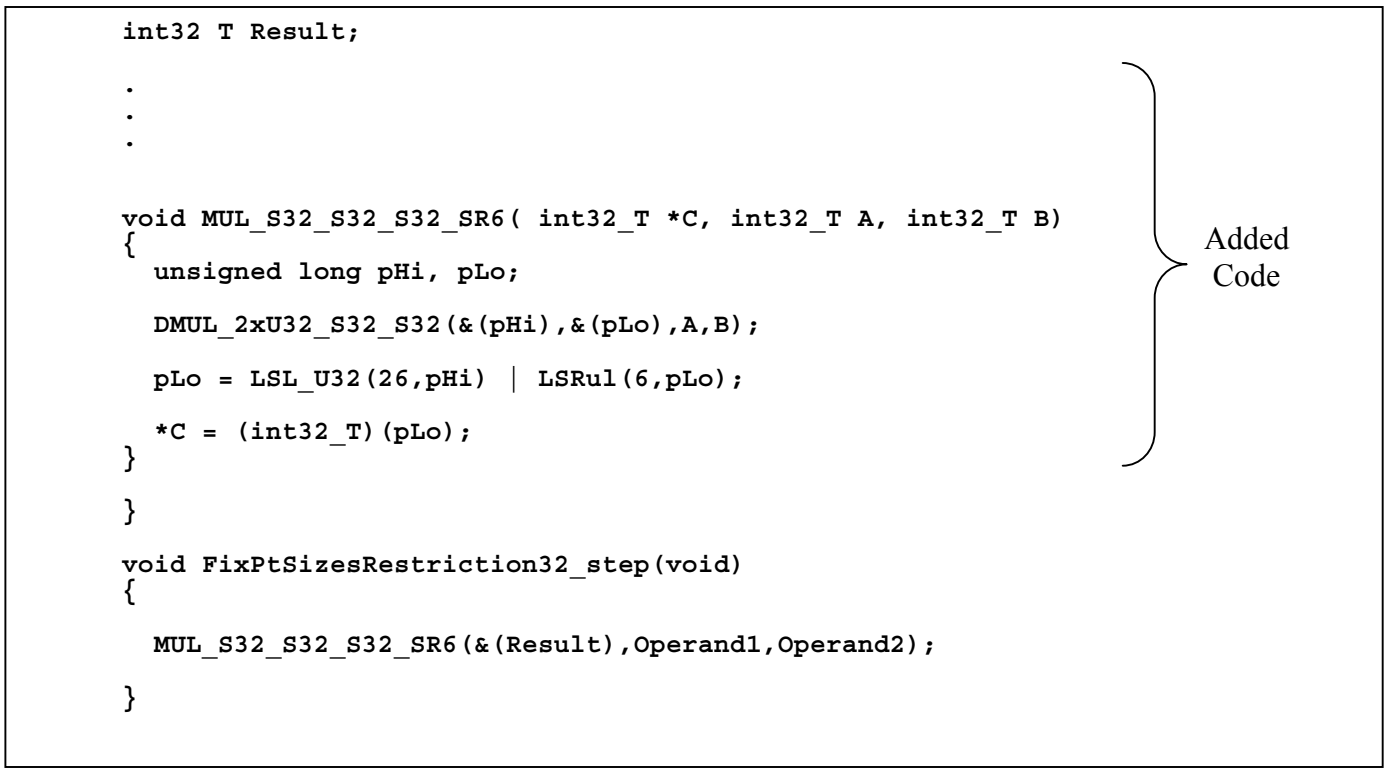

Figure 13: Code has been addeddue to the type size being larger than the target integer size.

The model below uses 16-bit data types for the 16-bit target microcontroller. The generated code is shown in Figure 15. Notice how clear and concise the code is compared with that shown in Figure 13.

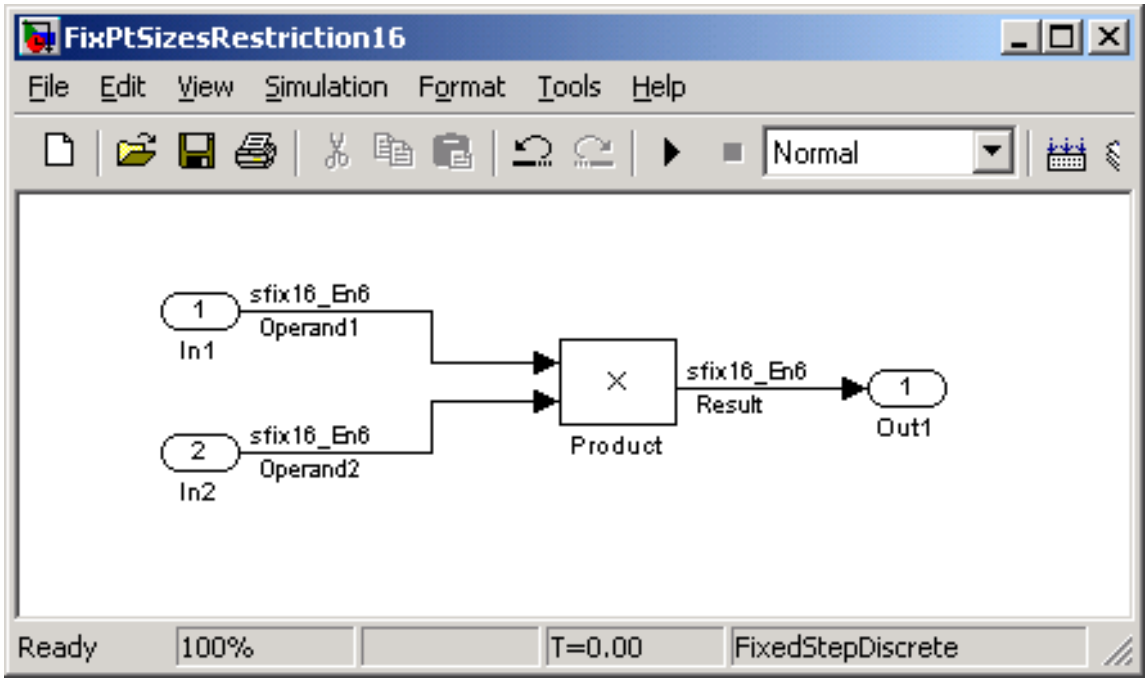

Figure 14: Model FixPtSizesRestriction16.mdl.

```
int16 T Result;

void MUL_S16_S16_S16_SR6( int16_T *C, int16_T A, int16_T B)
{
  *C = ((int16_T)ASRl(6,(((long)(A)) * ((long)(B)))));
}

void FixPtSizesRestriction16_step(void)
{

  MUL_S16_S16_S16_SR6(&(Result),Operand1,Operand2);

}
```

Figure 15: This code is more concise due to the type size being less than thetarget integer size.

To reduce ROM and execution time:

1. Determine the integer size of the target microcontroller.
2. Open the model.
3. Restrict fixed-point data type sizes to be equal to or less than the integer size of the target microcontroller.
4. Generate code.

Tip 4. Using blocks that support fixed-point production code generation

During the initial floating-point design that will support fixed-point implementation, select floating-point blocks that support fixed-point. This eliminates the need to replace floating-point blocks using a conversion script or manual methods. With Release 13, most Simulink blocks now have intrinsic fixed-point support. The Fixed-Point Blockset now comes as part of the standard Release 13 Simulink blockset. It can be used at no extra charge (without a license) for floating-point, double-precision operations, such as simulation or code generation.

A number of block types, such as the Gain Block, exist within both the Fixed-Point Blockset and the core Simulink blockset. The only differences between these are the default values set within them. The defaults in the blocks from the Fixed-Point Blockset have fixed-point characteristics already enabled.

Fixed-Point Blockset blocks without a corresponding block in the core Simulink blockset have the letter “F” on the bottom right corner of their block representation. With Release 13, all of the blocks with the "F" designation are available to floating-point developers. This includes the Rate Limiter block.

MATLAB includes a reference table that shows the data types that support simulation and code generation for each block. This table also indicates whether or not the blocks are optimal for production code generation. ll

The MATLAB command that accesses this table is **`showblockdatatypetable`**.

A portion of this table is shown below.

Sublibrary	Block	Double	Single	Boolean	Base Integer	Fixed-Point	Suitable for Production Code?
Sources	Band-Limited White Noise	X					X (C1)
	Chirp	X					
	Clock	X					
	Constant	X	X	X	X	X	X
	Digital Clock	X					
	From File	X					
	From Workspace	X					
	Ground	X	X	X	X	X	X
	Inport (In1)	X	X	X	X	X	X
	Pulse Generator	X	X	X	X		X (C1, C2)
	Ramp	X					
	Random Number	X					X
	Repeating Sequence	X					
	Signal Builder	X					
	Signal Generator	X					
	Sine Wave	X					X (C2, C3)
	Step	X					
	Uniform Random Number	X					X

Figure 16: Portion of block data type table.

- During the initial floating-point design of the model, select only those floating-point blocks that support fixed-point.

Tip 5. Achieving maximum benefit of expression folding by preventing overflow check and rounding

Expression folding combines outputs of multiple sequential Simulink blocks into a single expression during code generation in order to reduce ROM and execution time. To obtain the full benefit of expression folding for each block in the model, clear the S*aturate on integer overflow* check box and select *Floor in the Round integer calculations toward* field on the Block Parameters dialog, as shown for the model below. Figure 19 shows the generated code that results.

Caution: Turn off saturation only if you are not concerned about overflow, and set the rounding option to Floor only if you do not have a need for maximum numerical precision.

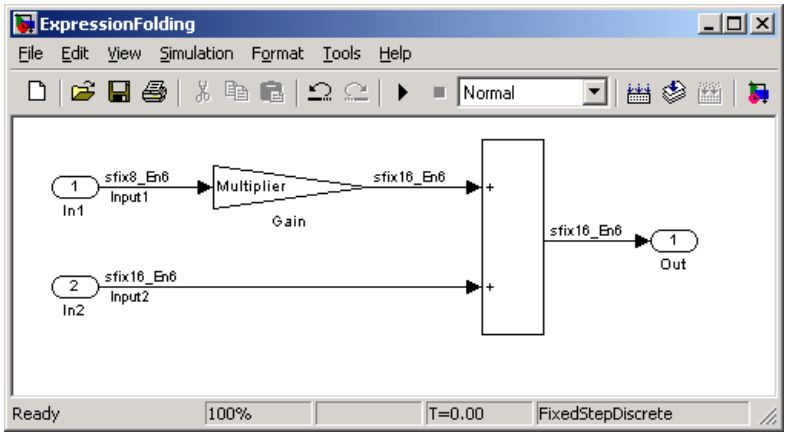

Figure 17: Model ExpressionFolding.mdl.

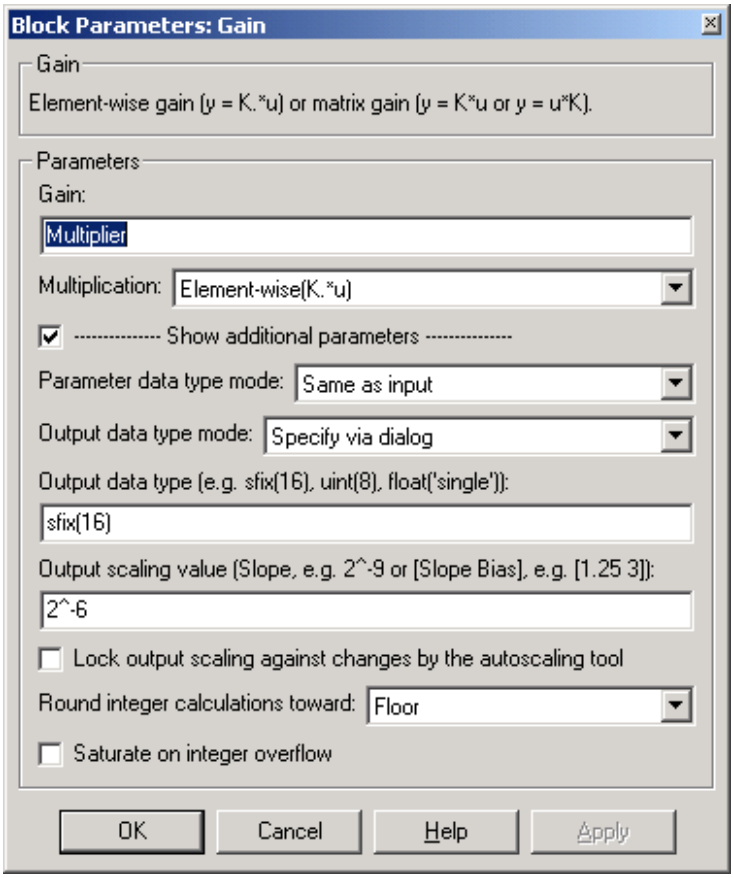

Figure 18: Settings for expression folding.

```
const int8 T Multiplier = 77;

ExternalOutputs rtY;

void ExpressionFolding_step(void)
{

  rtY.Out = (ASR(6,(Multiplier*Input1)) + Input2);

}
```

Figure 19: This code has neither overflow check or rounding.

Note that code for the multiplication and addition blocks are combined into the single statement `rtY.Out = (ASR(6,(Multiplier*Input1)) + Input2);`.

By comparison, Figure 20 shows the less efficient code that results from selecting *Round integer calculations toward* to be Zero. Figure 21 shows the less efficient code that results from selecting *Saturate on integer overflow*, and changing the Gain block output to type sfix(8).

```
const int8 T Multiplier = 77;

ExternalOutputs rtY;

void ExpressionFolding_step(void)
{

  int16_T rtb_Gain;

  {
    MUL_S16_S8_S8_SR6_ZERO(rtb_Gain,Input1,Multiplier);
  }

  rtY.Out = (rtb_Gain + Input2);

}
```

Figure 20: Code with rounding.

```
const int8 T Multiplier = 77;

ExternalOutputs rtY;

void ExpressionFolding_step(void)
{

  int8_T rtb_Gain;

  {
    MUL_S8_S8_S8_SR6_SAT(rtb_Gain,Input1,Multiplier);
  }

  rtY.Out = (((int16_T)rtb_Gain) + Input2);

}
```

Figure 21: Code with overflow check.

For each block in the model:
1. Double-click the block. The Block Parameters dialog box, appears.
2. Select Floor in the *Round integer calculations toward* field. (Floor is the default.)
3. Clear *Saturate on integer overflow.* (If the block is from the Fixed-Point Blockset, clear is the default.)
4. Generate code.

Note: For more information about obtaining the full benefit of expression folding, see the next tip.

Tip 6. Trading off naming versus ROM by selecting a storage class

Note: This tip is not limited to fixed-point modeling.

To obtain more benefits of expression folding, in addition to those mentioned in the previous tip, set the *RTW storage class* field on the Signal Properties dialog to Auto for each signal. This is the default. If you choose a setting in *RTW storage class* other than Auto, you will be forced to name the signal and a separate statement will be generated. However, some signals need the storage class setting to be other than Auto. Change the storage class to a selection other than Auto only for those signals that need visibility (i.e., need to be global variables). For example, signals for calibration need to be global variables. Otherwise, the code may consume additional memory unnecessarily.

For the example model below, Figure 24 shows code when Auto is selected. However, when the storage class of the Gain block named MultiOutput is set to ExportedGlobal, this generates the code shown in Figure 25. In this figure, the generated code has multiplication and addition as separate statements.

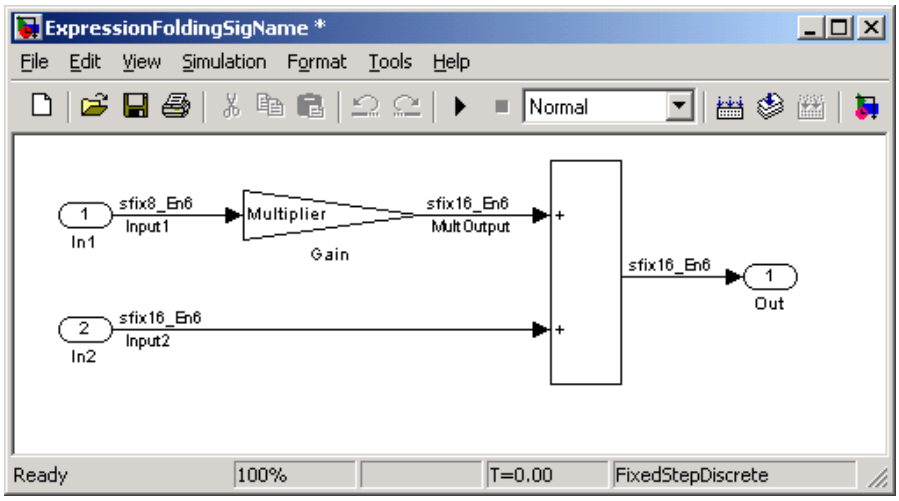

Figure 22: Model ExpressionFolding.mdl.

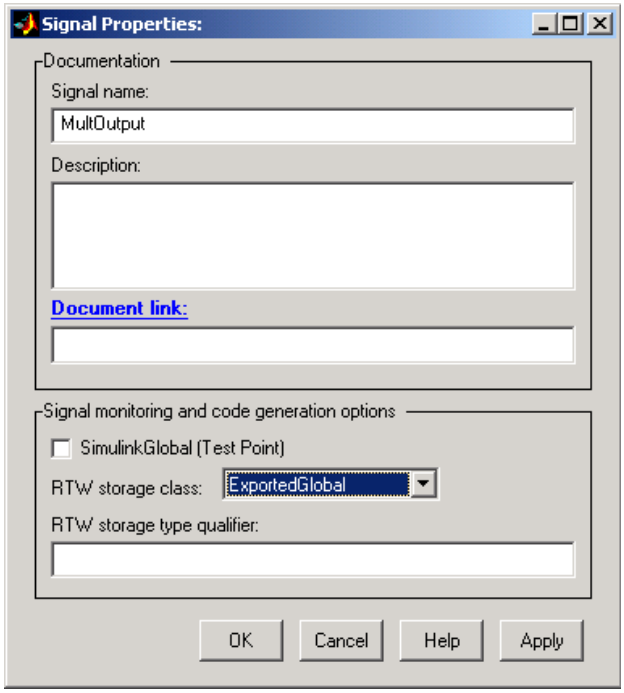

Figure 23: Setting for more benefit of expression folding.

```
const int8 T Multiplier = 77;

ExternalOutputs rtY;

void ExpressionFoldingSigName_step(void)
{

  rtY.Out = (ASR(6,(Multiplier*Input1)) +
Input2);

}
```

Figure 24: Code with auto selection.

```
int16 T MultOutput;

const int8_T Multiplier = 77;

ExternalOutputs rtY;

void ExpressionFoldingSigName_step(void)
{

  {
    MUL_S16_S8_S8_SR6(MultOutput,Input1,Multiplier);
  }

  rtY.Out = (MultOutput + Input2);

}
```

Figure 25: Code with auto not selected.

For all of the signals in the model that do not need visibility:

1. Right-click the signal. The Signal Properties dialog appears.
2. Ensure that Auto is selected in the *RTW storage class* field.

For all of the signals in the model that do need visibility,

1. Right-click the signal. The Signal Properties dialog appears.
2. Choose the appropriate selection in the *RTW storage class* field other than Auto.
3. Generate code.

Tip 7. Converting a floating-point model to a fixed-point model

It is easier to develop a model using idealized floating-point values than to develop a model using implementation-specific, fixed-point values. An idealized floating-point based model does not require you to be concerned with implementation-specific data type and scaling details. However, after simulating and validating the algorithm, you do need to run the model's code on a target microcontroller. In most cases, the microcontroller's constraints require that the idealized model be converted to an implementation-specific, fixed-point model so that fixed-point code is generated for use on the target microcontroller.

The conversion process introduces blocks that are necessary for simulation with idealized, floating-point data. However, these added blocks are unnecessary, and add extra data and code not intended for the target controller. Because of this, after you validate the implementation-specific fixed-point algorithm, it is best to remove these added blocks and set the appropriate data types and scalings on the appropriate blocks in the model.

To convert a floating-point model to a fixed-point model:

1. Use the fixtp_convert( ) utility to convert the floating-point model to a fixed-point model. The utility places gateways in the model at inputs and outputs where needed. Also, if the model has blocks that cannot be converted to blocks that support fixed-point, the utility changes these blocks to subsystems, colors them red, and places gateways inside the subsystems as necessary. An input gateway is between the input port and the model. It changes a double data type input to a data type that a fixed-point model can recognize. An output gateway is between the model and an output port. It changes a data type that a fixed-point block can recognize to a double data type.

2. Prior to validating the fixed-point model, set the BaseType appropriate to the model and target controller, and set LogicType as desired.

3. Validate the fixed-point model using the same input vector that was used to validate the floating-point model. (The input vector contains the same double data type inputs that were used to validate the floating-point model.)

4. Once the fixed-point model is validated, make the model pure fixed-point by getting rid of all gateways:
    a. For each model input port, set the appropriate fixed-point data type and scaling on the Block Parameters dialog. This eliminates the need for gateways at the model's input ports.
    b. For each model output port, delete the gateway and reconnect the output. This eliminates the gateways at the model's output ports.
    c. For each red subsystem, manually replace the block that does not support fixed-point with a block that does support fixed-point. (Drag fixed-point blocks from the fixed-point block palate in the Simulink library browser.)
    d. For each Stateflow chart, select *Use strong data typing with Simulink I/O* on the Stateflow Properties dialog box.
    e. On inputs and outputs of a Stateflow block, delete the gateways.
    f. Set the data types and scalings of the data in the Stateflow data dictionary to be consistent with the inputs and outputs of the model and equations in the Stateflow diagram.
    g. In each red subsystem repeat a.
    h. For each red subsystem repeat b.

5. Now generate code for the fixed-point model. The code has the desired implementation-specific data types and scaling.

Tip 8. Reducing workload using the autoscale blockset tool

You can use the FixPtGUI in the Fixed-Point Blockset of the Simulink Library to perform autoscaling on all data in the Simulink and Stateflow model. (Autoscaling is the automatic selection of scaling values of signals and parameters based on the changing values of each signal during a simulation run.) Since every signal ultimately relates back to an input port, we recommend that the input vectors have enough range to exercise all the signals in the model appropriately. This avoids having to set manually data types and scalings for each signal, and having to repeat this process until you obtain the desired results.

Clicking the *Autoscale Blocks* button (tool) on the Fixed-Point Settings dialog box automatically changes the scaling for each block that does not have its scaling locked. The tool uses the maximum and minimum data obtained from the last simulation run to log data to the workspace. If the maximum and minimum data cover the intended range of your design, the autoscaling tool changes the scaling such that the simulation range is covered and the precision is maximized.

Note: The maximum and minimum simulation data must cover the full intended operating range of your design in order for the autoscaling tool to yield meaningful results. The autoscaling tool changes scaling only for those blocks for which you select *Specify via dialog* in the *Output data type mode* field on the Block Parameters dialog.

In order to obtain meaningful results from the autoscaling tool, the maximum and minimum simulation data used by the tool must exercise the full range of values over which your design is meant to run. Therefore, the simulation you run prior to using the autoscaling tool should simulate your design over its full operating range.

It is especially important that you select inputs with appropriate speed and amplitude profiles for dynamic systems. The response of a linear dynamic system is frequency dependent. For example, a bandpass filter will show almost no response to very slow and very fast sinusoid inputs, whereas the signal of a sinusoid input with a frequency in the passband will be passed or even significantly amplified. The response of nonlinear dynamic systems can have complicated dependence on both the signal speed and amplitude. For such reasons, practical knowledge of the intended use of your design is the best basis for selecting inputs to exercise your system. Even with well selected inputs, however, it is often good engineering practice to add a safety margin. If you use the RangeFactor variable as described below, the autoscaling tool can set the binary points so that an even larger simulation range is covered. A larger range reduces the chance of an overflow occurring. However, increased range results in reduced precision, so the safety margin you choose must be limited.

To reduce your workload using the autoscale blockset tool:

1. Determine the appropriate minimum and maximum values for each input signal in the model.

2. Open the model.

3. Using the Signal Builder in Simulink Sources of the Simulink Library, configure the input vectors for each signal, taking into account these minimum and maximum values.

4. Use the FixPtGUI to perform autoscaling on all signals in the Simulink model.

5. Generate code.

Tip 9. Avoiding fixed-point numbers with bias

In most cases, mathematical operations involving fixed-point numbers with bias will increase ROM consumption and execution time. In certain cases, when selecting appropriate biases for the mathematical operations, you may be able to avoid these increases. For example, if you are performing an addition operation, the bias of the inputs should add up to the bias of the output.

Therefore, we recommend not using fixed-point numbers with bias, unless it is essential. For example, if you are interfacing to hardware devices, the biases are required. The hardware device may have a built-in bias. In such cases, you must have fixed-point numbers with bias in order to interface to the device correctly.

- Where possible, avoid using fixed-point numbers with bias.

Tip 10. Setting integer sizes for the target microcontroller

It is necessary to specify word sizes for integer data types (such as long, int, and short) for a specific target microcontroller. Real-Time Workshop Embedded Coder uses this information to map types in the model (such as uint8, uint16, int8, and int16) to types that the microcontroller accommodates. If word sizes of integer data types are set inappropriately, the wrong code will be generated, resulting in compile-time error.

To set the integer size for the target microcontroller,

1. Open the file example_rtw_info_hook.m. This is in the path $MATLB_ROOT\toolbox\rtw\rtwdemos\.

2. Change the function title from **`function varargout = example_rtw_info_hook(varargrin)`**
   to **`function varargout = ert_rtw_info_hook(varargin)`**

3. In the **`case 'wordlengths'`** of the **`switch Action`** statement, set the values that are appropriate to your target microcontroller. For example, the settings for the Siemens C167 are as indicated below:

```
% specify the target word lengths

value.CharNumBits  = 8;
value.ShortNumBits = 16;
value.IntNumBits   = 16;
value.LongNumBits  = 32;
varargout{1} = value;
```

4. Ensure that the sizes for integer data types shown in the preceding step match those entered on the Simulation Parameters dialog box (Advanced tab) when Microprocessor is selected in that dialog's *Production hardware characteristics* field.

5. Change the filename from example_rtw_info_hook.m to ert_rtw_info_hook.m, and place it in the MATLAB path.

Tip 11. Changing preferences for fixed-point blocks in a model globally

You can change the preferences of each block of a model one-by-one by making the desired settings on the Block Preferences dialog box. However, this can be tedious. There is a script called fixpt_blk_pref.m in the fixpt_tips.zip folder that you downloaded under **Tips for Optimizing the General Solution**. This script contains a function that has two arguments: `fixpt_blk_pref ('Model_Name', Block_Preferences)`. Model_Name is the name of the open model. Block_Preferences is a structure containing block preferences. As shown below, after you build the structure, running this function call on the MATLAB command line while the model is open changes block preferences globally for all of the model's blocks or for selected blocks.

The following code builds a structure that will set the *SaturateOnIntegerOverflow* preference for all blocks. (This code is in ExampleSaturationOff.m.)

```
Block_Preferences = {};
clear pref
pref.category = 'global';
pref.param = {};
pref.param{end+1}.NameStr  = 'SaturateOnIntegerOverflow';
pref.param{end  }.ValueStr = 'off';
Block_Preferences{end+1} = pref;
```

Here is code that builds a structure that will set the *OutDataTypeMode* preference of all (and only) Sum blocks in the model to *Inherit via internal rule*. For example:

```
Block_Preferences = {};
clear pref
pref.category = 'block';
pref.BlockType = 'Sum';
pref.param = {};
pref.param{end+1}.NameStr  = 'OutDataTypeMode';
pref.param{end  }.ValueStr = 'Inherit via internal rule';
Block_Preferences{end+1} = pref;
```

Note that the file fixpt_blk_pref_default.m contains code that builds structures for all available block preferences. This file is located in the fixpt_tips.zip folder that you downloaded under **Tips for Optimizing the General Solution**.

To change your preferences for fixed-point blocks in a model globally:

1. Open the model.

2. Copy fixpt_blk_pref.m from fixpt_tips.zip and place it on the MATLAB path.

3. Build the desired block preferences structure, or copy it from fixpt_blk_pref_default.m, and place it in a new .m file. Call this new .m file an appropriate name and place it on the MATLAB path.

4. On the MATLAB command line, run the new .m file that was created in the preceding step. Then, on the MATLAB command line, run fixpt_blk_pref ('Model_Name', Block_Preference), replacing Model_Name with the name of the open model (between single quotes), and replacing Block_Preferences with the name of the new .m file created in the preceding step.

5. Click a block on the model and notice on the Block_Preferences dialog that the property was changed according to the block preferences structure that you built.

Tip 12. Eliminating I/O structures

Note: This tip is not limited to fixed-point modeling.

Oftentimes, you are required to integrate legacy code with the generated code. In this case, the legacy code may need to interface with the generated code using the names of input and output signals in the model that have no reference to a structure. In this case, it is necessary to eliminate I/O structures from the generated code.

Consider the Signal Properties dialog that appears when you right-click a signal line in a model, as shown below.

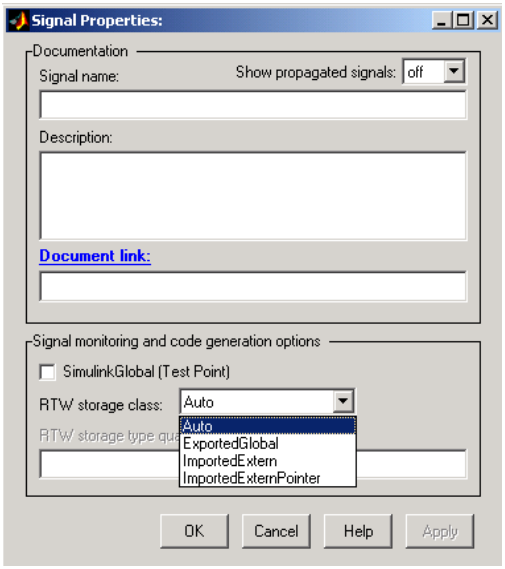

Figure 26: Signal properties dialog.

When the default Auto is selected in the RTW storage class field, Real-Time Workshop Embedded Coder will place in the generated code all inputs into a structure called ExternalInput, and all outputs into a structure called ExternalOutput .

For the model below, the generated code is shown in Figure 28.

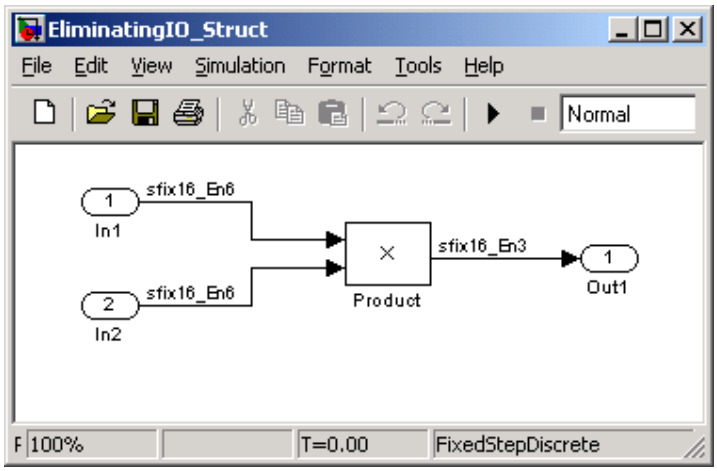

Figure 27: Model EliminatingIO_Struct.mdl

```
typedef struct  ExternalInputs tag {
  int16_T In1;
  int16_T In2;
} ExternalInputs;

typedef struct _ExternalOutputs_tag {
  int16_T Out1;
} ExternalOutputs;
.
.
.

ExternalInputs rtU;

ExternalOutputs rtY;

void EliminatingIO_Struct_step(void)
{

  rtY.Out1 =
((int16_T)ASR1(9,(((long)rtU.In1)*((long)rtU.In2))));

}
```

Figure 28: Code with ExternalInput and ExternalOutput structures.

If you choose one of the other selections in the *RTW storage class* field instead of the default Auto, the ExternalInput and ExternalOutput structures will not appear in the generated code. In addition, the signal names themselves will replace the Embedded Coder-defined "structure_name.signal_name" nomenclature. Furthermore, for certain compilers you may achieve the added benefit of ROM savings. This is due to the fact that the code accesses the signal directly rather than through a structure.

For example, for the model below, the generated code is shown in Figure 30. Compare the two figures and notice that the code in Figure 30 does not have references to structures.

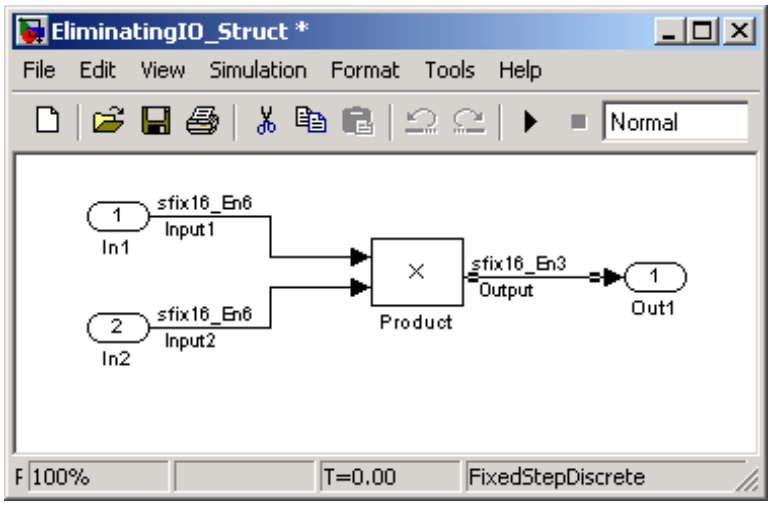

Figure 29: Model EliminationgIO_Structure.mdl.

```
extern int16 T Input1;
extern int16_T Input2;
.
.
.
int16_T Output;

void EliminatingIO_Struct_step(void)
{

  MUL_S16_S16_S16_SR9(Output,Input1,Input2);

}
```

Figure 30: Code without ExternalInput and ExternalOutput structures.

Note: See the Real-Time Workshop Embedded Coder documentation for a tutorial on storage classes for more details.

For each signal,

1. Right-click to display the Signal Properties dialog.
2. Choose the appropriate RTW storage class setting (not Auto).
3. Generate code.

Tip 13. Reducing time using back-propagation or internal rule for data type and scaling

Note: This tip is not limited to fixed-point modeling.

For each Simulink block output, you can take the time to type the data type and scaling on the Block Parameters dialog. This requires:

- Double clicking the block to open the Block Parameters dialog
- Selecting *Show additional parameters*, which expands the dialog
- Selecting *Specify via dialog* in the Output data type mode field, which expands the dialog more
- Typing the data type in the Output data type field and typing the scaling value in the Output scaling value field

A faster process is to select *Inherit via internal rule* or *Inherit via back propagation* in the Output data type mode field when appropriate, based on the algorithm. This saves you from having to type the data type and scaling value for the block.

When *Inherit via internal rule* is selected, Real-Time Workshop Embedded Coder selects the natural data type and scaling for the output. For example, as illustrated below, multiplying the two signed eight-bit signals (`sfix8_En3 * sfix8_En2`) results in `sfix16_En5`.

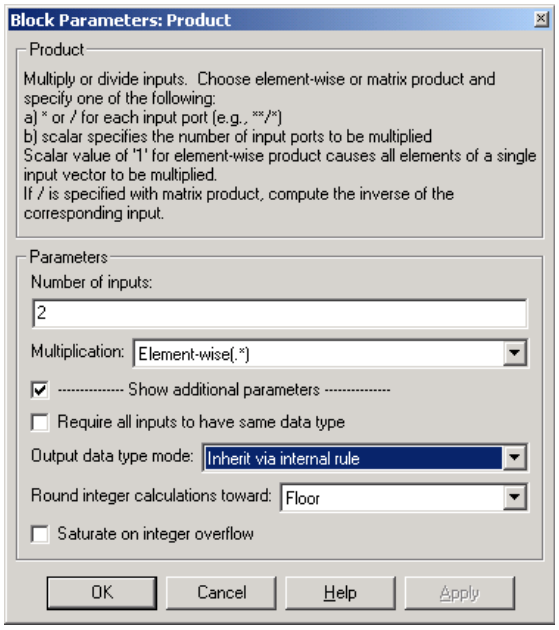

Figure 31: "Inherit via internal code" setting.

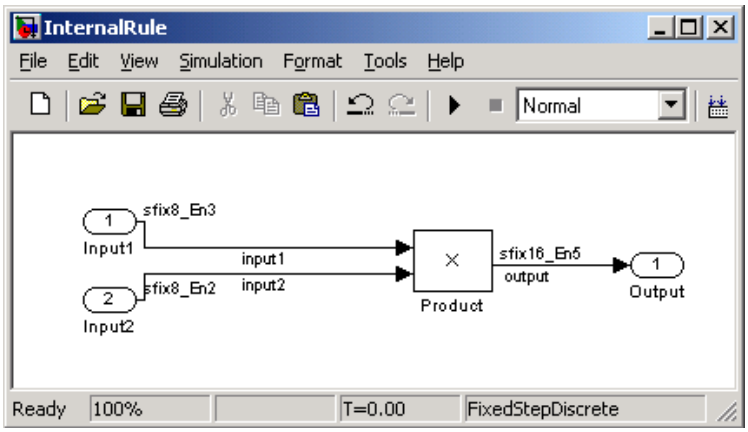

Figure 32: Model InternalRule.mdl.

When *Inherit via back propagation* is selected, Real-Time Workshop Embedded Coder determines the data type and scaling of the output signal by inheriting these from the block to which the output is connected. For example, as illustrated below, the Constant block output signal obtains its type (**`sfix8_En3`**) from the Relational Operator block to which the output signal is connected.

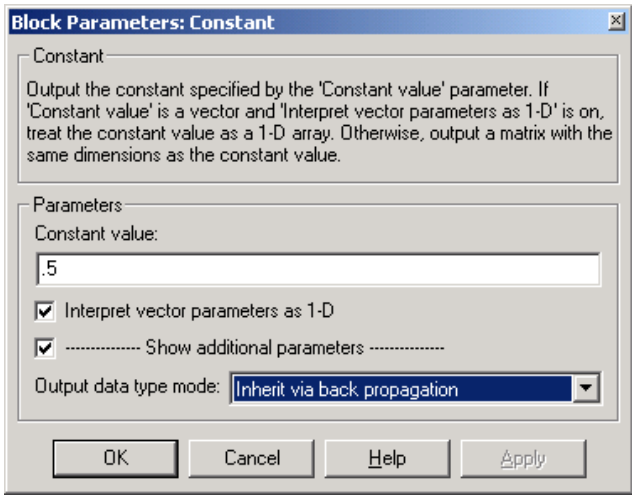

Figure 33: "Inherit via back propogation" setting

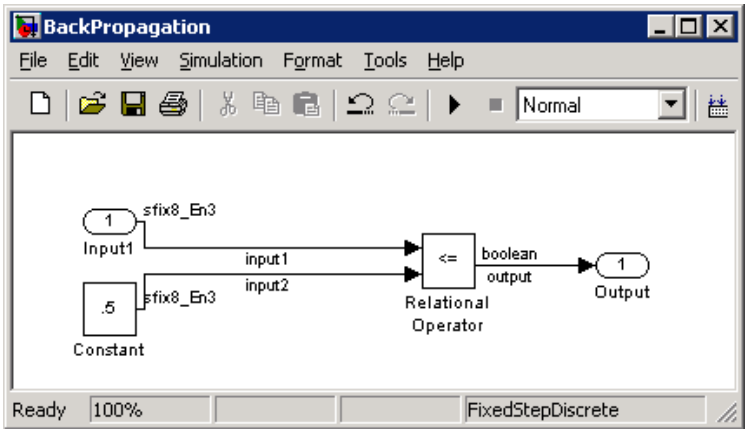

Figure 34: Model BackPropagation.mdl.

Regarding Stateflow

Data type and scaling do not autopropagate between Simulink and Stateflow. Therefore, to avoid having to enter data type and scaling in both Simulink and Stateflow for each input, you can select *Inherit via back propagation*. For example, in the next figure, the output from the Sum block is the input to the Limit chart. The user selects *Inherit via back propagation* in the Output data type mode field on the Block Parameters dialog for Sum, and clicks OK. Now the data type and scaling for the Sum block output (**`sfix16_En3`**) is driven solely by the Stateflow chart. Any change of data type or scaling in the Stateflow input will back propagate to Sum.

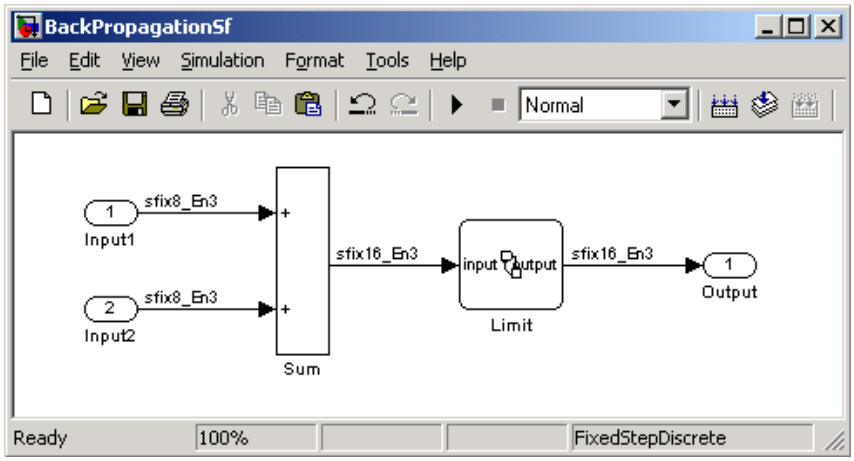

Figure 35: Model BackPropagationSF.mdl

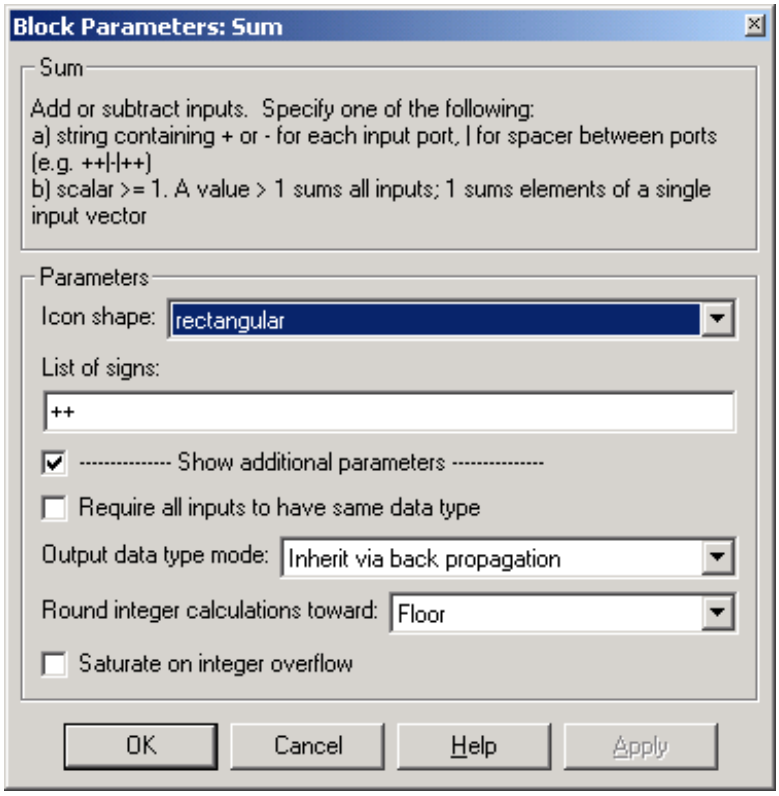

Figure 36: "Inherit for back propagation" for sum block.

To avoid having to enter data type and scaling manually for each block, :

1. Open the model.
2. Select either *Inherit via internal rule* or *Inherit via back propagation* in the Output data type mode field of the Block Parameters dialog as appropriate for each block.
3. Generate code.

Tip 14. Enforcing type compatibility of parameters between workspace and model

Note: This tip only applies when the parameter used in the model is of any of the following built-in data types: single, int8, uint8, int16, uint16, int32, and uint32.

By default, all the workspace tunable parameters are of type double. If the data type of a parameter in the base workspace is not the same as that parameter's data type in the model, casts are inserted into the generated code to make it so. Using a workspace parameter of data type double in the model can result in code with unexpected floating-point values. This can increase ROM consumption.

Also, this can cause an additional cast operation in the generated code. In the example below, the constant block named "NumOfSamples" references a parameter in the base workspace named NumberOfSamples. NumberOfSamples = 5.0 is set in the workspace. The data type is double by default. When code is generated for this model, the data type of NumOfSamples will be a double. The downcast from double to uint16 will be inserted in the generated code. This also can increase ROM consumption.

In order for NumberOfSamples to be of type uint16 in the generated code, the value assigned to NumberOfSamples needs to be type cast in the workspace as `NumberOfSamples = uint16(5.0);`. Figure 38 shows the code that results.

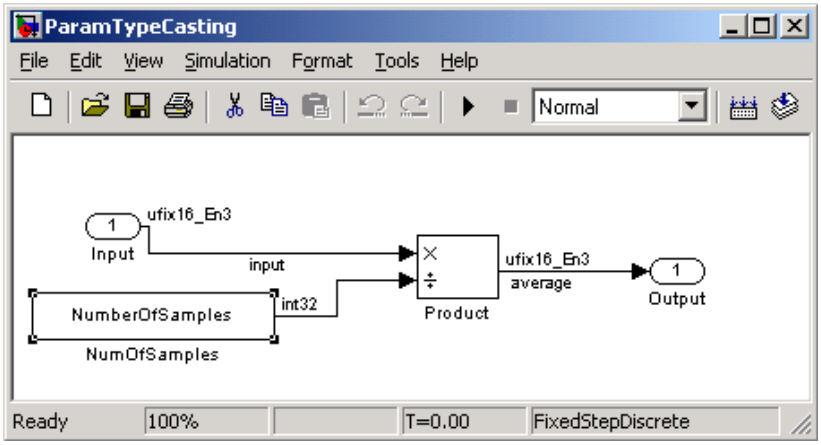

Figure 37: Model ParamTypeCasting.mdl.

```
uint16_T average;

uint16_T NumberOfSamples = 5U;

void ParamTypeCasting_step(void)
{

  DIV_U16_U16_S32_FLOOR(average,input,((int32_T)( NumberOfSamples )));

}
```

Figure 38: Code Resulting from Correct Type Casting.

Cautions

Some users have made the natural assumption that specifying "uint16(NumberOfSamples)" in the *Constant value* field of the Block Parameters dialog, as shown below, will make NumberOfSamples be type uint16 in the generated code. However, this is not so. The only way to achieve this is to type cast in the workspace as mentioned previously.

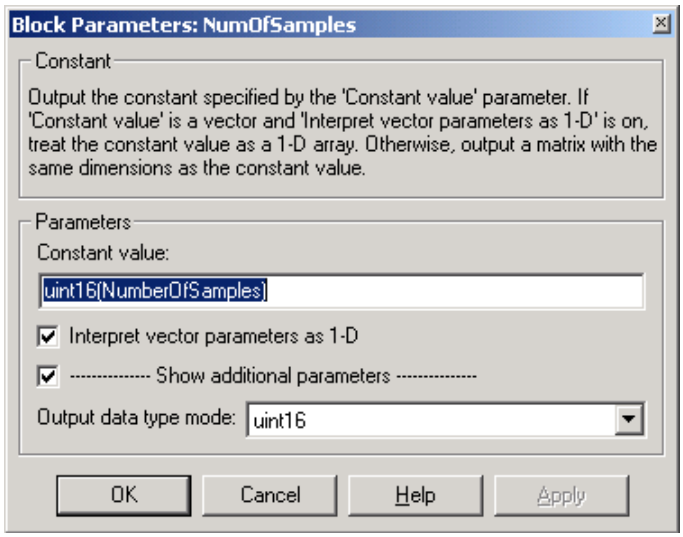

Figure 39: "Constant value" field.

Another misconception is that specifying uint16 in the *Output data type mode* field will define NumberOfSamples in the generated code to be data type uint16. However, this is not so. This setting only introduces a uint16 type cast in the generated code. Again, the only way to define NumberOfSamples in the generated code to uint16 (to obtain "`uint16_T NumberOfSamples = 5U;`" in Figure 38, for example) is to type cast in the workspace as mentioned previously. To define NumberOfSamples in the generated code as data type uint16:

1. Cast the parameter in the base workspace so that the correct data type is used for that parameter in the generated code.
2. In the *Output data type mode* field of the Block Parameters dialog box, specify the "correct data type" of the previous step.

Tip 15. Reducing ROM by making a reusable subsystem

Note: This tip is not limited to fixed-point modeling.

A single reusable subsystem can replace groups of common blocks. In the generated code, the reusable subsystem becomes a reusable function. Only the arguments change. This decreases ROM consumption. (You can also reuse an entire model, which will reduce ROM.)

For example, the Equation subsystem is replicated in the model below as Equation1.

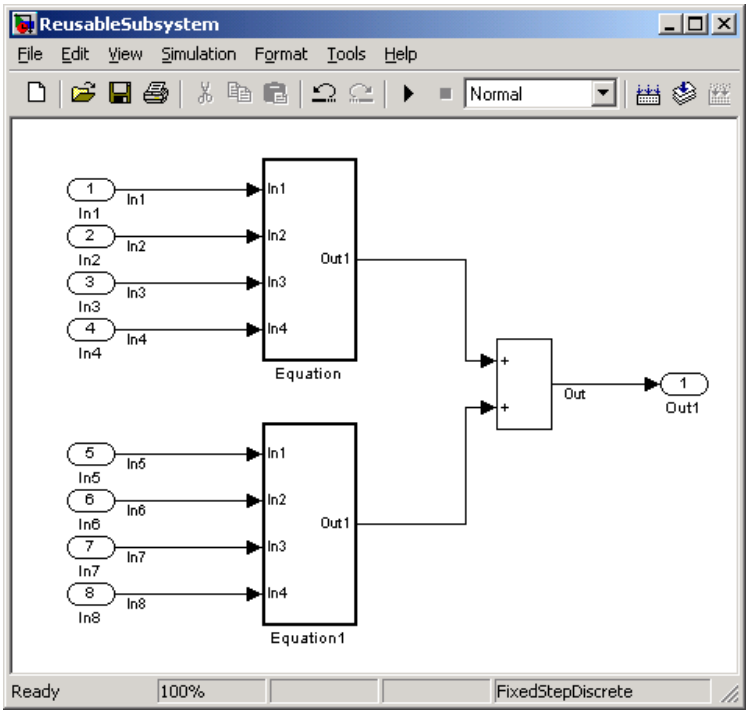

Figure 40: Model ReusableSubsystem.mdl.

Selecting the *Treat as atomic unit* checkbox, and selecting *Reusable function* in the *RTW system code* field on the Block Parameters dialog mades the subsystem Equation reusable.

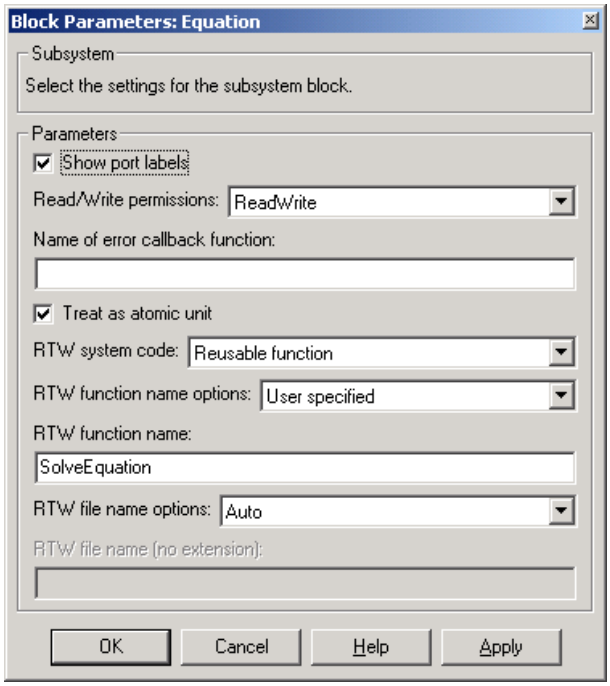

Figure 41: Settings for reusable subsystem.

The content of the Equation subsystem is shown below.

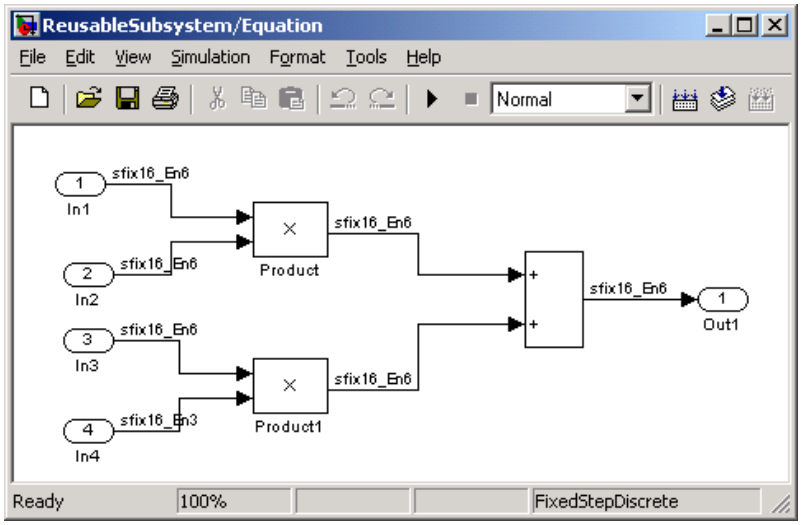

Figure 42: Equation subsystem in ReusableSubsystem.mdl.

Notice in the generated code shown below that the function SolveEquation () is invoked twice, with appropriate arguments for each instance.

```
int16_T Out;

BlockIO rtB;

void SolveEquation(int16_T In1, int16_T In2, int16_T In3, int16_T In4,
 rtB_SolveEquation *localB)
{

  localB->Sum = ((int16_T)ASR1(6,(((long)In1)*((long)In2))));
  localB->Sum += ((int16_T)ASR1(3,(((long)In3)*((long)In4))));
}

void ReusableSubsystem_step(void)
{

  SolveEquation(In1, In2, In3, In4, &rtB.Equation);

  SolveEquation(In5, In6, In7, In8, &rtB.Equation1);

  Out = rtB.Equation.Sum;
  Out += rtB.Equation1.Sum;

}
```

Figure 43: Reusable functions in the generated code.

To make a reusable subsystem,

1. Open the model.

2. Find any two or more groups of blocks whose input and output data type and scaling are identical.

3. Make one of the groups a subsystem as follows:
    a. Select all the blocks in the group, then select *Create subsystem* on the Edit menu.
    b. Right-click the subsystem and select *Subsystem parameters*. The Block Parameters: Subsystem dialog appears.
    c. Select *Treat as atomic unit*.and, in the *RTW system code* field, select *Reusable function.* Click OK.
    d. Select all the blocks in the second group (or groups) and delete them. This leaves an empty area with unconnected inputs and outputs.
    e. Copy the subsystem you made and paste it in the empty area or areas.
    f. Connect all the inputs and outputs to the subsystem or subsystems.

4. Generate code. The code contains a single reusable function for the subsystems.

To make a reusable model,

1. Open the model.

2. Select *Simulation parameters* on the Simulation menu.

3. Select *Real-Time Workshop*.

4. In the *Category* field, select *ERT code generation options (3).*

5. Select *Generate reusable code*.

6. Generate code. The code contains a single reusable function for the entire model.

Tip 16. Trading off between unevenly spaced versus evenly spaced lookup tables

The points along one or more independent input axes of a lookup table could be evenly spaced or unevenly spaced. There are advantages and disadvantages to each. An unevenly spaced input axis provides clustered points where necessary only at certain regions of the axis so as to produce greater lookup accuracy. As a result, the table can have fewer total points to achieve the same lookup accuracy compared with an evenly spaced table. However, unlike a table with one or more evenly spaced axes, a table with an unevenly spaced axis requires a search routine and memory for each input axis. This increases ROM and execution time.

Note: This tip applies only to tables with nontunable input axes. For a table with tunable input axes, this tip provides no benefit. This is due to the fact that tunable input axes always are treated as unevenly spaced.

- Considering the needs of the algorithm, decide whether or not to have an unevenly or evenly spaced table.

Figures 45 and 46 compare code generated for an unevenly spaced table versus that generated for an evenly spaced table, for the model below.

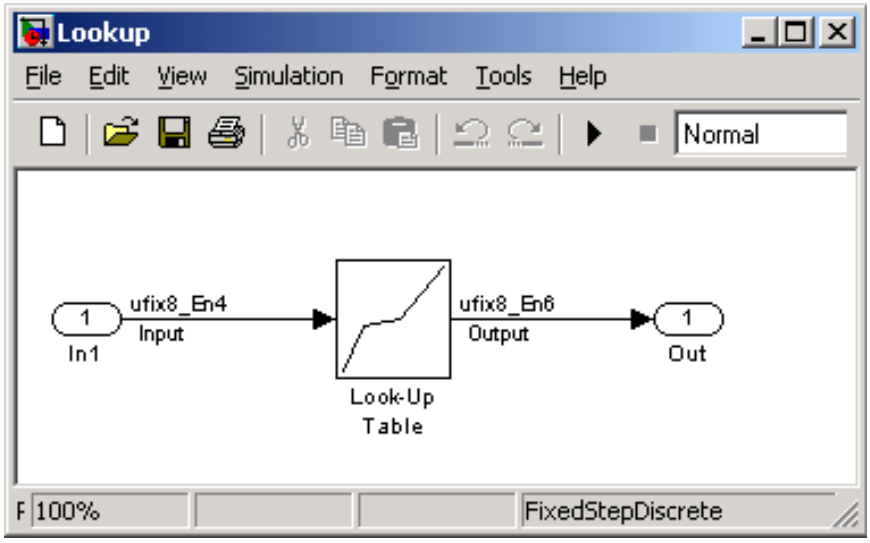

Figure 44: Model Lookup.mdl.

Figure 45 shows the generated code for uneven points along the x (input) axis:
x_breakpoints = [2.5 5 9.5 15]
y_outputs = [0 1.62 1.85 3.67]

Figure 46 shows the generated code for even points along the x (input) axis:
x_breakpoints = [0 5.0 10.0 15.0]
y_outputs = [0 1.62 1.85 3.67]

You can see that the code for the table with evenly spaced axes requires no search function and no memory for each input axis. Note that even better efficiency results if the input-axis spacing is 2^n, where n is any integer.

```
const uint8_T rtcP_Look_Up_Table_XData[4] = { 40U, 80U, 152U, 240U };
const uint8_T rtcP_Look_Up_Table_YData[4] = { 0U, 104U, 118U, 235U };
```

This code is in a different file than the code shown below. Notice that the X- axis data is not in the code for the evenly spaced table.

```
uint8_T Output;
```

The functions `BINARYSEARCH_U8_U8_Below_iL_iR()` and `INTERPOLATE_U8_U8_U8_U8_FLOOR()` appear here but were deleted from this figure to avoid clutter. Notice that the search function is not in the code for the evenly spaced table, shown in the next figure.

```
void Lookup_step(void)
{
  {
    unsigned int iLeft;

    unsigned int iRght;

BINARYSEARCH_U8_U8_Below_iL_iR(&(iLeft),&(iRght),Input,&rtcP_Look_Up_Table
_XData[0],3);

INTERPOLATE_U8_U8_U8_U8_FLOOR(&(Output),(rtcP_Look_Up_Table_YData[iLeft]),
(rtcP_Look_Up_Table_YData[iRght]),

Input,(rtcP_Look_Up_Table_XData[iLeft]),(rtcP_Look_Up_Table_XData[iRght]))
;
  }

}
```

Figure 45: Code generated for unevenly spaced table

```
const uint8 T rtcP Look Up Table YData[4] = { 0U, 104U, 118U, 235U };
```

This code is in a different file than the code shown below.Notice that the x-axis data in the previous figure is not needed in this code.

```
uint8_T Output;

void INTERPOLATE_EVENSPACE_U8_U8_U8 FLOOR(uint8 T *pY, uint8 T yL, uint8 T
yR,
 uint8_T x, uint8_T spacing)
{
  uint16_T bigProd;
  uint8_T yDiff;
  *pY = yL;

  if ( yR >= yL )
  {
    yDiff = yR;
    yDiff -= yL;
  }
  else
  {
    yDiff = yL;
    yDiff -= yR;
  }

  MUL_U16_U8_U8(bigProd,yDiff,x);

  DIV_U8_U16_U8(yDiff,bigProd,spacing);

  if ( yR >= yL )
  {
    *pY += yDiff;
  }
  else
  {
    *pY -= yDiff;
  }
}

void Lookup_step(void)
{

  {
    unsigned int iLeft;

    if ( Input >= 240U )
    {

      Output = (235U);
    }
    else
    {

      iLeft = (unsigned int)( Input ) / 80U;

INTERPOLATE_EVENSPACE_U8_U8_U8_FLOOR(&(Output),(rtcP_Look_Up_Table_YData[i
Left]),(rtcP_Look_Up_Table_YData[((iLeft)+1)]),((uint8_T)((Input)-
iLeft*80U)),80U);
```

Figure 46: Code generated for evenly spaced table

Tip 17. Preventing evenly spaced lookup table from being treated as unevenly spaced

For this example, let X be the decimal number being converted to a binary fixed-point number, and Y be the resolution of the fixed-point number. If X/Y results in a whole number, there will be no quantization error. Otherwise, there will be a quantization error. On page 2 we said that resolution is $1/2^E$. The greater the value of E, the smaller the quantization error.

Often, when Real-Time Workshop Embedded Coder translates an evenly spaced lookup table to a fixed-point lookup table, a quantization error results. That is, the points along an input axis of what began in the model as an evenly spaced lookup table are unevenly spaced in the generated code. This adds an x-axis data statement and a binary search routine to the generated code.

Note: This tip applies only to tables with nontunable input axes. For a table with tunable input axes, this tip provides no benefit. This is due to the fact that tunable input axes always are treated as unevenly spaced.

Observe the model below, for example.

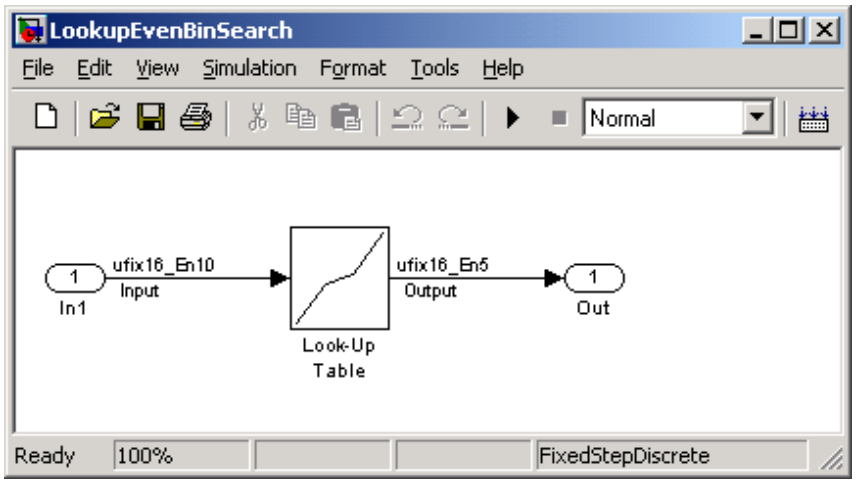

Figure 47: Model LookupEvenBinSearchExample.mdl

The lookup table has these settings:
x_breakpoints = [20.05 20.1 20.15 20.2]
y_outputs = [0 22.3 28.5 52.5]

Notice that the x-axis (input axis) has evenly spaced point value of 0.05. Figure 48 shows the generated code. The fact that the code has an x-axis data statement and a search routine means that the x-axis points in the code are unevenly spaced.

Note: Instead of performing the solution below, we recommend first changing the resolution of the input signal to the lookup table. If this can be done without resulting in an unacceptable range of this input signal, the solution below may be unnecessary.

Here is the solution:

1. Find the script called fixpt_evenspace_cleanup.m, located in the fixpt_tips.zip folder that you downloaded under **Tips for Optimizing the General Solution**. This script contains a function that has the three arguments mentioned in the next step. Place this file in the MATLAB path.
2. On the MATLAB command line, type the following function call:
   fixpt_evenspace_cleanup (x_axis, data_type, resolution), where x_axis is the name in the x-axis field of the Block Parameters dialog for the lookup table, data_type is the data type of the input signal to the lookup table, and resolution is the resolution of the input signal to the lookup table (the $1/2^E$ described on page 2). For our example, you would type:
   fixpt_evenspace_cleanup (x_breakpoints, ufix(16), 2^-10)

3. MATLAB displays new points for the x-axis. Our example results in
   ans =
   20.0498 20.0996 20.1494 20.1992
4. On the MATLAB command line type x_axis =[A B C D], where the arguments A, B, C, and D are the new values that MATLAB displayed in the previous step. For our example, you would type
   x_breakpoints = [20.0498 20.0996 20.1494 20.1992]. This redefines the x-axis points so that there will be negligible quantization error.

The code that results is similiar to that shown in the Figure 49. Notice there is no x-axis data statement and no search routine, indicating that the x-axis points in the code are evenly spaced.

```
const uint16 T rtcP Look Up Table XData[4] = { 20531U, 20582U, 20634U,
20685U };

const uint16_T rtcP_Look_Up_Table_YData[4] = { 0U, 714U, 912U, 1680U };



void LookupEvenBinSearch_step(void)
{

  {
    unsigned int iLeft;

    unsigned int iRght;


BINARYSEARCH_U16_U16_Below_iL_iR(&(iLeft),&(iRght),Input,&rtcP_Look_Up_
Table_XData[0],3);


INTERPOLATE_U16_U16_U16_U16_FLOOR(&(Output),(rtcP_Look_Up_Table_YData[i
Left]),(rtcP_Look_Up_Table_YData[iRght]),

Input,(rtcP_Look_Up_Table_XData[iLeft]),(rtcP_Look_Up_Table_XData[iRght
]));
  }

}
```

Figure 48: Evenly spaced table in model but unevenly spaced table in code.

```
const uint16 T rtcP Look Up Table YData[4] = { 0U, 714U, 912U, 1680U
};


void LookupEvenBinSearch_step(void)
{

  {
    unsigned int iLeft;

    if ( Input <= 20531U )
    {

      Output = (0U);
    }
    else if ( Input >= 20684U )
    {

      Output = (1680U);
    }
    else
    {

      iLeft = (unsigned int)( Input - (20531U) ) / 51U;

INTERPOLATE_EVENSPACE_U16_U16_U16_FLOOR(&(Output),(rtcP_Look_Up_Table_
YData[iLeft]),(rtcP_Look_Up_Table_YData[((iLeft)+1)]),((Input
          - (20531U))-iLeft*51U),51U);
    }
  }

}
```

Figure 49: Evenly spaced table in model and evenly spaced table in code.

Tip 18. Using special features with Stateflow fixed-point operations

There are two special features in Stateflow for performing fixed-point mathematical operations, namely the ":=" operator and the "c" (or "C") qualifier. The := operator yields more precise results. The c qualifier automatically selects the appropriate data type and scaling of a constant, based on context. Using these will produce a higher degree of optimized embedded code. Here is an explanation of each. (See the Stateflow documentation for additional details.)

The := Assignment Operator

Using the := operator in a Stateflow chart instead of the = operator is useful, especially in multiplication and division. The := preserves precision in the result of multiplication or division that = may not retain. The := is less useful in addition and subtraction as to precision. But even here its use can avoid overflow.

The figure below shows a Simulink model in which there is a Stateflow chart that has two inputs and outputs. The content of the Stateflow chart, shown in the next figure, allows us to compare the use of = and :=. The "general" case performs multiplication and division using =. The "better" case performs the same multiplication and division using := instead.

Figure 52 shows the generated code. In the multiplication example, the casting of this 32-bit intermediate result ("`(int16_T)`") occurs before the shifting (">>"), in the general case. This can result in losing useful bits. However, in the better case, the casting takes place after the shifting. This preserves bits, yielding a more accurate result.

Now we will compare the division. In the general case, all division occurs with 16-bit operands, whereas in the := case all division occurs with 32-bit operands. The example using the := operator yields a more accurate result.

Therefore, using := in mathematical operations yields more precise results. However, note that := may increase ROM consumption and execution time compared with using the = operator.

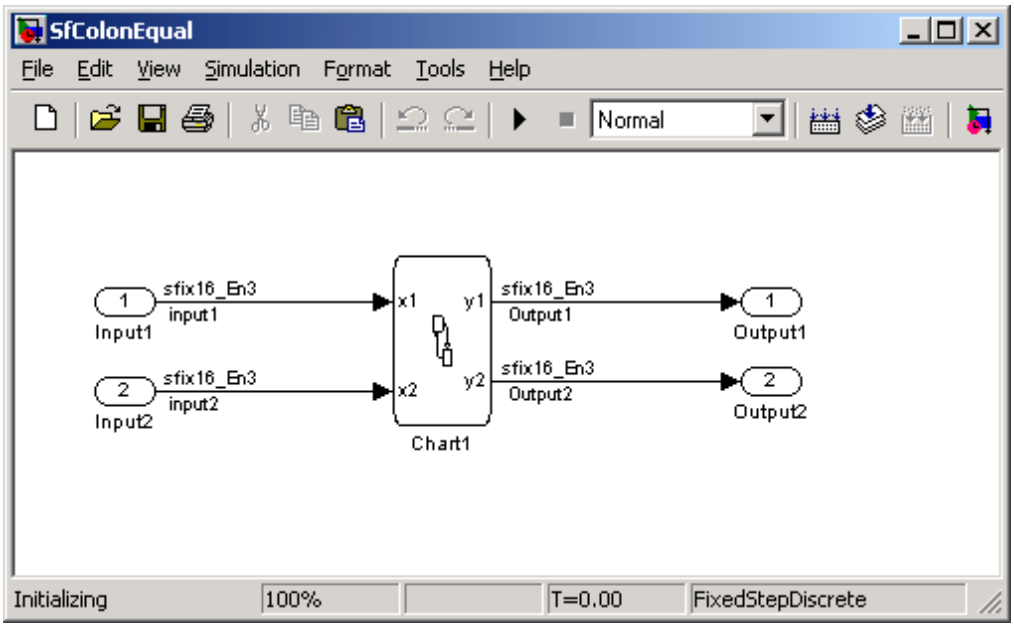

Figure 50: Model SfColonEqual.mdl.

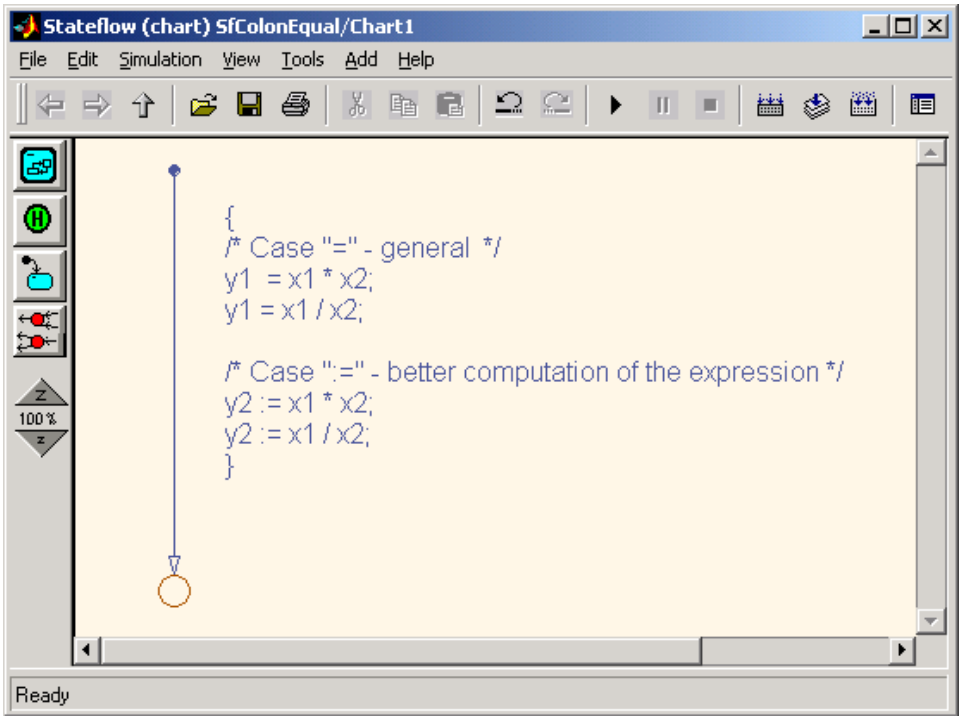

Figure 51: Content of Stateflow chart in SfColonEqual.mdl.

```
/* Model step function */
void SfColonEqual_step(void)
{

  /* Stateflow: '<Root>/Chart1' incorporates:
   *   Inport: '<Root>/Input1'
   *   Inport: '<Root>/Input2'
   */

  {
    /*  Case "=" - general   */
    Output1 = (int16_T)((int32_T)input1 * (int32_T)input2) >> 3;
    Output1 = input1 / input2 << 3;
    /*  Case ":=" - better computation of the expression  */
    Output2 = (int16_T)((int32_T)input1 * (int32_T)input2 >> 3);
    Output2 = (int16_T)(((int32_T)input1 << 3) / (int32_T)input2);
  }

  /* (no update code required) */

}
```

Figure 52: Code comparing the = and := assignment operators.

Using the C qualifier

The figure below shows a Stateflow chart in a Simulink model. The content of this Stateflow chart allows us to compare not using the C qualifier with using it. Notice that the first example, called "simple" in the figure, does not use the C qualifier with the constant 4.5. The second, called "better," does. (The "c" is not case sensitive.)

Figure 55 shows the generated code. In the first equation (simple), since the c qualifier was not specified in the Stateflow chart for the constant 4.5, the addition operation occurs in floating-point. In contrast, since the c qualifier was specified in the second equation (better), the code generator converted the 4.5 constant to its equivalent fixed-point value (36). It did so based on context. "Context" refers to the data type and scaling of the other operands in the expression. In this example, the data type and scaling of the constant is that of the variable "input," namely **`sfix16_En3`**. Therefore, using the c qualifier automatically selects the appropriate data type and scaling of the constant, based on the context. Then the addition occurs with integer mathematics. This decreases ROM consumption and execution time compared with the first example.

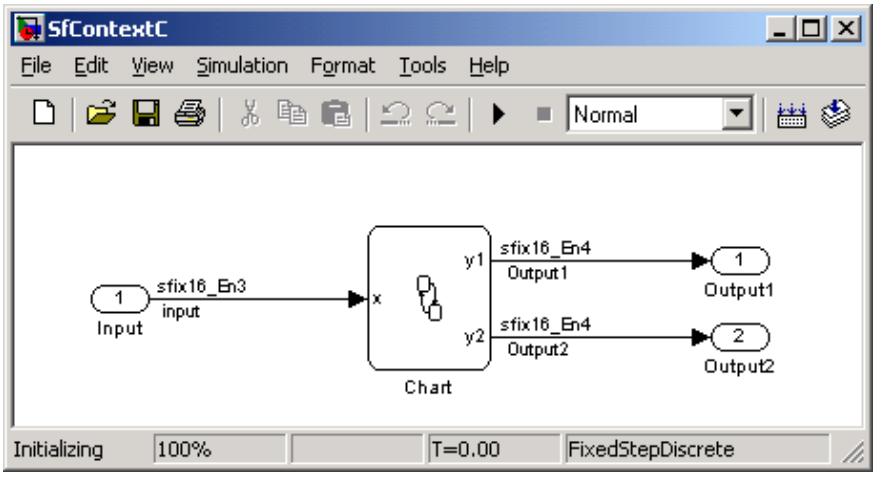

Figure 53: Model SfContextC.mdl

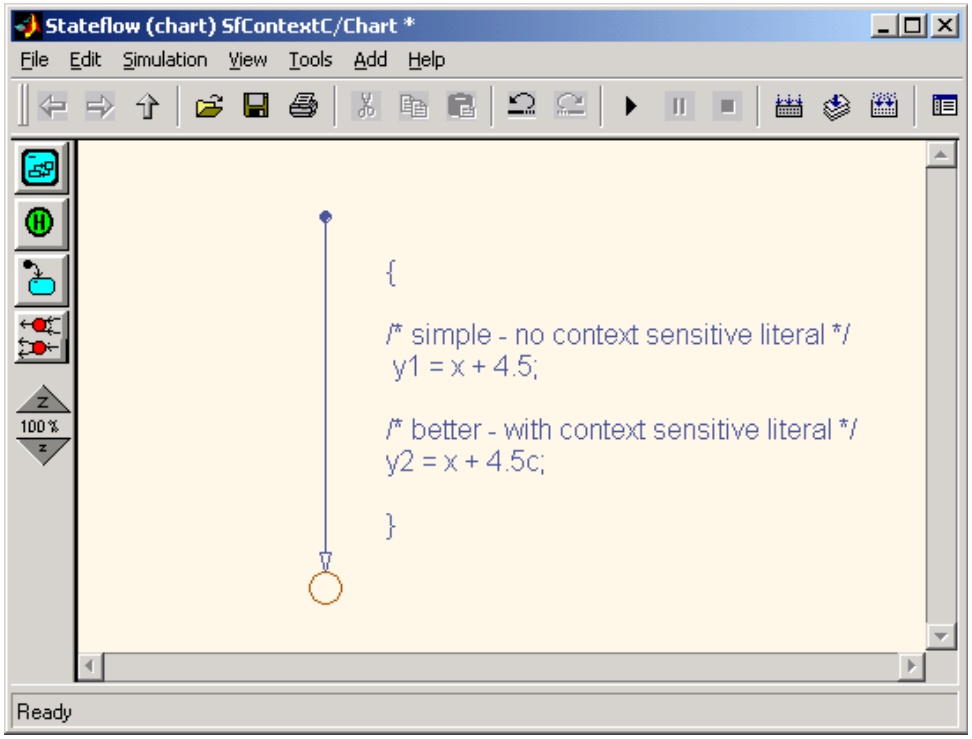

Figure 54: Content of Stateflow chart in SfContextC.mdl.

```
/* Model step function */
void SfContextC_step(void)
{

  /* Stateflow: '<Root>/Chart' incorporates:
   *   Inport: '<Root>/Input'
   */

  {
    /*  simple - no context sensitive literal  */
    Output1 = (int16_T)((ldexp((real_T)input, -3) + 4.5) * 16.0);
    /*  better - with context sensitive literal  */
    Output2 = (input + 36) << 1;
  }

  /* (no update code required) */
```

Figure 55: Code comparing absence and inclusion of C qualifier.

To perform this task:

1. Replace the assignment operator = with the assignment operator := to produce generated code that is optimized for accuracy.
2. Use the C qualifier after literal constants in Stateflow to make the code generator automatically consider the usage context of the constant. Then, the code generator will use the optimal data type for that constant.

Conclusion

There are many challenges faced when programming in fixed-point code manually. Similarly, care is required in automatically generating fixed-point code. The MathWorks has developed tools that enable you to find and understand proven remedies to these difficulties using the General Solution on page 7. However, the core of this document is the tips that start on page 8. These are the result of the combined experience of MathWorks developers and experienced users of MathWorks products. The systematic application of these tips, where recommended, will produce optimum code.