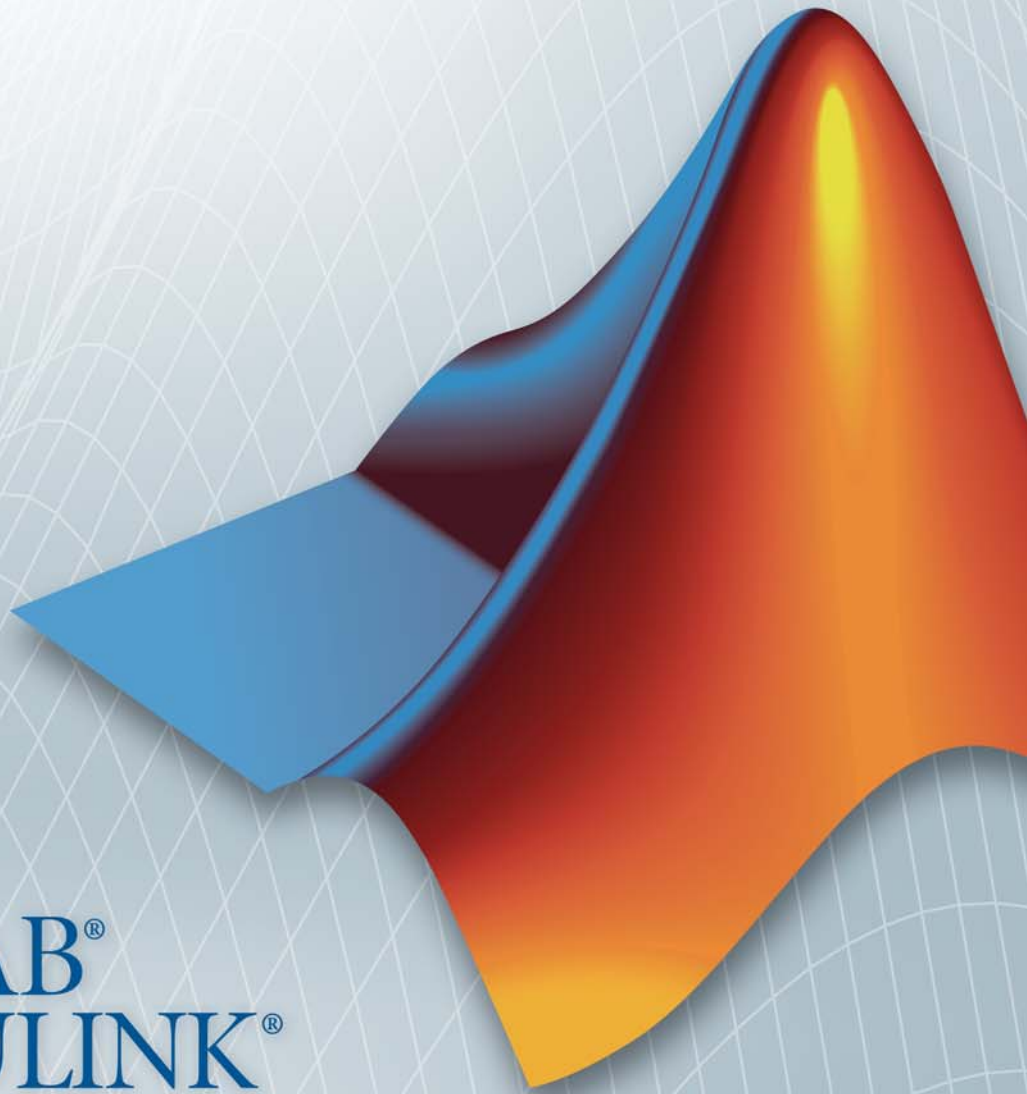
Simulink® Fixed Point™

User's Guide

R2011b



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support
suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Simulink® Fixed Point™ User's Guide

Revision History

March 1995	First Printing	New for Version 1.0
April 1997	Second Printing	Revised for MATLAB 5
January 1999	Third Printing	Revised for MATLAB 5.3 (Release 11)
September 2000	Fourth Printing	New for Version 3.0 (Release 12)
August 2001	Fifth Printing	Minor revisions for Version 3.1 (Release 12.1)
November 2002	Sixth Printing	Minor revisions for Version 4.0 (Release 13)
June 2004	Seventh Printing	Revised for Version 5.0 (Release 14) (Renamed from Fixed-Point Blockset)
October 2004	Online only	Minor revisions for Version 5.0.1 (Release 14SP1)
March 2005	Online only	Minor revisions for Version 5.1 (Release 14SP2)
September 2005	Online only	Minor revisions for Version 5.1.2 (Release 14SP3)
March 2006	Online only	Revised for Version 5.2 (Release 2006a)
May 2006	Online only	Revised for Version 5.2.1 (Release 2006a+)
September 2006	Online only	Revised for Version 5.3 (Release 2006b)
March 2007	Eighth Printing	Revised for Version 5.4 (Release 2007a)
September 2007	Online only	Revised for Version 5.5 (Release 2007b)
March 2008	Online only	Revised for Version 5.6 (Release 2008a)
October 2008	Online only	Revised for Version 6.0 (Release 2008b)
March 2009	Online only	Revised for Version 6.1 (Release 2009a)
September 2009	Online only	Revised for Version 6.2 (Release 2009b)
March 2010	Online only	Revised for Version 6.3 (Release 2010a)
September 2010	Online only	Revised for Version 6.4 (Release 2010b)
April 2011	Online only	Revised for Version 6.5 (Release 2011a)
September 2011	Online only	Revised for Version 7 (Release 2011b)

Contents

1 Getting Started

2 Data Types and Scaling

3 Arithmetic Operations

4 Realization Structures

5 Fixed-Point Advisor

6 Fixed-Point Tool

7 Tutorial: Conversion to Fixed Point Workflow

8 Tutorial: Producing Lookup Table Data

9 Automatic Data Typing

10 Range Analysis

11 Code Generation

12 Fixed-Point Advisor Reference

A Writing Fixed-Point S-Functions

Index

1

Getting Started

Product Overview

Simulink® Fixed Point™ enables the fixed-point capabilities of the Simulink® product family, letting you use those products to design, simulate, and implement fixed-point control and signal processing algorithms.

With Simulink Fixed Point, you specify fixed-point data attributes, including word length and scaling for signals and parameters, in your model. You can perform bit-true simulations to observe the effects of limited range and precision on designs built with Simulink, Stateflow®, DSP System Toolbox™, and other Simulink products. Automated fixed-point advisors guide you through the steps of converting floating-point models to fixed point. Additional tools analyze your model or use simulation results to recommend data types and scaling.

Simulink Fixed Point supports C, HDL, and PLC code generation with Simulink code-generation products.

You can use the Simulink Fixed Point software with Simulink products to simulate effects commonly encountered in fixed-point systems for applications such as control systems and time-domain filtering. The Simulink Fixed Point software includes these major features:

- Integer, fractional, and generalized fixed-point data types
  - Unsigned and two's complement formats
  - Word sizes in simulation from 1 to 128 bits

- Floating-point data types
  - IEEE-style singles and doubles
  - A nonstandard IEEE-style data type, where the fraction can range from 1 to 52 bits and the exponent can range from 1 to 11 bits
- Methods for overflow handling, scaling, and rounding of fixed-point data types
- Tools that facilitate
  - Collection of minimum and maximum simulation values
  - Optimization of scaling parameters
  - Display of input and output signals

In addition, you can generate C code and HDL code for execution on a fixed-point embedded processor with the Simulink® Coder™ product. The generated code uses only integer types and automatically includes all operations, such as shifts, needed to account for differences in fixed-point locations.

The Simulink Fixed Point software features listed above are all supported for fixed-point Simulink blocks. Other products in the Simulink family with fixed-point capabilities might support some or all of these features. To get specific information about the fixed-point features supported by a particular product, refer to the documentation for that product. For example,

- For information on fixed-point support in the DSP System Toolbox software, refer to “Fixed-Point Design” in the DSP System Toolbox documentation.
- For information on fixed-point support in the Stateflow software, refer to “Using Fixed-Point Data in Stateflow Charts” in the Stateflow documentation.

What You Need to Get Started

In this section...

Installation

To determine if the Simulink Fixed Point software is installed on your system, type

```
ver
```

at the MATLAB® command line. When you enter this command, the MATLAB Command Window displays information about the version of MATLAB software you are running, including a list of installed add-on products and their version numbers. Check the list to see if the Simulink Fixed Point software appears.

For information about installing this product, refer to the installation documentation.

If you experience installation difficulties and have Web access, look for the installation and license information at the MathWorks® Web site (`http://www.mathworks.com/support`).

Sharing Fixed-Point Models

You can edit a model containing fixed-point blocks without the Simulink Fixed Point software. However, you must have a Simulink Fixed Point software license to

- Update a Simulink diagram (**Ctrl+D**) containing fixed-point data types
- Run a model containing fixed-point data types
- Generate code from a model containing fixed-point data types
- Log the minimum and maximum values produced by a simulation

- Automatically scale the output of a model using the autoscaling tool

If you do not have the Simulink Fixed Point software, you can work with a model containing Simulink blocks with fixed-point settings as follows:

1. In the **Model Hierarchy** pane, select the root model.
2. From the Simulink model **Tools** menu, select **Fixed-Point > Fixed-Point Tool**.

   The Fixed-Point Tool appears.

   - Set the **Fixed-point instrumentation mode** parameter to `Force Off`.
   - Set the **Data type override** parameter to `Double` or `Single`.
   - Set the **Data type override applies to** parameter to `All numeric types`.
3. If you use `fi` objects or embedded numeric data types in your model, set the `fipref DataTypeOverride` property to `TrueDoubles` and the `DataTypeOverride` property to `All numeric types`.

   At the MATLAB command line, enter:

```
p = fipref('DataTypeOverride', 'TrueDoubles', ...
 'DataTypeOverrideAppliesTo', 'AllNumericTypes');
```

Note If you use fi objects or embedded numeric data types in your model or workspace, you might introduce fixed-point data types into your model. You can set fipref to prevent the checkout of a Fixed-Point Toolbox™ license. For more information, see "Licensing" in the Fixed-Point Toolbox documentation.

Demos

To help you learn how to use the Simulink Fixed Point software, a collection of demos is provided. You can explore specific features of the product by changing the parameters of Simulink blocks with fixed-point support and observing the effects of those changes.

The demos are divided into the following groups:

- Application Examples
- Feature Demonstrations
- Filters
- Tools and Utilities
- Custom S-Function Examples

All demos are located in the `fxpdemos` directory.

To view the complete list of demos, see Simulink Fixed Point Demos.

Physical Quantities and Measurement Scales

In this section...

Introduction

The decision to use fixed-point hardware is simply a choice to represent numbers in a particular form. This representation often offers advantages in terms of the power consumption, size, memory usage, speed, and cost of the final product.

A measurement of a physical quantity can take many numerical forms. For example, the boiling point of water is 100 degrees Celsius, 212 degrees Fahrenheit, 373 kelvin, or 671.4 degrees Rankine. No matter what number is given, the physical quantity is exactly the same. The numbers are different because four different scales are used.

Well known standard scales like Celsius are very convenient for the exchange of information. However, there are situations where it makes sense to create and use unique nonstandard scales. These situations usually involve making the most of a limited resource.

For example, nonstandard scales allow map makers to get the maximum detail on a fixed size sheet of paper. A typical road atlas of the USA will show each state on a two-page display. The scale of inches to miles will be unique for most states. By using a large ratio of miles to inches, all of Texas can fit on two pages. Using the same scale for Rhode Island would make poor use of the page. Using a much smaller ratio of miles to inches would allow Rhode Island to be shown with the maximum possible detail.

Fitting measurements of a variable inside an embedded processor is similar to fitting a state map on a piece of paper. The map scale should allow all the boundaries of the state to fit on the page. Similarly, the binary scale for a measurement should allow the maximum and minimum possible values to fit. The map scale should also make the most of the paper in order to get

maximum detail. Similarly, the binary scale for a measurement should make the most of the processor in order to get maximum precision.

Use of standard scales for measurements has definite compatibility advantages. However, there are times when it is worthwhile to break convention and use a unique nonstandard scale. There are also occasions when a mix of uniqueness and compatibility makes sense. See the sections that follow for more information.

Selecting a Measurement Scale

Suppose that you want to make measurements of the temperature of liquid water, and that you want to represent these measurements using 8-bit unsigned integers. Fortunately, the temperature range of liquid water is limited. No matter what scale you use, liquid water can only go from the freezing point to the boiling point. Therefore, this is the range of temperatures that you must capture using just the 256 possible 8-bit values: 0,1,2,...,255.

One approach to representing the temperatures is to use a standard scale. For example, the units for the integers could be Celsius. Hence, the integers 0 and 100 represent water at the freezing point and at the boiling point, respectively. On the upside, this scale gives a trivial conversion from the integers to degrees Celsius. On the downside, the numbers 101 to 255 are unused. By using this standard scale, more than 60% of the number range has been wasted.

A second approach is to use a nonstandard scale. In this scale, the integers 0 and 255 represent water at the freezing point and at the boiling point, respectively. On the upside, this scale gives maximum precision since there are 254 values between freezing and boiling instead of just 99. On the downside, the units are roughly 0.3921568 degree Celsius per bit so the conversion to Celsius requires division by 2.55, which is a relatively expensive operation on most fixed-point processors.

A third approach is to use a “semistandard” scale. For example, the integers 0 and 200 could represent water at the freezing point and at the boiling point, respectively. The units for this scale are 0.5 degrees Celsius per bit. On the downside, this scale doesn’t use the numbers from 201 to 255, which represents a waste of more than 21%. On the upside, this scale permits relatively easy conversion to a standard scale. The conversion to Celsius involves division by 2, which is a very easy shift operation on most processors.

Measurement Scales: Beyond Multiplication

One of the key operations in converting from one scale to another is multiplication. The preceding case study gave three examples of conversions from a quantized integer value Q to a real-world Celsius value V that involved only multiplication:

$$V = \begin{cases} \dfrac{100^{\circ}\mathrm{C}}{100\ \mathrm{bits}} Q_1 & \text{Conversion 1} \\ \dfrac{100^{\circ}\mathrm{C}}{255\ \mathrm{bits}} Q_2 & \text{Conversion 2} \\ \dfrac{100^{\circ}\mathrm{C}}{200\ \mathrm{bits}} Q_3 & \text{Conversion 3} \end{cases}$$

Graphically, the conversion is a line with slope S, which must pass through the origin. A line through the origin is called a purely linear conversion. Restricting yourself to a purely linear conversion can be very wasteful and it is often better to use the general equation of a line:

$$V = SQ + B.$$

By adding a bias term B, you can obtain greater precision when quantizing to a limited number of bits.

The general equation of a line gives a very useful conversion to a quantized scale. However, like all quantization methods, the precision is limited and errors can be introduced by the conversion. The general equation of a line with quantization error is given by

$$V = SQ + B \pm Error.$$

If the quantized value Q is rounded to the nearest representable number, then

$$-\frac{S}{2} \le Error \le \frac{S}{2}.$$

That is, the amount of quantization error is determined by both the number of bits and by the scale. This scenario represents the best-case error. For other rounding schemes, the error can be twice as large.

Example: Selecting a Measurement Scale

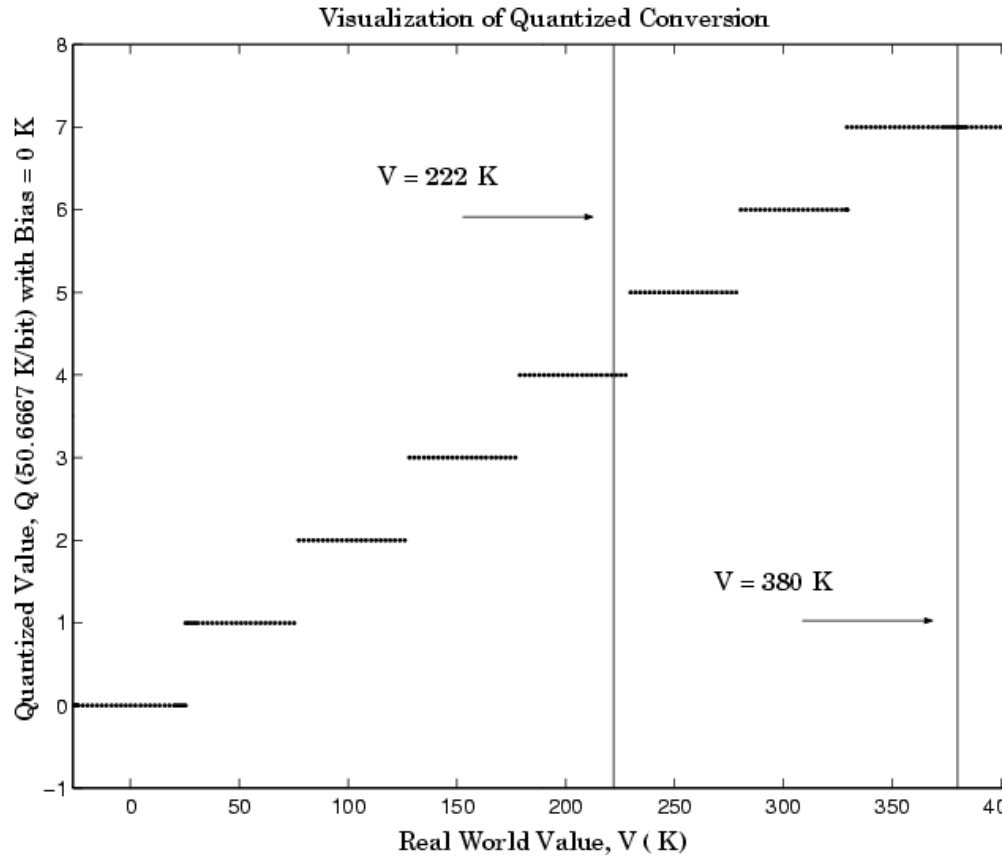
On typical electronically controlled internal combustion engines, the flow of fuel is regulated to obtain the desired ratio of air to fuel in the cylinders just prior to combustion. Therefore, knowledge of the current air flow rate is required. Some manufacturers use sensors that directly measure air flow, while other manufacturers calculate air flow from measurements of related signals. The relationship of these variables is derived from the ideal gas equation. The ideal gas equation involves division by air temperature. For proper results, an absolute temperature scale such as kelvin or Rankine must be used in the equation. However, quantization directly to an absolute temperature scale would cause needlessly large quantization errors.

The temperature of the air flowing into the engine has a limited range. On a typical engine, the radiator is designed to keep the block below the boiling point of the cooling fluid. Assume a maximum of 225°F (380 K). As the air flows through the intake manifold, it can be heated to this maximum temperature. For a cold start in an extreme climate, the temperature can be as low as -60°F (222 K). Therefore, using the absolute kelvin scale, the range of interest is 222 K to 380 K.

The air temperature needs to be quantized for processing by the embedded control system. Assuming an unrealistic quantization to 3-bit unsigned numbers: 0,1,2,...,7, the purely linear conversion with maximum precision is

$$V = \frac{380\ \mathrm{K}}{7.5\ \mathrm{bit}} Q.$$

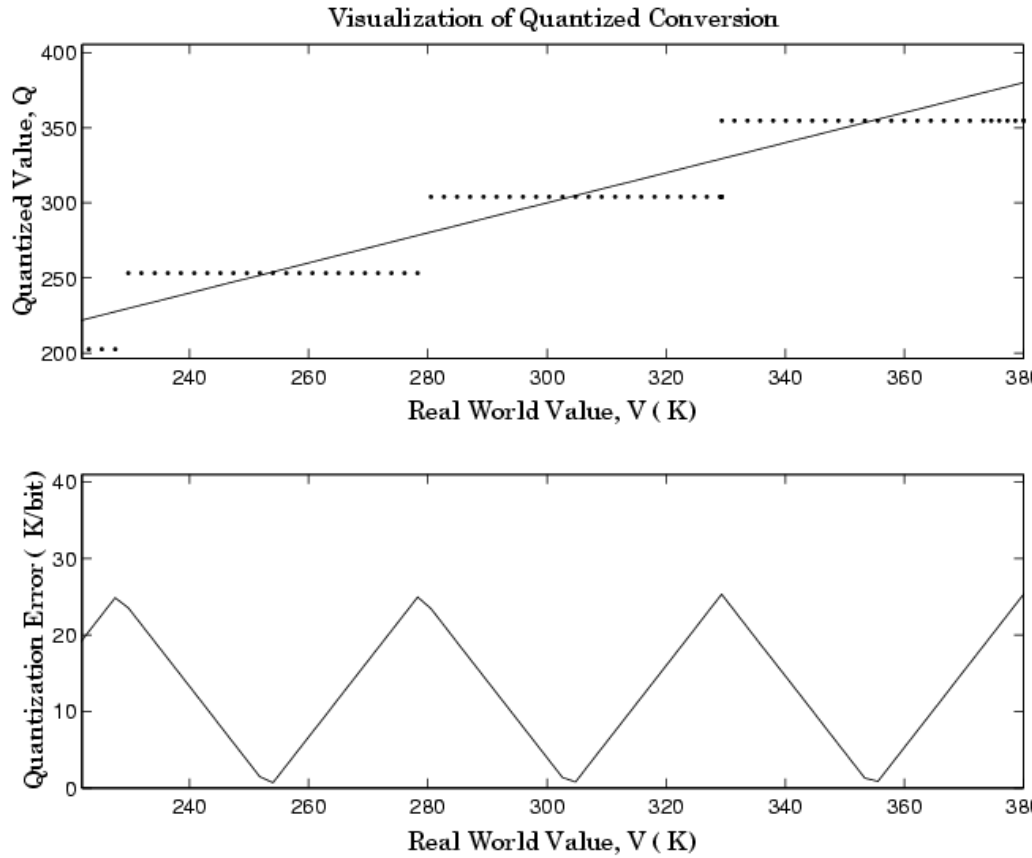
The quantized conversion and range of interest are shown in the following figure.



Notice that there are 7.5 possible quantization values. This is because only half of the first bit corresponds to temperatures (real-world values) greater than zero.

The quantization error is -25.33 K/bit $\leq$ *Error* $\leq$ 25.33 K/bit.

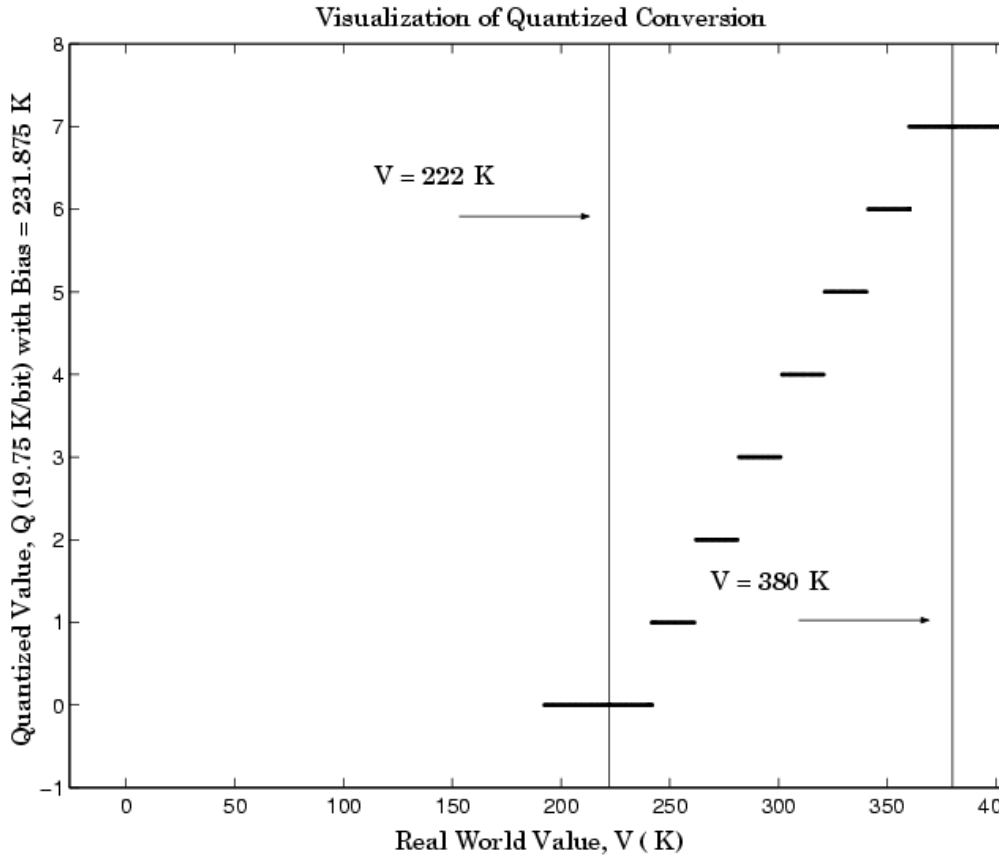
The range of interest of the quantized conversion and the absolute value of the quantized error are shown in the following figure.



As an alternative to the purely linear conversion, consider the general linear conversion with maximum precision:

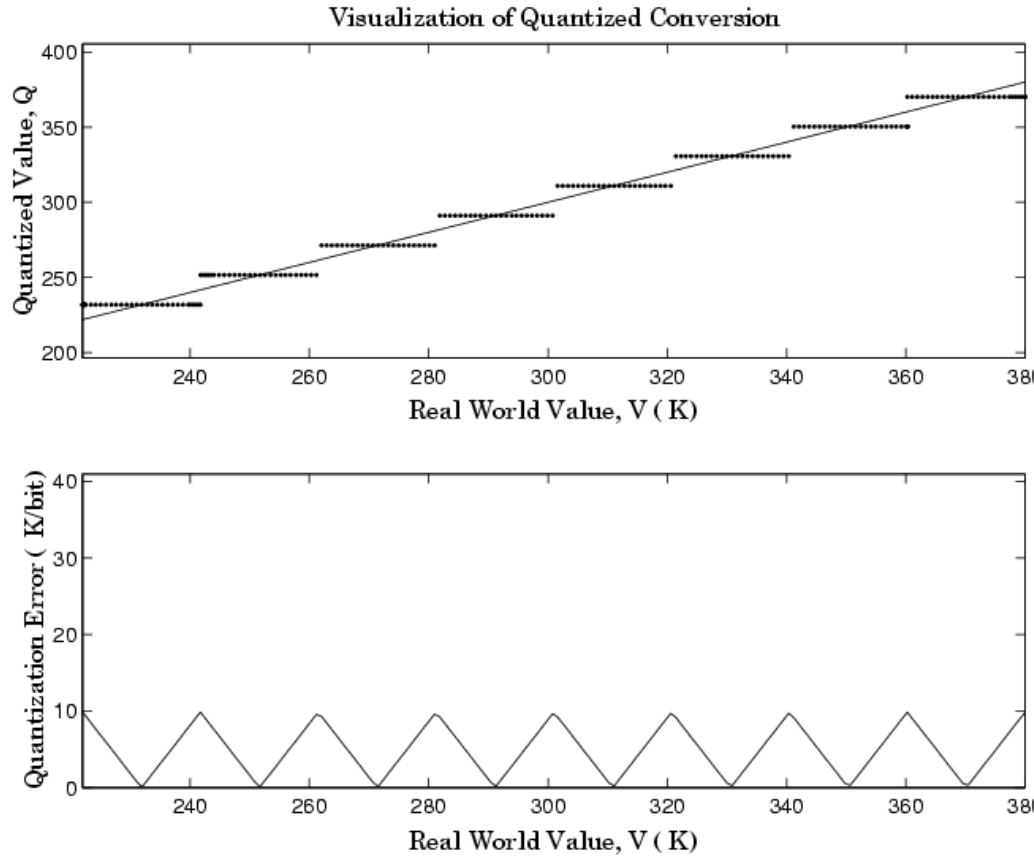
$$V = \left(\frac{380\ \mathrm{K} - 222\ \mathrm{K}}{8}\right)Q + 222\ \mathrm{K} + 0.5\left(\frac{380\ \mathrm{K} - 222\ \mathrm{K}}{8}\right)$$

The quantized conversion and range of interest are shown in the following figure.



The quantization error is -9.875 K/bit ≤ *Error* ≤ 9.875 K/bit, which is approximately 2.5 times smaller than the error associated with the purely linear conversion.

The range of interest of the quantized conversion and the absolute value of the quantized error are shown in the following figure.



Clearly, the general linear scale gives much better precision than the purely linear scale over the range of interest.

Why Use Fixed-Point Hardware?

Digital hardware is becoming the primary means by which control systems and signal processing filters are implemented. Digital hardware can be classified as either off-the-shelf hardware (for example, microcontrollers, microprocessors, general-purpose processors, and digital signal processors) or custom hardware. Within these two types of hardware, there are many architecture designs. These designs range from systems with a single instruction, single data stream processing unit to systems with multiple instruction, multiple data stream processing units.

Within digital hardware, numbers are represented as either fixed-point or floating-point data types. For both these data types, word sizes are fixed at a set number of bits. However, the dynamic range of fixed-point values is much less than floating-point values with equivalent word sizes. Therefore, in order to avoid overflow or unreasonable quantization errors, fixed-point values must be scaled. Since floating-point processors can greatly simplify the real-time implementation of a control law or digital filter, and floating-point numbers can effectively approximate real-world numbers, then why use a microcontroller or processor with fixed-point hardware support?

- **Size and Power Consumption** — The logic circuits of fixed-point hardware are much less complicated than those of floating-point hardware. This means that the fixed-point chip size is smaller with less power consumption when compared with floating-point hardware. For example, consider a portable telephone where one of the product design goals is to make it as portable (small and light) as possible. If one of today's high-end floating-point, general-purpose processors is used, a large heat sink and battery would also be needed, resulting in a costly, large, and heavy portable phone.
- **Memory Usage and Speed** — In general fixed-point calculations require less memory and less processor time to perform.
- **Cost** — Fixed-point hardware is more cost effective where price/cost is an important consideration. When digital hardware is used in a product, especially mass-produced products, fixed-point hardware costs much less than floating-point hardware and can result in significant savings.

After making the decision to use fixed-point hardware, the next step is to choose a method for implementing the dynamic system (for example, control

system or digital filter). Floating-point software emulation libraries are generally ruled out because of timing or memory size constraints. Therefore, you are left with fixed-point math where binary integer values are scaled.

Why Use the Simulink Fixed Point Software?

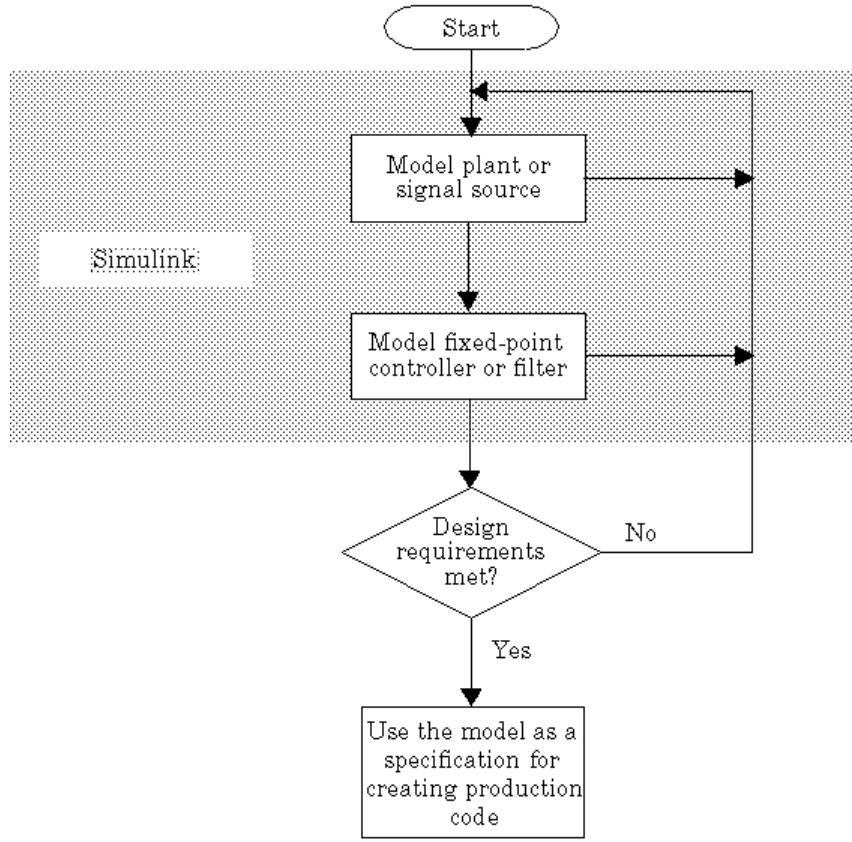
The Simulink Fixed Point software allows you to efficiently design control systems and digital filters that you will implement using fixed-point arithmetic. With the Simulink Fixed Point software, you can construct Simulink and Stateflow models that contain detailed fixed-point information about your systems. You can then perform bit-true simulations with the models to observe the effects of limited range and precision on your designs.

You can configure the Fixed-Point Tool to automatically log the overflows, saturations, and signal extremes of your simulations. You can also use it to automate scaling decisions and to compare your fixed-point implementations against idealized, floating-point benchmarks.

You can use the Simulink Fixed Point software with the Simulink Coder product to automatically generate efficient, integer-only C code representations of your designs. You can use this C code in a production target or for rapid prototyping. You can also use the Simulink Fixed Point software with the Embedded Coder™ product to generate real-time C code for use on an integer production, embedded target.

The Development Cycle

The Simulink Fixed Point software provides tools that aid in the development and testing of fixed-point dynamic systems. You directly design dynamic system models in the Simulink software that are ready for implementation on fixed-point hardware. The development cycle is illustrated below.



Using the MATLAB, Simulink, and Simulink Fixed Point software, you follow these steps of the development cycle:

1. Model the system (plant or signal source) within the Simulink software using double-precision numbers. Typically, the model will contain nonlinear elements.

2. Design and simulate a fixed-point dynamic system (for example, a control system or digital filter) with fixed-point Simulink blocks that meets the design, performance, and other constraints.

3. Analyze the results and go back to step 1 if needed.

When you have met the design requirements, you can use the model as a specification for creating production code using the Simulink Coder product.

The above steps interact strongly. In steps 1 and 2, there is a significant amount of freedom to select different solutions. Generally, you fine-tune the model based upon feedback from the results of the current implementation (step 3). There is no specific modeling approach. For example, you may obtain models from first principles such as equations of motion, or from a frequency response such as a sine sweep. There are many controllers that meet the same frequency-domain or time-domain specifications. Additionally, for each controller there are an infinite number of realizations.

The Simulink Fixed Point software helps expedite the design cycle by allowing you to simulate the effects of various fixed-point controller and digital filter structures.

Data Type Support

The Simulink Fixed Point software supports the following integer and fixed-point data types for simulation and code generation:

- Unsigned data types from 1 to 128 bits
- Signed data types from 2 to 128 bits
- Boolean, double, and single
- Scaled doubles

The software supports all scaling choices including pure integer, binary point, and slope bias. For slope bias scaling, it does not support complex fixed-point types that have non-zero bias or non-trivial slope.

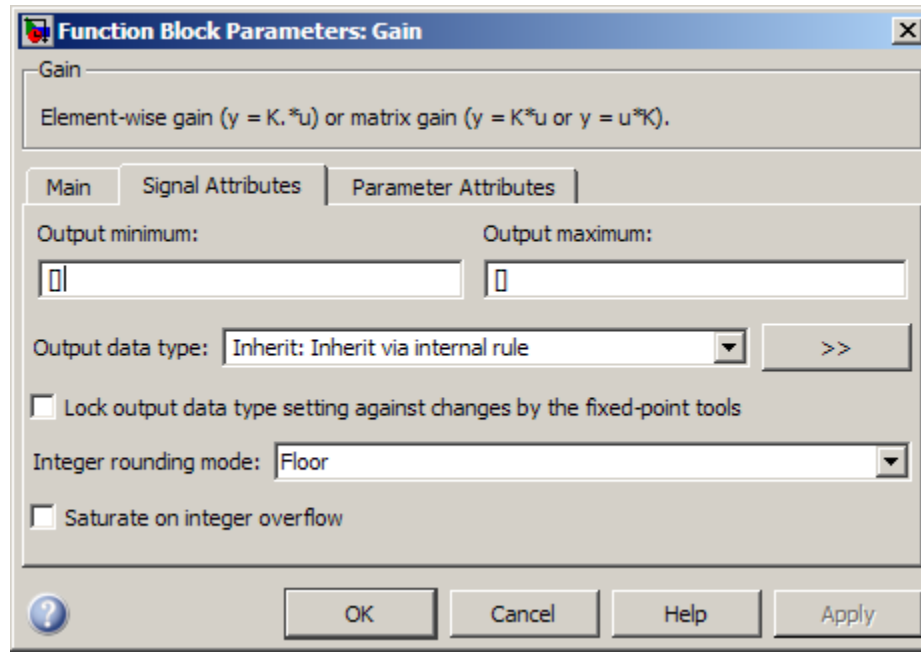
The save data type support extends to signals, parameters, and states.

Simulink Fixed Point Software Features

In this section...

Configuring Blocks with Fixed-Point Output

You can create a fixed-point model by configuring Simulink blocks to output fixed-point signals. Simulink blocks that support fixed-point output provide parameters that allow you to specify whether a block should output fixed-point signals and, if so, the size, scaling, and other attributes of the fixed-point output. These parameters typically appear on the **Signal Attributes** pane of the block's parameter dialog box.



The following sections explain how to use these parameters to configure a block for fixed-point output.

- "Specifying the Output Data Type and Scaling" on page 1-22
- "Specifying Fixed-Point Data Types with the Data Type Assistant" on page 1-25
- "Rounding" on page 1-28
- "Overflow Handling" on page 1-29
- "Locking the Output Data Type Setting" on page 1-29
- "Real-World Values Versus Stored Integer Values" on page 1-29

Specifying the Output Data Type and Scaling

Many Simulink blocks allow you to specify an output data type and scaling using a parameter that appears on the block dialog box. This parameter (typically named **Output data type**) provides a pull-down menu that lists the

data types a particular block supports. In general, you can specify the output data type as a rule that inherits a data type, a built-in data type, an expression that evaluates to a data type, or a Simulink data type object. See "Specifying Block Output Data Types" in *Simulink User's Guide* for more information.

The Simulink Fixed Point software enables you to configure Simulink blocks with:

- **Fixed-point data types**

  Fixed-point data types are characterized by their word size in bits and by their binary point—the means by which fixed-point values are scaled. See "Fixed-Point Numbers" on page 2-3 for more information.

- **Floating-point data types**

  Floating-point data types are characterized by their sign bit, fraction (mantissa) field, and exponent field. See "Floating-Point Numbers" on page 2-23 for more information.

To configure blocks with Simulink Fixed Point data types, specify the data type parameter on a block dialog box as an expression that evaluates to a data type. Alternatively, you can use an assistant that simplifies the task of entering data type expressions (see "Specifying Fixed-Point Data Types with the Data Type Assistant" on page 1-25). The sections that follow describe varieties of fixed-point and floating-point data types, and the corresponding functions that you use to specify them.

Integers. You can specify unsigned and signed integers with the `uint` and `sint` functions, respectively.

For example, to configure a 16-bit unsigned integer via the block dialog box, specify the **Output data type** parameter as `uint(16)`. To configure a 16-bit signed integer, specify the **Output data type** parameter as `sint(16)`.

For integer data types, the default binary point is assumed to lie to the right of all bits.

Fractional Numbers. You can specify unsigned and signed fractional numbers with the `ufrac` and `sfrac` functions, respectively.

For example, to configure the output as a 16-bit unsigned fractional number via the block dialog box, specify the **Output data type** parameter to be `ufrac(16)`. To configure a 16-bit signed fractional number, specify **Output data type** to be `sfrac(16)`.

Fractional numbers are distinguished from integers by their default scaling. Whereas signed and unsigned integer data types have a default binary point to the right of all bits, unsigned fractional data types have a default binary point to the left of all bits, while signed fractional data types have a default binary point to the right of the sign bit.

Both unsigned and signed fractional data types support *guard bits*, which act to guard against overflow. For example, `sfrac(16,4)` specifies a 16-bit signed fractional number with 4 guard bits. The guard bits lie to the left of the default binary point.

Generalized Fixed-Point Numbers. You can specify unsigned and signed generalized fixed-point numbers with the `ufix` and `sfix` functions, respectively.

For example, to configure the output as a 16-bit unsigned generalized fixed-point number via the block dialog box, specify the **Output data type** parameter to be `ufix(16)`. To configure a 16-bit signed generalized fixed-point number, specify **Output data type** to be `sfix(16)`.

Generalized fixed-point numbers are distinguished from integers and fractionals by the absence of a default scaling. For these data types, a block typically inherits its scaling from another block.

Note Alternatively, you can use the `fixdt` function to create integer, fractional, and generalized fixed-point objects. The `fixdt` function also allows you to specify scaling for fixed-point data types.

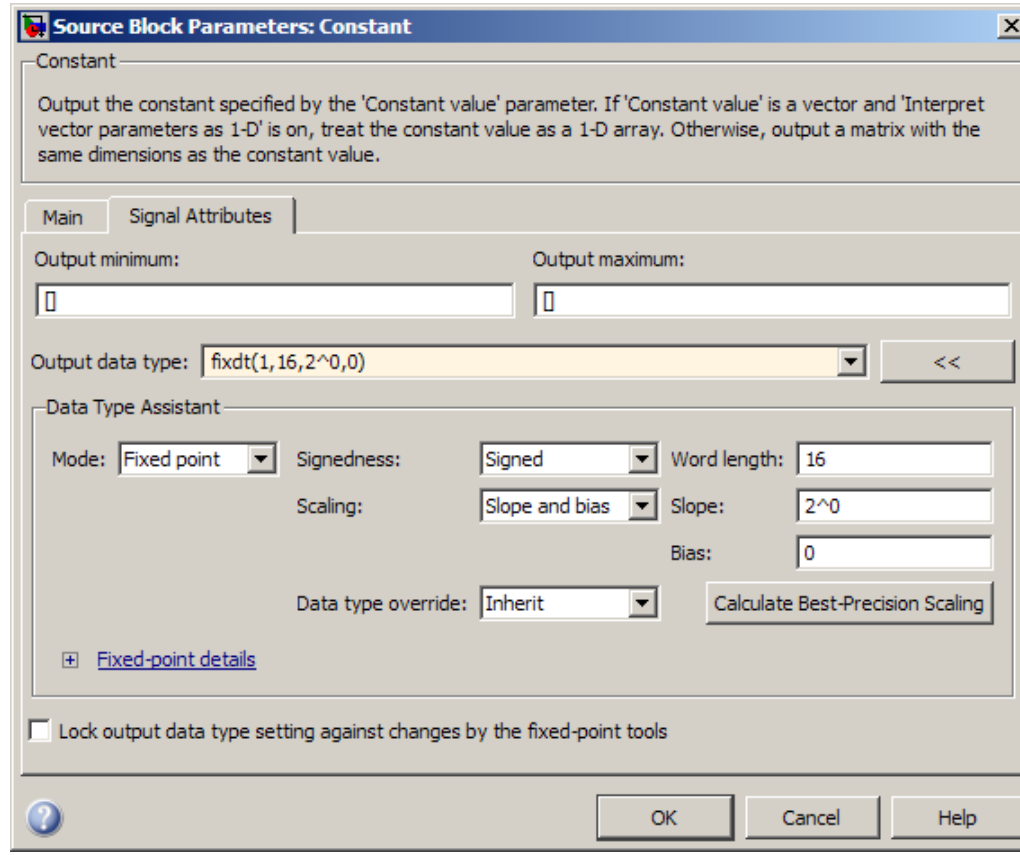
Floating-Point Numbers. The Simulink Fixed Point software supports single-precision and double-precision floating-point numbers as defined by the IEEE® Standard 754-1985 for Binary Floating-Point Arithmetic. You can specify floating-point numbers with the Simulink `float` function.

For example, to configure the output as a single-precision floating-point number via the block dialog box, specify the **Output data type** parameter as `float('single')`. To configure a double-precision floating-point number, specify **Output data type** as `float('double')`.

Specifying Fixed-Point Data Types with the Data Type Assistant

The **Data Type Assistant** is an interactive graphical tool that simplifies the task of specifying data types for Simulink blocks and data objects. The assistant appears on block and object dialog boxes, adjacent to parameters that provide data type control, such as the **Output data type** parameter. For more information about accessing and interacting with the assistant, see "Using the Data Type Assistant" in *Simulink User's Guide*.

You can use the **Data Type Assistant** to specify a fixed-point data type. When you select `Fixed point` in the **Mode** field, the assistant displays fields for describing additional attributes of a fixed-point data type, as shown in this example:

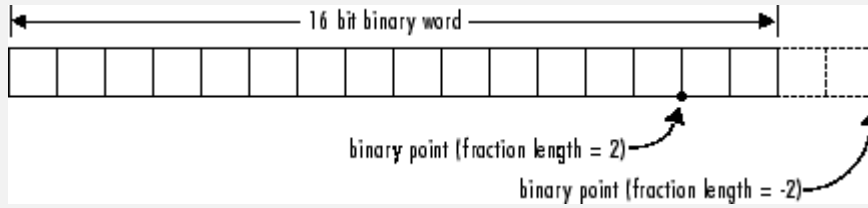


You can set the following fixed-point attributes:

Signedness. Select whether you want the fixed-point data to be `Signed` or `Unsigned`. Signed data can represent positive and negative quantities. Unsigned data represents positive values only.

Word length. Specify the size (in bits) of the word that will hold the quantized integer. Large word sizes represent large quantities with greater precision than small word sizes. Fixed-point word sizes up to 128 bits are supported for simulation.

Scaling. Specify the method for scaling your fixed-point data to avoid overflow conditions and minimize quantization errors. You can select the following scaling modes:

Scaling Mode	Description
`Binary point`	If you select this mode, the assistant displays the **Fraction length** field, specifying the binary point location. Binary points can be positive or negative integers. A positive integer moves the binary point left of the rightmost bit by that amount. For example, an entry of 2 sets the binary point in front of the second bit from the right. A negative integer moves the binary point further right of the rightmost bit by that amount, as in this example:  See “Binary-Point-Only Scaling” on page 2-6 for more information.
`Slope and bias`	If you select this mode, the assistant displays fields for entering the **Slope** and **Bias**. • Slope can be any *positive* real number. • Bias can be any real number. See “Slope and Bias Scaling” on page 2-6 for more information.
`Best precision`	If you select this mode, the block scales a constant vector or matrix such that the precision of its elements is maximized. This mode is available only for particular blocks. See “Constant Scaling for Best Precision” on page 2-12 for more information.

Calculate Best-Precision Scaling. The Simulink Fixed Point software can automatically calculate "best-precision" values for both `Binary point` and `Slope and bias` scaling, based on the values that you specify for other parameters on the dialog box. To calculate best-precision-scaling values automatically, enter values for the block's **Output minimum** and **Output maximum** parameters. Afterward, click the **Calculate Best-Precision Scaling** button in the assistant.

Rounding

You specify how fixed-point numbers are rounded with the **Integer rounding mode** parameter. The following rounding modes are supported:

- `Ceiling` — This mode rounds toward positive infinity and is equivalent to the MATLAB `ceil` function.
- `Convergent` — This mode rounds toward the nearest representable number, with ties rounding to the nearest even integer. Convergent rounding is equivalent to the Fixed-Point Toolbox `convergent` function.
- `Floor` — This mode rounds toward negative infinity and is equivalent to the MATLAB `floor` function.
- `Nearest` — This mode rounds toward the nearest representable number, with the exact midpoint rounded toward positive infinity. Rounding toward nearest is equivalent to the Fixed-Point Toolbox `nearest` function.
- `Round` — This mode rounds to the nearest representable number, with ties for positive numbers rounding in the direction of positive infinity and ties for negative numbers rounding in the direction of negative infinity. This mode is equivalent to the Fixed-Point Toolbox `round` function.
- `Simplest` — This mode automatically chooses between round toward floor and round toward zero to produce generated code that is as efficient as possible.
- `Zero` — This mode rounds toward zero and is equivalent to the MATLAB `fix` function.

For more information about each of these rounding modes, see "Rounding" on page 3-3.

Overflow Handling

You control how overflow conditions are handled for fixed-point operations with the **Saturate on integer overflow** check box.

If this box is selected, overflows saturate to either the maximum or minimum value represented by the data type. For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

If this box is not selected, overflows wrap to the appropriate value that is representable by the data type. For example, the number 130 does not fit in a signed 8-bit integer, and would wrap to -126.

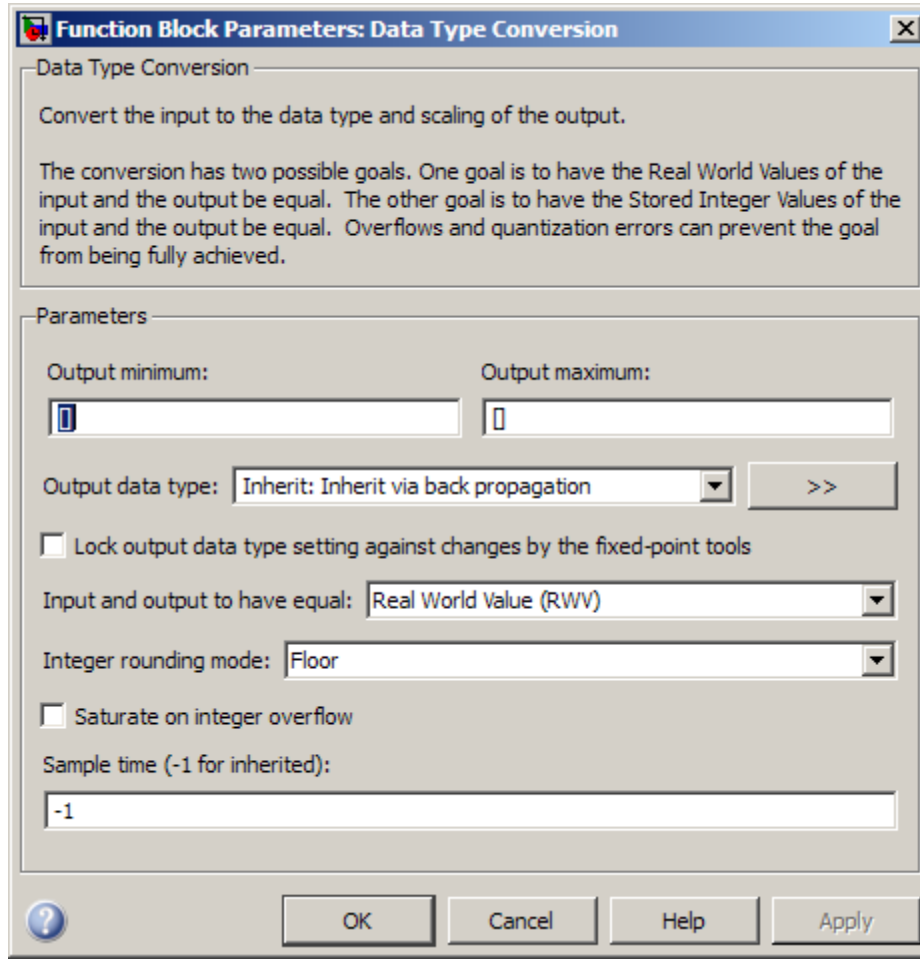
Locking the Output Data Type Setting

If the output data type is a generalized fixed-point number, you have the option of locking its output data type setting by selecting the **Lock output data type setting against changes by the fixed-point tools** check box.

When locked, the Fixed-Point Tool and automatic scaling script `autofixexp` do not change the output data type setting. For more information, see "Automatic Scaling Tools" on page 1-38. Otherwise, the Fixed-Point Tool and `autofixexp` script are free to adjust the output data type setting.

Real-World Values Versus Stored Integer Values

You can configure Data Type Conversion blocks to treat signals as real-world values or as stored integers with the **Input and output to have equal** parameter.



The possible values are `Real World Value (RWV)` and `Stored Integer (SI)`.

In terms of the variables defined in "Scaling" on page 2-5, the real-world value is given by V and the stored integer value is given by Q. You may want to treat numbers as stored integer values if you are modeling hardware that produces integers as output.

Configuring Blocks with Fixed-Point Parameters

Certain Simulink blocks allow you to specify fixed-point numbers as the values of parameters used to compute the block's output, e.g., the **Gain** parameter of a Gain block.

Note S-functions and the Stateflow Chart block do not support fixed-point parameters.

You can specify a fixed-point parameter value either directly by setting the value of the parameter to an expression that evaluates to a `fi` object, or indirectly by setting the value of the parameter to an expression that refers to a fixed-point `Simulink.Parameter` object.

- "Specifying Fixed-Point Values Directly" on page 1-31
- "Specifying Fixed-Point Values Via Parameter Objects" on page 1-32

Note Simulating or performing data type override on a model with `fi` objects requires a Fixed-Point Toolbox software license. See "Sharing Fixed-Point Models" on page 1-4 for more information.

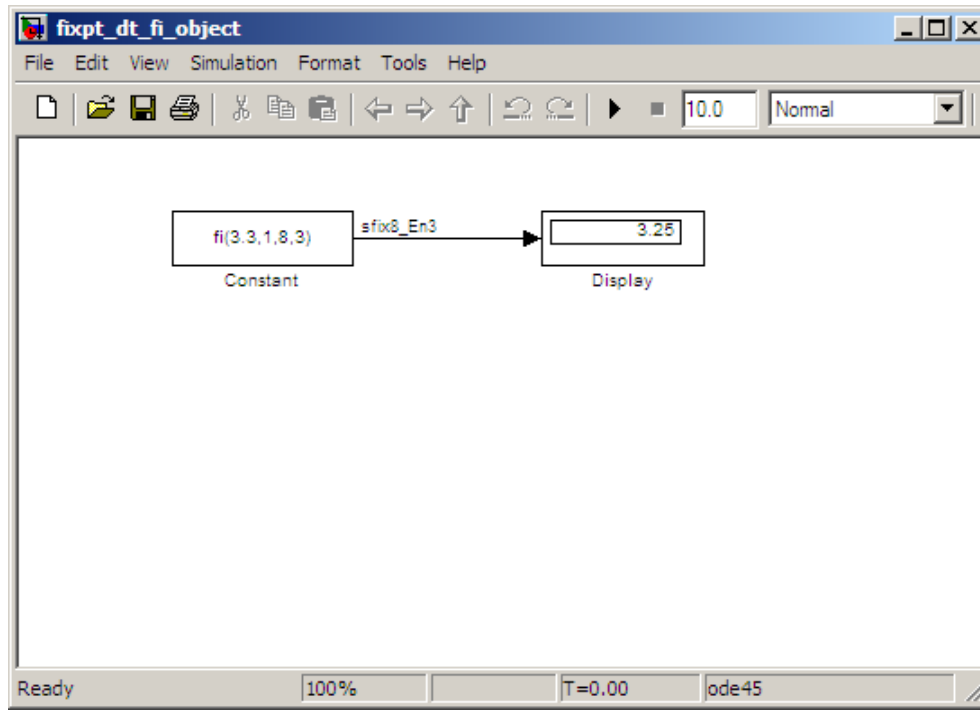
Specifying Fixed-Point Values Directly

You can specify fixed-point values for block parameters using `fi` objects (see "Working with fi Objects" in the *Fixed-Point Toolbox User's Guide* for more information). In the block dialog's parameter field, simply enter the name of a `fi` object or an expression that includes the `fi` constructor function.

For example, entering the expression

```
fi(3.3,1,8,3)
```

as the **Constant value** parameter for the Constant block specifies a signed fixed-point value of 3.3, with a word length of 8 bits and a fraction length of 3 bits.



Specifying Fixed-Point Values Via Parameter Objects

You can specify fixed-point parameter objects for block parameters using instances of the `Simulink.Parameter` class. To create a fixed-point parameter object, either specify a `fi` object as the parameter object's `Value` property, or specify the relevant fixed-point data type for the parameter object's `DataType` property.

For example, suppose you want to create a fixed-point constant in your model. You could do this using a fixed-point parameter object and a Constant block as follows:

1. Enter the following command at the MATLAB prompt to create an instance of the `Simulink.Parameter` class:

   ```
   my_fixpt_param = Simulink.Parameter
   ```

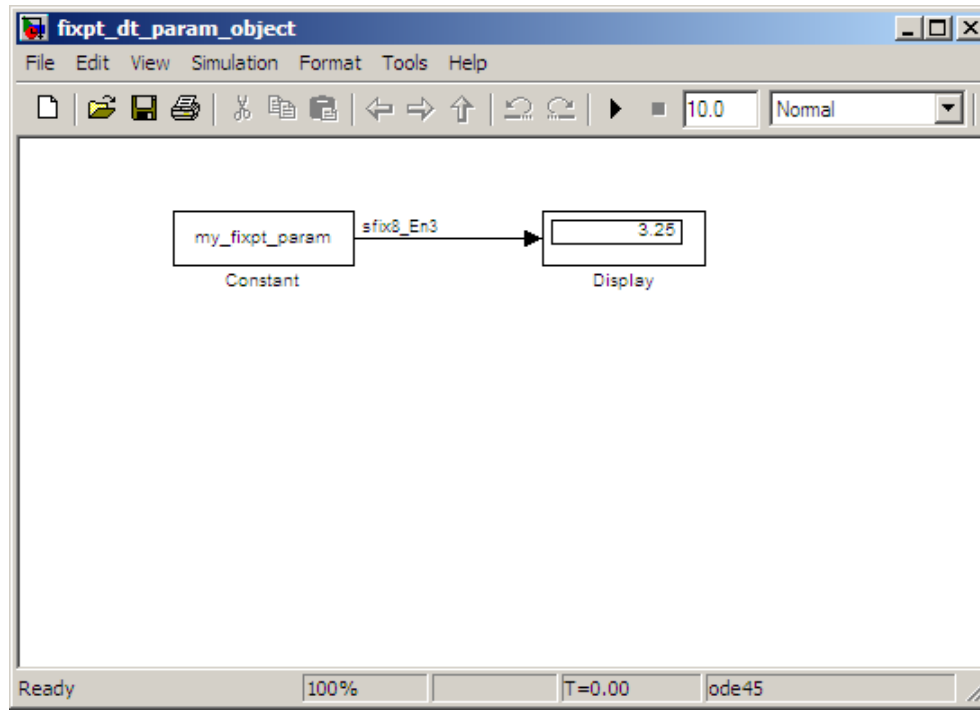
2. Specify either the name of a `fi` object or an expression that includes the `fi` constructor function as the parameter object's `Value` property:

   ```
   my_fixpt_param.Value = fi(3.3,true,8,3)
   ```

   Alternatively, you can set the parameter object's `Value` and `DataType` properties separately. In this case, specify the relevant fixed-point data type using a `Simulink.AliasType` object, a `Simulink.NumericType` object, or a `fixdt` expression. For example, the following commands independently set the parameter object's value and data type, using a `fixdt` expression as the `DataType` string:

   ```
   my_fixpt_param.Value = 3.3;
   my_fixpt_param.DataType = 'fixdt(true,8,2^-3,0)'
   ```

3. Specify the parameter object as the value of a block's parameter. For example, `my_fixpt_param` specifies the **Constant value** parameter for the Constant block in the following model:



Consequently, the Constant block outputs a signed fixed-point value of 3.3, with a word length of 8 bits and a fraction length of 3 bits.

Passing Fixed-Point Data Between Simulink Models and the MATLAB Software

You can read fixed-point data from the MATLAB software into your Simulink models, and there are a number of ways in which you can log fixed-point information from your models and simulations to the workspace.

Reading Fixed-Point Data from the Workspace

You can read fixed-point data from the MATLAB workspace into a Simulink model via the From Workspace block. To do so, the data must be in structure format with a Fixed-Point Toolbox `fi` object in the `values` field. In array format, the From Workspace block only accepts real, double-precision data.

To read in `fi` data, the **Interpolate data** parameter of the From Workspace block must not be selected, and the **Form output after final data value by** parameter must be set to anything other than `Extrapolation`.

Writing Fixed-Point Data to the Workspace

You can write fixed-point output from a model to the MATLAB workspace via the To Workspace block in either array or structure format. Fixed-point data written by a To Workspace block to the workspace in structure format can be read back into a Simulink model in structure format by a From Workspace block.

Note To write fixed-point data to the workspace as a `fi` object, select the **Log fixed-point data as a fi object** check box on the To Workspace block dialog. Otherwise, fixed-point data is converted to `double` and written to the workspace as `double`.

For example, you can use the following code to create a structure in the MATLAB workspace with a `fi` object in the `values` field. You can then use the From Workspace block to bring the data into a Simulink model.

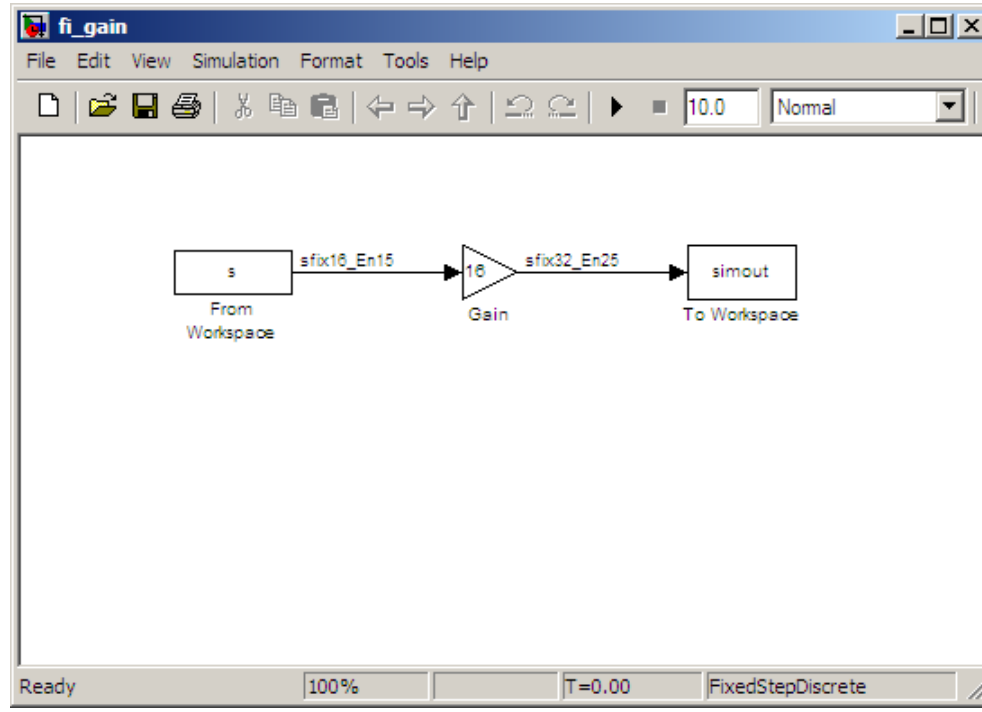
```
a = fi([sin(0:10)' sin(10:-1:0)'])

a =

         0   -0.5440
    0.8415    0.4121
    0.9093    0.9893
    0.1411    0.6570
   -0.7568   -0.2794
   -0.9589   -0.9589
   -0.2794   -0.7568
    0.6570    0.1411
    0.9893    0.9093
    0.4121    0.8415
   -0.5440         0

          DataTypeMode: Fixed-point: binary point scaling
```

```
                 Signed: true
             WordLength: 16
        FractionLength: 15

             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true

s.signals.values = a

s =

    signals: [1x1 struct]

s.signals.dimensions = 2

s =

    signals: [1x1 struct]

s.time = [0:10]'

s =

    signals: [1x1 struct]
       time: [11x1 double]
```

The From Workspace block in the following model has the `fi` structure `s` in the **Data** parameter. In the model, the following parameters in the **Solver** pane of the Configuration Parameters dialog box have the indicated settings:

- **Start time** — `0.0`
- **Stop time** — `10.0`
- **Type** — `Fixed-step`
- **Solver** — `Discrete (no continuous states)`

- **Fixed-step size (fundamental sample time)** — `1.0`



The To Workspace block writes the result of the simulation to the MATLAB workspace as a `fi` structure.

```
simout.signals.values

ans =

         0   -8.7041
   13.4634    6.5938
   14.5488   15.8296
    2.2578   10.5117
  -12.1089   -4.4707
  -15.3428  -15.3428
   -4.4707  -12.1089
   10.5117    2.2578
```

```
   15.8296   14.5488
    6.5938   13.4634
   -8.7041         0
```

Logging Fixed-Point Signals

When fixed-point signals are logged to the MATLAB workspace via signal logging, they are always logged as Fixed-Point Toolbox `fi` objects. To enable signal logging for a signal, select the **Log signal data** option in the signal's Signal Properties dialog box. For more information, refer to "Signal Logging" in *Simulink User's Guide*.

When you log signals from a referenced model or Stateflow chart in your model, the word lengths of `fi` objects may be larger than you expect. The word lengths of fixed-point signals in referenced models and Stateflow charts are logged as the next larger data storage container size.

Accessing Fixed-Point Block Data During Simulation

Simulink provides an application programming interface (API) that enables programmatic access to block data, such as block inputs and outputs, parameters, states, and work vectors, while a simulation is running. You can use this interface to develop MATLAB programs capable of accessing block data while a simulation is running or to access the data from the MATLAB command line. Fixed-point signal information is returned to you via this API as `fi` objects. For more information about the API, refer to "Accessing Block Data During Simulation" in *Simulink User's Guide*.

Automatic Scaling Tools

In addition to the features described in the previous sections, the Simulink Fixed Point software provides you with two automatic scaling (autoscaling) tools:

- Fixed-Point Advisor
- Fixed-Point Tool

Fixed-Point Advisor

The Fixed-Point Advisor provides a set of tasks to facilitate converting a floating-point model or subsystem to an equivalent fixed-point representation.

Note After conversion, use the Fixed-Point Tool to refine the model scaling.

For more information, see "Fixed-Point Advisor" on page 12-2.

To learn how to use the Fixed-Point Advisor, see "Working with the Fixed-Point Advisor" on page 5-2.

Fixed-Point Tool

The Fixed-Point Tool provides a graphical user interface that allows you to configure the parameters associated with automatic scaling. The tool collects range data for model objects, either from design minimum and maximum values that objects specify explicitly, or from logged minimum and maximum values that occur during simulation. It uses this information to propose fixed-point scaling that covers the range with maximum precision.

Using the tool, you can view the simulation results and scaling proposals for a model. After reviewing the scaling proposals, you can choose whether or not to apply them to objects in your model.

Note To prepare a model for conversion and obtain an initial scaling, first use the Fixed-Point Advisor.

For more information, see "Overview of the Fixed-Point Tool" on page 6-2.

To learn how to use the Fixed-Point Tool, refer to Chapter 6, "Fixed-Point Tool".

Note You can also use the `autofixexp` script to automatically change the scaling for each Simulink block that has generalized fixed-point output and does not have its scaling locked. The script uses the maximum and minimum values logged during the last simulation run. The scaling is changed such that the simulation range is covered and the precision is maximized.

Code Generation Capabilities

With the Simulink Coder product, the Simulink Fixed Point software can generate C code. The code generated from fixed-point blocks uses only integer types and automatically includes all operations, such as shifts, needed to account for differences in fixed-point locations.

You can use the generated code on embedded fixed-point processors or rapid prototyping systems even if they contain a floating-point processor. The code is structured so that key operations can be readily replaced by optimized target-specific libraries that you supply. You can also use Target Language Compiler to customize the generated code. Refer to Chapter 11, "Code Generation" for more information about code generation using fixed-point blocks.

Example: Converting from Doubles to Fixed Point

In this section...

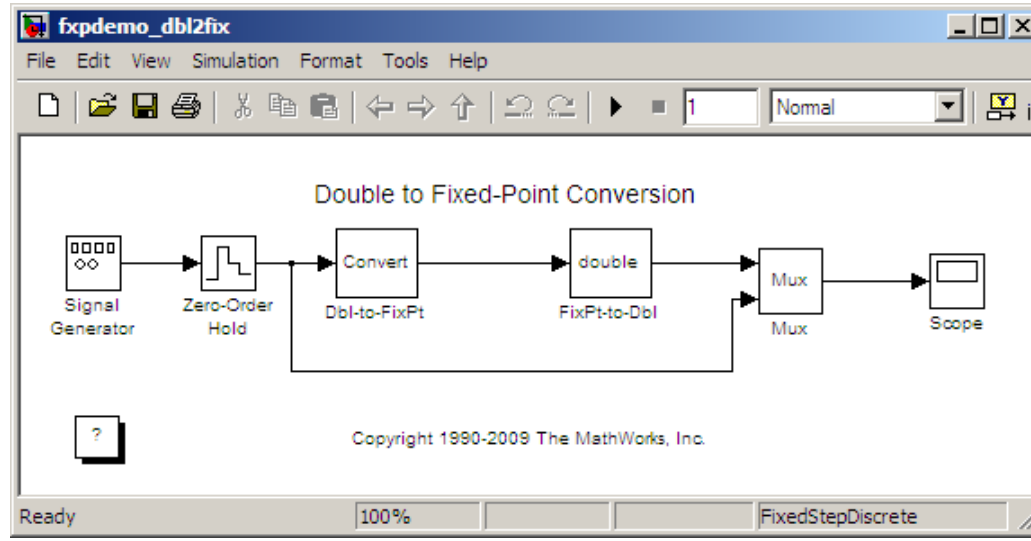
About This Example

The purpose of this example is to show you how to simulate a continuous real-world doubles signal using a generalized fixed-point data type. Although simple in design, the model gives you an opportunity to explore many of the important features of the Simulink Fixed Point software, including

- Data types
- Scaling
- Rounding
- Logging minimum and maximum simulation values to the workspace
- Overflow handling

This example uses the model in the `fxpdemo_dbl2fix` demo. You launch this demo by typing its name at the MATLAB command line:

```
fxpdemo_dbl2fix
```



The sections that follow describe the model and its simulation results.

Block Descriptions

For purposes of this documentation example, you configure the Signal Generator block to output a sine wave signal with an amplitude defined on the interval `[-5 5]`. The Signal Generator block always outputs double-precision numbers.

The Data Type Conversion (Dbl-to-FixPt) block converts the double-precision numbers from the Signal Generator block into one of the Simulink Fixed Point data types. For simplicity, the size of the output signal is 5 bits in this example.

The Data Type Conversion (FixPt-to-Dbl) block converts one of the Simulink Fixed Point data types into a Simulink data type. In this example, it outputs double-precision numbers.

Simulations

The following sections describe how to simulate the model using binary-point-only scaling and [Slope Bias] scaling.

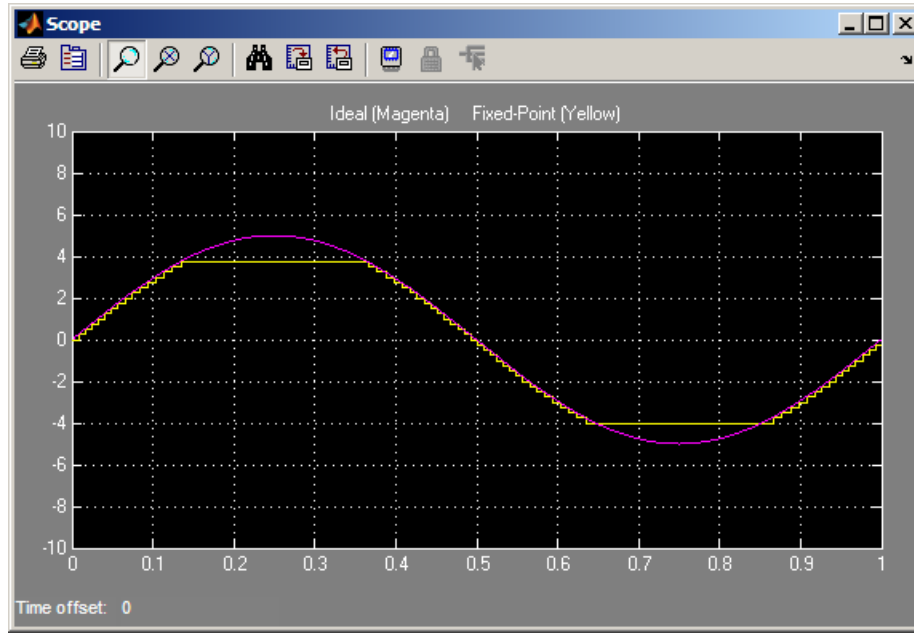
Binary-Point-Only Scaling

When using binary-point-only scaling, your goal is to find the optimal power-of-two exponent *E*, as defined in "Scaling" on page 2-5. For this scaling mode, the fractional slope *F* is 1 and there is no bias.

To run the simulation:

1. Configure the Signal Generator block to output a sine wave signal with an amplitude defined on the interval `[-5 5]`.
   - **a** Double-click the Signal Generator block to open the `Block Parameters` dialog.
   - **b** Set the **Wave form** parameter to `sine`.
   - **c** Set the **Amplitude** parameter to `5`.
   - **d** Click **OK**.
2. Configure the Data Type Conversion (Dbl-to-FixPt) block.
   - **a** Double-click the **Dbl-to-FixPt** block to open the `Block Parameters` dialog.
   - **b** Verify that the **Output data type** parameter is `fixdt(1,5,2)`. `fixdt(1,5,2)` specifies a 5-bit, signed, fixed-point number with scaling `2^-2`, which puts the binary point two places to the left of the rightmost bit. Hence the maximum value is 011.11 = 3.75, a minimum value of 100.00 = -4.00, and the precision is $(1/2)^2 = 0.25$.
   - **c** Verify that the **Integer rounding mode** parameter is `Floor`. `Floor` rounds the fixed-point result toward negative infinity.
   - **d** Select the **Saturate on integer overflow** checkbox to prevent the block from wrapping on overflow.
   - **e** Click **OK**.
3. Select **Simulation > Start** in your Simulink model window.

The Scope displays the real-world and fixed-point simulation results.



The simulation demonstrates the quantization effects of fixed-point arithmetic. Using a 5-bit word with a precision of $(1/2)^2 = 0.25$ produces a discretized output that does not span the full range of the input signal.

If you want to span the complete range of the input signal with 5 bits using binary-point-only scaling, then your only option is to sacrifice precision. Hence, the output scaling is `2^-1`, which puts the binary point one place to the left of the rightmost bit. This scaling gives a maximum value of 0111.1 = 7.5, a minimum value of 1000.0 = -8.0, and a precision of $(1/2)^1 = 0.5$.

To simulate using a precision of 0.5, set the **Output data type** parameter of the Data Type Conversion (Dbl-to-FixPt) block to `fixdt(1,5,1)` and rerun the simulation.

[Slope Bias] Scaling

When using [Slope Bias] scaling, your goal is to find the optimal fractional slope *F* and fixed power-of-two exponent *E*, as defined in "Scaling" on page

2-5. There is no bias for this example because the sine wave is on the interval `[-5 5]`.

To arrive at a value for the slope, you begin by assuming a fixed power-of-two exponent of -2. To find the fractional slope, you divide the maximum value of the sine wave by the maximum value of the scaled 5-bit number. The result is 5.00/3.75 = 1.3333. The slope (and precision) is 1.3333.(0.25) = 0.3333. You specify the [Slope Bias] scaling as [0.3333 0] by entering the expression `fixdt(1,5,0.3333,0)` as the value of the **Output data type** parameter.

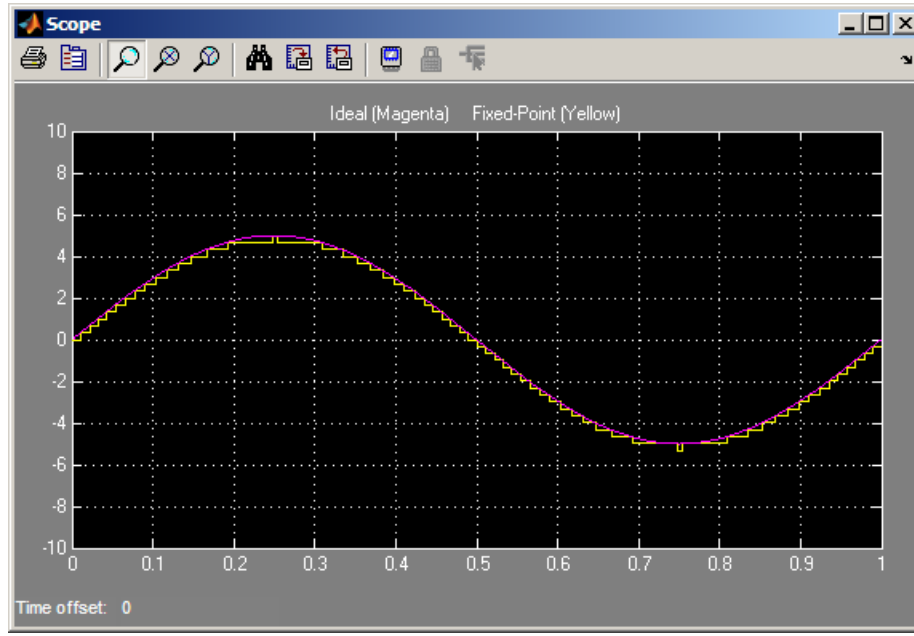
You could also specify a fixed power-of-two exponent of -1 and a corresponding fractional slope of 0.6667. The resulting slope is the same since E is reduced by 1 bit but F is increased by 1 bit. The Simulink Fixed Point software would automatically store F as 1.3332 and E as -2 because of the normalization condition of $1 \leq F < 2$.

To run the simulation:

1. Configure the Signal Generator block to output a sine wave signal with an amplitude defined on the interval `[-5 5]`.
   - **a** Double-click the Signal Generator block to open the `Block Parameters` dialog.
   - **b** Set the **Wave form** parameter to `sine`.
   - **c** Set the **Amplitude** parameter to `5`.
   - **d** Click **OK**.
2. Configure the Data Type Conversion (Dbl-to-FixPt) block.
   - **a** Double-click the **Dbl-to-FixPt** block to open the `Block Parameters` dialog.
   - **b** Set the **Output data type** parameter to `fixdt(1,5,0.3333,0)` to specify [Slope Bias] scaling as [0.3333 0].
   - **c** Verify that the **Integer rounding mode** parameter is `Floor`. `Floor` rounds the fixed-point result toward negative infinity.
   - **d** Select the **Saturate on integer overflow** checkbox to prevent the block from wrapping on overflow.
   - **e** Click **OK**.

3 Select **Simulation > Start** in your Simulink model window.

The Scope displays the real-world and fixed-point simulation results.



You do not need to find the slope using this method. You need only the range of the data you are simulating and the size of the fixed-point word used in the simulation. You can achieve reasonable simulation results by selecting your scaling based on the formula

$$\frac{(max\_value - min\_value)}{2^{ws} - 1},$$

where

- *max_value* is the maximum value to be simulated.
- *min_value* is the minimum value to be simulated.
- *ws* is the word size in bits.

- 2^{ws} - 1 is the largest value of a word with size *ws*.

For this example, the formula produces a slope of 0.32258.

2

Data Types and Scaling

Overview

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of binary digits (1's and 0's). The way in which hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

Binary numbers are represented as either fixed-point or floating-point data types. A fixed-point data type is characterized by the word size in bits, the binary point, and whether it is signed or unsigned. The binary point is the means by which fixed-point values are scaled. With the Simulink Fixed Point software, fixed-point data types can be integers, fractionals, or generalized fixed-point numbers. The main difference between these data types is their default binary point.

Floating-point data types are characterized by a sign bit, a fraction (or mantissa) field, and an exponent field. The blockset adheres to the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic (referred to simply as the IEEE Standard 754 throughout this guide) and supports singles, doubles, and a nonstandard IEEE-style floating-point data type.

When choosing a data type, you must consider these factors:

- The numerical range of the result
- The precision required of the result
- The associated quantization error (i.e., the rounding mode)
- The method for dealing with exceptional arithmetic conditions

These choices depend on your specific application, the computer architecture used, and the cost of development, among others.

With the Simulink Fixed Point software, you can explore the relationship between data types, range, precision, and quantization error in the modeling of dynamic digital systems. With the Simulink Coder product, you can generate production code based on that model.

Fixed-Point Numbers

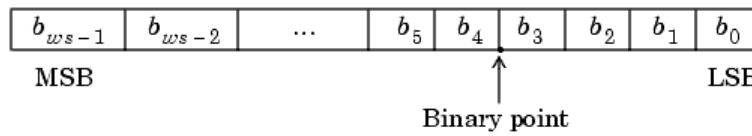
In this section...

About Fixed-Point Numbers

Fixed-point numbers and their data types are characterized by their word size in bits, binary point, and whether they are signed or unsigned. The Simulink Fixed Point software supports integers, fixed-point numbers. The main difference among these data types is their binary point.

Note Fixed-point numbers can have a word size up to 128 bits.

A common representation of a binary fixed-point number , either signed or unsigned, is shown in the following figure.



where

- b_i are the binary digits (bits)
- ws is the word length in bits
- The most significant bit (MSB) is the leftmost bit, and is represented by location b_{ws-1}
- The least significant bit (LSB) is the rightmost bit, and is represented by location b_0
- The binary point is shown four places to the left of the LSB

Signed Fixed-Point Numbers

Computer hardware typically represents the negation of a binary fixed-point number in three different ways: sign/magnitude, one's complement, and two's complement. Two's complement is the preferred representation of signed fixed-point numbers and supported by the Simulink Fixed Point software.

Negation using two's complement consists of a bit inversion (translation into one's complement) followed by the addition of a one. For example, the two's complement of 000101 is 111011.

Whether a fixed-point value is signed or unsigned is usually not encoded explicitly within the binary word; that is, there is no sign bit. Instead, the sign information is implicitly defined within the computer architecture.

Binary Point Interpretation

The binary point is the means by which fixed-point numbers are scaled. It is usually the software that determines the binary point. When performing basic math functions such as addition or subtraction, the hardware uses the same logic circuits regardless of the value of the scale factor. In essence, the logic circuits have no knowledge of a scale factor. They are performing signed or unsigned fixed-point binary algebra as if the binary point is to the right of b_0.

Scaling

The dynamic range of fixed-point numbers is much less than floating-point numbers with equivalent word sizes. To avoid overflow conditions and minimize quantization errors, fixed-point numbers must be scaled.

With the Simulink Fixed Point software, you can select a fixed-point data type whose scaling is defined by its binary point, or you can select an arbitrary linear scaling that suits your needs. This section presents the scaling choices available for fixed-point data types.

You can represent a fixed-point number by a general slope and bias encoding scheme

$$V \approx \tilde{V} = SQ + B,$$

where

- V is an arbitrarily precise real-world value.
- $\tilde{V}$ is the approximate real-world value.
- Q, the stored value, is an integer that encodes V.
- $S = F\,2^E$ is the slope.
- B is the bias.

The slope is partitioned into two components:

- 2^E specifies the binary point. E is the fixed power-of-two exponent.
- F is the slope adjustment factor. It is normalized such that $1 \le F < 2$.

Note S and B are constants and do not show up in the computer hardware directly. Only the quantization value Q is stored in computer memory.

The scaling modes available to you within this encoding scheme are described in the sections that follow. For detailed information about how the supported

scaling modes effect fixed-point operations, refer to "Recommendations for Arithmetic and Scaling" on page 3-34.

Binary-Point-Only Scaling

Binary-point-only or power-of-two scaling involves moving the binary point within the fixed-point word. The advantage of this scaling mode is to minimize the number of processor arithmetic operations.

With binary-point-only scaling, the components of the general slope and bias formula have the following values:

- $F = 1$
- $S = F2^E = 2^E$
- $B = 0$

The scaling of a quantized real-world number is defined by the slope S, which is restricted to a power of two. The negative of the power-of-two exponent is called the fraction length. The fraction length is the number of bits to the right of the binary point. For Binary-Point-Only scaling, specify fixed-point data types as

- signed types — `fixdt(1, WordLength, FractionLength)`
- unsigned types — `fixdt(0, WordLength, FractionLength)`

Integers are a special case of fixed-point data types. Integers have a trivial scaling with slope 1 and bias 0, or equivalently with fraction length 0. Specify integers as

- signed integer — `fixdt(1, WordLength, 0)`
- unsigned integer — `fixdt(0, WordLength, 0)`

Slope and Bias Scaling

When you scale by slope and bias, the slope S and bias B of the quantized real-world number can take on any value. The slope must be a positive number. Using slope and bias, specify fixed-point data types as

- `fixdt(Signed, WordLength, Slope, Bias)`

Unspecified Scaling

Specify fixed-point data types with an unspecified scaling as

- `fixdt(Signed, WordLength)`

Simulink signals, parameters, and states must never have unspecified scaling. When scaling is unspecified, you must use some other mechanism such as automatic best precision scaling to determine the scaling that the Simulink software uses.

Quantization

The quantization Q of a real-world value V is represented by a weighted sum of bits. Within the context of the general slope and bias encoding scheme, the value of an unsigned fixed-point quantity is given by

$$\tilde{V} = S.\left[\sum_{i=0}^{ws-1} b_i 2^i\right] + B,$$

while the value of a signed fixed-point quantity is given by

$$\tilde{V} = S.\left[-b_{ws-1}2^{ws-1} + \sum_{i=0}^{ws-2} b_i 2^i\right] + B,$$

where

- b_i are binary digits, with $b_i = 1,0$, for $i = 0,1,\ldots,ws-1$.
- The word size in bits is given by ws, with $ws = 1, 2, 3,\ldots, 128$.
- S is given by $F2^E$, where the scaling is unrestricted because the binary point does not have to be contiguous with the word.

b_i are called *bit multipliers* and 2^i are called the *weights*.

Example: Fixed-Point Format

Formats for 8-bit signed and unsigned fixed-point values are shown in the following figure.

0	0	1	1	0	1	0	1	Unsigned data type
1	0	1	1	0	1	0	1	Signed data type

Note that you cannot discern whether these numbers are signed or unsigned data types merely by inspection since this information is not explicitly encoded within the word.

The binary number `0011.0101` yields the same value for the unsigned and two's complement representation because the MSB = `0`. Setting B = `0` and using the appropriate weights, bit multipliers, and scaling, the value is

$$\begin{aligned}\tilde{V} &= \left(F2^E\right)Q = 2^E\left[\sum_{i=0}^{ws-1} b_i 2^i\right] \\ &= 2^{-4}\left(0\times 2^7 + 0\times 2^6 + 1\times 2^5 + 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0\right) \\ &= 3.3125.\end{aligned}$$

Conversely, the binary number `1011.0101` yields different values for the unsigned and two's complement representation since the MSB = `1`.

Setting B = `0` and using the appropriate weights, bit multipliers, and scaling, the unsigned value is

$$\begin{aligned}\tilde{V} &= \left(F2^E\right)Q = 2^E\left[\sum_{i=0}^{ws-1} b_i 2^i\right] \\ &= 2^{-4}\left(1\times 2^7 + 0\times 2^6 + 1\times 2^5 + 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0\right) \\ &= 11.3125,\end{aligned}$$

while the two's complement value is

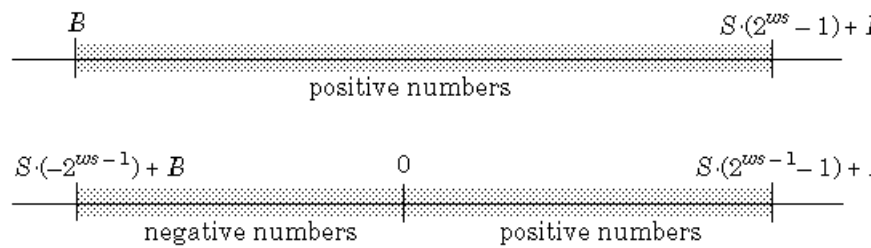
$$\begin{aligned}\tilde{V} &= \left(F2^E\right)Q = 2^E\left[-b_{ws-1}2^{ws-1} + \sum_{i=0}^{ws-2} b_i 2^i\right] \\ &= 2^{-4}\left(-1\times 2^7 + 0\times 2^6 + 1\times 2^5 + 1\times 2^4 + 0\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0\right) \\ &= -4.6875.\end{aligned}$$

Range and Precision

The *range* of a number gives the limits of the representation, while the *precision* gives the distance between successive numbers in the representation. The range and precision of a fixed-point number depends on the length of the word and the scaling.

Range

The range of representable numbers for an unsigned and two's complement fixed-point number of size *ws*, scaling *S*, and bias *B* is illustrated in the following figure.



For both the signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is 2^{ws}.

For example, if the fixed-point data type is an integer with scaling defined as $S = 1$ and $B = 0$, then the maximum unsigned value is 2^{ws-1}, because zero must be represented. In two's complement, negative numbers must be represented as well as zero, so the maximum value is $2^{ws-1}-1$. Additionally, since there is only one representation for zero, there must be an unequal number of positive

and negative numbers. This means there is a representation for -2^{ws-1} but not for 2^{ws-1}.

Precision

The precision of a data type is given by the slope. In this usage, precision means the difference between neighboring representable values.

Fixed-Point Data Type Parameters

The low limit, high limit, and default binary-point-only scaling for the supported fixed-point data types discussed in "Binary Point Interpretation" on page 2-4 are given in the following table. See "Limitations on Precision" on page 3-3 and "Limitations on Range" on page 3-28 for more information.

Fixed-Point Data Type Range and Default Scaling

Name	Data Type	Low Limit	High Limit	Default Scaling (~Precision)
Unsigned Integer	`fixdt(0,ws,0)`	0	$2^{ws} - 1$	1
Signed Integer	`fixdt(1,ws,0)`	-2^{ws-1}	$2^{ws-1} - 1$	1
Unsigned Binary Point	`fixdt(0,ws,fl)`	0	$(2^{ws} - 1)2^{-fl}$	2^{-fl}
Signed Binary Point	`fixdt(1,ws,fl)`	$-2^{ws-1-fl}$	$(2^{ws-1} - 1)2^{-fl}$	2^{-fl}
Unsigned Slope Bias	`fixdt(0,ws,s,b)`	0	$s(2^{ws} - 1) + b$	s
Signed Slope Bias	`fixdt(1,ws,s,b)`	$-s(2^{ws-1}) + b$	$s(2^{ws-1} - 1) + b$	s

s = Slope, *b* = Bias, *ws* = WordLength, *fl* = FractionLength

Range of an 8-Bit Fixed-Point Data Type — Binary-Point-Only Scaling

The precisions, range of signed values, and range of unsigned values for an 8-bit generalized fixed-point data type with binary-point-only scaling are listed in the follow table. Note that the first scaling value (2^1) represents a binary point that is not contiguous with the word.

Scaling	Precision	Range of Signed Values (Low, High)	Range of Unsigned Values (Low, High)
2^1	2.0	-256, 254	0, 510
2^0	1.0	-128, 127	0, 255
2^{-1}	0.5	-64, 63.5	0, 127.5
2^{-2}	0.25	-32, 31.75	0, 63.75
2^{-3}	0.125	-16, 15.875	0, 31.875
2^{-4}	0.0625	-8, 7.9375	0, 15.9375
2^{-5}	0.03125	-4, 3.96875	0, 7.96875
2^{-6}	0.015625	-2, 1.984375	0, 3.984375
2^{-7}	0.0078125	-1, 0.9921875	0, 1.9921875
2^{-8}	0.00390625	-0.5, 0.49609375	0, 0.99609375

Range of an 8-Bit Fixed-Point Data Type — Slope and Bias Scaling

The precision and ranges of signed and unsigned values for an 8-bit fixed-point data type using slope and bias scaling are listed in the following table. The slope starts at a value of `1.25` with a bias of `1.0` for all slopes. Note that the slope is the same as the precision.

Bias	Slope/Precision	Range of Signed Values (low, high)	Range of Unsigned Values (low, high)
1	1.25	-159, 159.75	1, 319.75
1	0.625	-79, 80.375	1, 160.375
1	0.3125	-39, 40.6875	1, 80.6875
1	0.15625	-19, 20.84375	1, 40.84375
1	0.078125	-9, 10.921875	1, 20.921875
1	0.0390625	-4, 5.9609375	1, 10.9609375
1	0.01953125	-1.5, 3.48046875	1, 5.98046875
1	0.009765625	-0.25, 2.240234375	1, 3.490234375
1	0.0048828125	0.375, 1.6201171875	1, 2.2451171875

Constant Scaling for Best Precision

The following fixed-point Simulink blocks provide a mode for scaling parameters whose values are constant vectors or matrices:

- Constant
- Discrete FIR Filter
- Gain
- Relay
- Repeating Sequence Stair

This scaling mode is based on binary-point-only scaling. Using this mode, you can scale a constant vector or matrix such that a common binary point is found based on the best precision for the largest value in the vector or matrix.

Constant scaling for best precision is available only for fixed-point data types with unspecified scaling. All other fixed-point data types use their specified scaling. You can use the **Data Type Assistant** (see "Using the Data Type Assistant" in *Simulink User's Guide*) on a block dialog box to enable the best precision scaling mode.

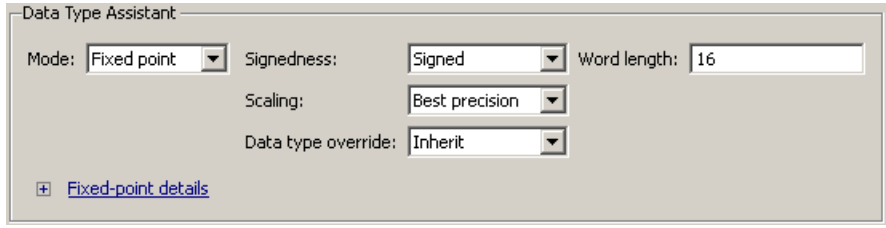
1 On a block dialog box, click the **Show data type assistant** button >> .

   The **Data Type Assistant** appears.

2 In the **Data Type Assistant**, and from the **Mode** list, select `Fixed point`.

   The **Data Type Assistant** displays additional options associated with fixed-point data types.

3 From the **Scaling** list, select `Best precision`.



To understand how you might use this scaling mode, consider a 3-by-3 matrix of doubles, M, defined as

```
3.3333e-003  3.3333e-004  3.3333e-005
3.3333e-002  3.3333e-003  3.3333e-004
3.3333e-001  3.3333e-002  3.3333e-003
```

Now suppose you specify M as the value of the **Gain** parameter for a Gain block. The results of specifying your own scaling versus using the constant scaling mode are described here:

- **Specified Scaling**

  Suppose the matrix elements are converted to a signed, 10-bit generalized fixed-point data type with binary-point-only scaling of 2^{-7} (that is, the binary point is located seven places to the left of the right most bit). With this data format, M becomes

  ```
  0            0            0
  3.1250e-002  0            0
  3.3594e-001  3.1250e-002  0
  ```

  Note that many of the matrix elements are zero, and for the nonzero entries, the scaled values differ from the original values. This is because a double is converted to a binary word of fixed size and limited precision for each element. The larger and more precise the conversion data type, the more closely the scaled values match the original values.

- **Constant Scaling for Best Precision**

  If M is scaled based on its largest matrix value, you obtain

  ```
  2.9297e-003  0            0
  3.3203e-002  2.9297e-003  0
  3.3301e-001  3.3203e-002  2.9297e-003
  ```

  Best precision would automatically select the fraction length that minimizes the quantization error. Even though precision was maximized for the given word length, quantization errors can still occur. In this example, a few elements still quantize to zero.

Fixed-Point Data Type and Scaling Notation

Simulink data type names must be valid MATLAB identifiers with less than 128 characters. The data type name provides information about container type, number encoding, and scaling.

You can represent a fixed-point number using the fixed-point scaling equation

$$V \approx \tilde{V} = SQ + B,$$

where

- V is the real-world value.
- $\tilde{V}$ is the approximate real-world value.
- $S = F2^E$ is the slope.
- F is the slope adjustment factor.
- E is the fixed power-of-two exponent.
- Q is the stored integer.
- B is the bias.

For more information, see "Scaling" on page 2-5.

The following table provides a key for various symbols that appear in Simulink products to indicate the data type and scaling of a fixed-point value.

Symbol	Description	Example
Container Type		
`ufix`	Unsigned fixed-point data type	`ufix8` is an 8-bit unsigned fixed-point data type
`sfix`	Signed fixed-point data type	`sfix128` is a 128-bit signed fixed-point data type

Symbol	Description	Example
`fltu`	Scaled Doubles override of an unsigned fixed-point data type (`ufix`)	`fltu32` is a scaled doubles override of `ufix32`
`flts`	Scaled Doubles override of a signed fixed-point data type (`sfix`)	`flts64` is a scaled doubles override of `sfix64`
Number Encoding		
`e`	10^	`125e8` equals `125*(10^(8))`
`n`	Negative	`n31` equals `-31`
`p`	Decimal point	`1p5` equals `1.5` `p2` equals `0.2`
Scaling Encoding		
`S`	Slope	`ufix16_S5_B7` is a 16-bit unsigned fixed-point data type with `Slope` of `5` and `Bias` of `7`
`B`	Bias	`ufix16_S5_B7` is a 16-bit unsigned fixed-point data type with `Slope` of `5` and `Bias` of `7`
`E`	Fixed exponent (2^) A negative fixed exponent describes the fraction length	`sfix32_En31` is a 32-bit signed fixed-point data type with a fraction length of `31`
`F`	Slope adjustment factor	`ufix16_F1p5_En50` is a 16-bit unsigned fixed-point data type with a `SlopeAdjustmentFactor` of `1.5` and a `FixedExponent` of `-50`

Symbol	Description	Example
C,c,D, or d	Compressed encoding for Bias **Note** If you pass this string to the `slDataTypeAndScale` function, it returns a valid `fixdt` data type.	No example available. For backwards compatibility only. To identify and replace calls to `slDataTypeAndScale`, use the "Check for calls to slDataTypeAndScale" Model Advisor check.
T or t	Compressed encoding for Slope **Note** If you pass this string to the `slDataTypeAndScale`, it returns a valid `fixdt` data type.	No example available. For .backwards compatibility only. To identify and replace calls to `slDataTypeAndScale`, use the "Check for calls to slDataTypeAndScale" Model Advisor check.

Scaled Doubles

What are Scaled Doubles?

Scaled doubles are a hybrid between floating-point and fixed-point numbers. The Simulink Fixed Point software stores them as doubles with the scaling, sign, and word length information retained. For example, the storage container for a fixed-point data type `sfix16_En14` is `int16`. The storage container of the equivalent scaled doubles data type, `flts16_En14` is floating-point `double`. For details of the fixed-point scaling notation, see "Fixed-Point Data Type and Scaling Notation" on page 2-15. The Simulink Fixed Point software applies the scaling information to the stored floating-point double to obtain the real-world value. Storing the value in a double almost always eliminates overflow and precision issues.

What is the Difference between Scaled Double and Double Data Types?. The storage container for both the scaled double and double data types is floating-point `double`. Therefore both data type override settings, `Double` and `Scaled double`, provide the range and precision advantages of floating-point doubles. Scaled doubles retain the information about the specified data type and scaling, but doubles do not retain this information.

Consider an example where you are storing `0.75001` degrees Celsius in a data type `sfix16_En13`. For this data type:

- The slope, $S = 2^{-13}$.
- The bias, $B = 0$.

Using the scaling equation $V \approx \tilde{V} = SQ + B$, where V is the real-world value and Q is the stored value.

- $B = 0$.
- $\tilde{V} = SQ = 2^{-13}Q = 0.75001$.
- $Q = \tilde{V} / S = 0.75001/2^{-13} = 6144.08192$.

The data type `sfix16_En13` can only represent integers, so the ideal value of Q is quantized to `6144` causing precision loss.

If you override the data type `sfix16_En13` with `Double`, the data type changes to `Double` and you lose the information about the scaling. The stored-value equals the real-world value `0.75001`.

If you override the data type `sfix16_En13` with `Scaled Double`, the data type changes to `flts16_En13`. The scaling is still given by `_En13` and is identical to that of the original data type. The only difference is the storage container used to hold the stored value which is now `double` so the stored-value is `6144.08192`. This example demonstrates one advantage of using scaled doubles: the virtual elimination of quantization errors.

When to Use Scaled Doubles

The Fixed-Point Tool enables you to perform various data type overrides on fixed-point signals in your simulations. Use scaled doubles to override the fixed-point data types and scaling using double-precision numbers to avoid quantization effects. Overriding the fixed-point data types provides a floating-point benchmark that represents the ideal output.

Scaled doubles are useful for:

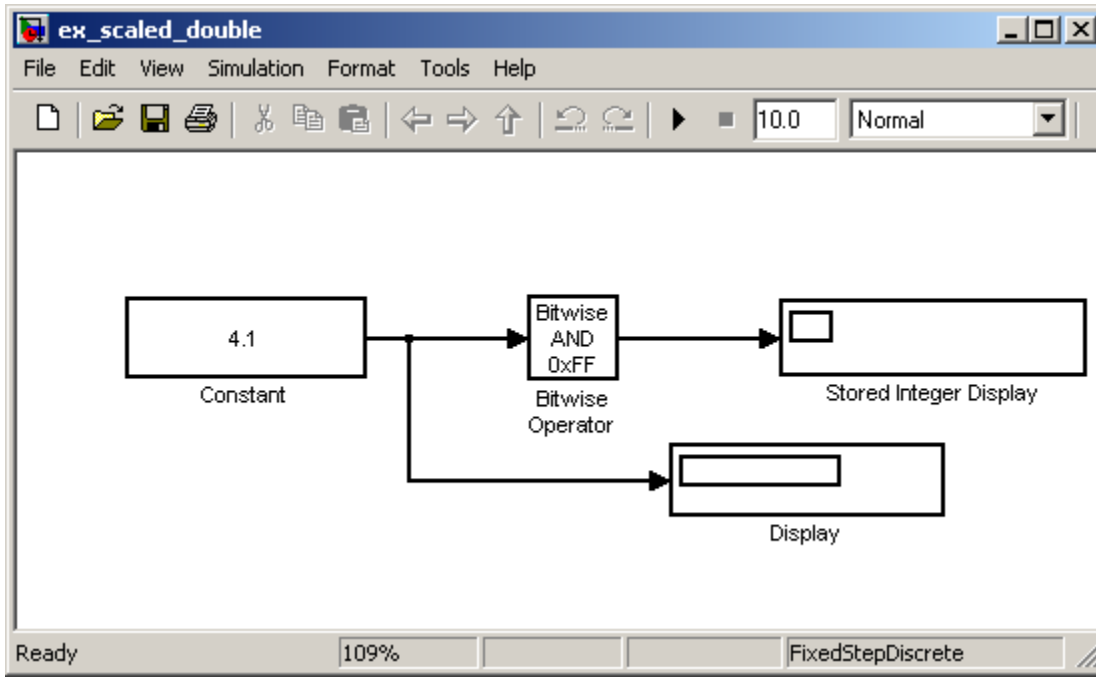
- Testing and debugging
- Applying data type overrides to individual subsystems

  If you apply a data type override to subsystems in your model rather than to the whole model, Scaled doubles provide the information that the fixed-point portions of the model need for consistent data type propagation.

Example: Scaled Doubles

This example uses the `ex_scaled_double` model to show how you can avoid precision loss by overriding the data types in your model with scaled doubles. For more information about scaled doubles, see "Scaled Doubles" on page 2-17.

About the Model



In this model:

- The Constant block output data type is `fixdt(1,8,4)`.
- The Bitwise Operator block uses the `AND` operator and the bit mask `0xFF` to pass the input value to the output. Because the **Treat mask as** parameter is set to `Stored Integer`, the block outputs the stored integer value, *S*, of its input. The encoding scheme is $V=SQ+B$, where V is the real-world value and Q is the stored integer value. For more information, see "Scaling" on page 2-5.

Running the Example

1 Open the `ex_scaled_double` model. At the MATLAB command line, enter:

```
run(docpath(fullfile(docroot,'toolbox','fixpoint','examples','ex_scaled_double.mdl')))
```

2. From the model menu, select **Tools > Fixed-Point > Fixed-Point Tool**.

   The Fixed-Point Tool opens.

3. In the Fixed-Point Tool, set the **Data type override** parameter to `Use local settings` and click **Apply**.

4. From the model menu, select **Simulation > Start**.

   The simulation runs and the `Display` block displays `4.125` as the output value of the Constant block. The `Stored Integer Display` block displays `0100 0010`, which is the binary equivalent of the stored integer value. Precision loss occurs because the output data type, `fixdt(1,8,4)`, cannot represent the output value `4.1` exactly.

5. In the Fixed-Point Tool, set the **Data type override** parameter to `Scaled double` and the **Data type override applies to** parameter to `All numeric types`. Then click **Apply** and rerun the simulation.

   Note You cannot use a **Data type override** setting of `Double` because the Bitwise Operator block does not support floating-point data types.

   The simulation runs and this time the `Display` block correctly displays `4.1` as the output value of the Constant block. The `Stored Integer Display` block displays `65`, which is the binary equivalent of the stored integer value. Because the model uses scaled doubles to override the data type `fixdt(1,8,4)`, the compiled output data type changes to `flts8_En4`, which is the scaled doubles equivalent of `fixdt(1,8,4)`. No precision loss occurs because the scaled doubles retain the information about the specified data type and scaling, and they use a double to hold the stored value.

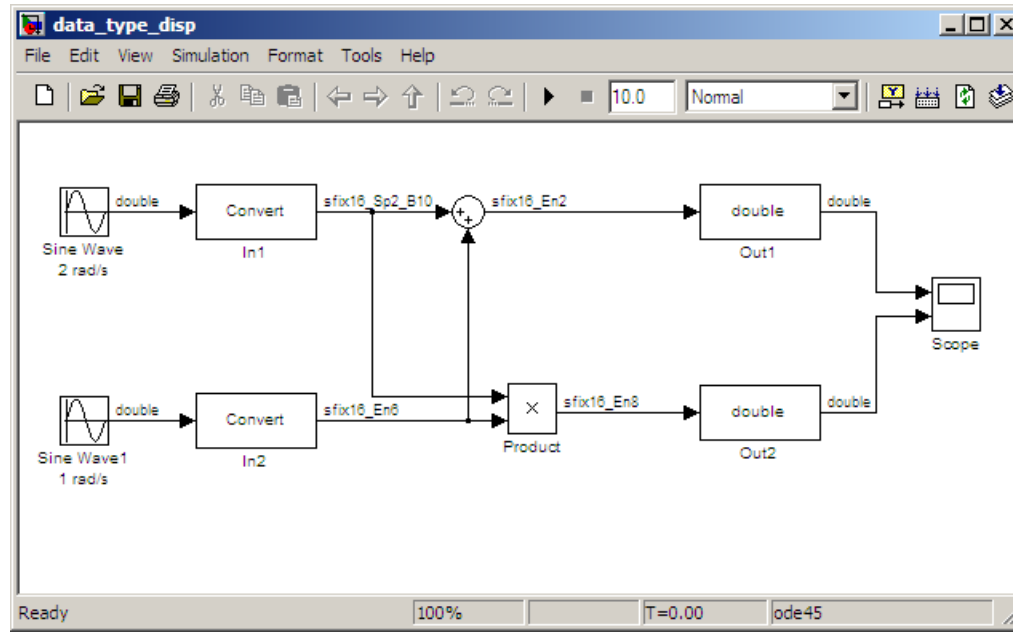
Example: Port Data Type Display

To display the data types for the ports in your model.

1. From the Simulink **Format** menu, point to **Port/Signal Displays**, and then click **Port Data Types** .

The port display for fixed-point signals consists of three parts: the data type, the number of bits, and the scaling. These three parts reflect the block **Output data type** parameter value or the data type and scaling that is inherited from the driving block or through back propagation.

The following model displays its port data types.



In the model, the data type displayed with the `In1` block indicates that the output data type name is `sfix16_Sp2_B10`. This corresponds to `fixdt(1, 16, 0.2, 10)` which is a signed `16` bit fixed-point number with slope `0.2` and bias `10.0`. The data type displayed with the `In2` block indicates that the output data type name is `sfix16_En6`. This corresponds to `fixdt(1, 16, 6)` which is a signed `16` bit fixed-point number with fraction length of `6`.

Floating-Point Numbers

In this section...

About Floating-Point Numbers

Fixed-point numbers are limited in that they cannot simultaneously represent very large or very small numbers using a reasonable word size. This limitation can be overcome by using scientific notation. With scientific notation, you can dynamically place the binary point at a convenient location and use powers of the binary to keep track of that location. Thus, you can represent a range of very large and very small numbers with only a few digits.

You can represent any binary floating-point number in scientific notation form as $f \times 2^e$, where f is the fraction (or mantissa), 2 is the radix or base (binary in this case), and e is the exponent of the radix. The radix is always a positive number, while f and e can be positive or negative.

When performing arithmetic operations, floating-point hardware must take into account that the sign, exponent, and fraction are all encoded within the same binary word. This results in complex logic circuits when compared with the circuits for binary fixed-point operations.

The Simulink Fixed Point software supports single-precision and double-precision floating-point numbers as defined by the IEEE Standard 754. Additionally, a nonstandard IEEE-style number is supported.

Scientific Notation

A direct analogy exists between scientific notation and radix point notation. For example, scientific notation using five decimal digits for the fraction would take the form

$$\pm d.dddd \times 10^{p} = \pm ddddd.0 \times 10^{p-4} = \pm 0.ddddd \times 10^{p+1},$$

where d = 0,...,9 and p is an integer of unrestricted range.

Radix point notation using five bits for the fraction is the same except for the number base

$$\pm b.bbbb \times 2^{q} = \pm bbbbb.0 \times 2^{q-4} = \pm 0.bbbbb \times 2^{q+1},$$

where b = 0,1 and q is an integer of unrestricted range.

For fixed-point numbers, the exponent is fixed but there is no reason why the binary point must be contiguous with the fraction. For example, a word consisting of three unsigned bits is usually represented in scientific notation in one of these four ways.

$$bbb. = bbb. \times 2^{0}$$
$$bb.b = bbb. \times 2^{-1}$$
$$b.bb = bbb. \times 2^{-2}$$
$$.bbb = bbb. \times 2^{-3}$$

If the exponent were greater than 0 or less than -3, then the representation would involve lots of zeros.

$$bbb00000. = bbb. \times 2^{5}$$
$$bbb00. = bbb. \times 2^{2}$$
$$.00bbb = bbb. \times 2^{-5}$$
$$.00000bbb = bbb. \times 2^{-8}$$

These extra zeros never change to ones, however, so they don't show up in the hardware. Furthermore, unlike floating-point exponents, a fixed-point exponent never shows up in the hardware, so fixed-point exponents are not limited by a finite number of bits.

Note Restricting the binary point to being contiguous with the fraction is unnecessary; the Simulink Fixed Point software allows you to extend the binary point to any arbitrary location.

The IEEE Format

The IEEE Standard 754 has been widely adopted, and is used with virtually all floating-point processors and arithmetic coprocessors—with the notable exception of many DSP floating-point processors.

Among other things, this standard specifies four floating-point number formats, of which singles and doubles are the most widely used. Each format contains three components: a sign bit, a fraction field, and an exponent field. These components, as well as the specific formats for singles and doubles, are discussed in the sections that follow.

The Sign Bit

While two's complement is the preferred representation for signed fixed-point numbers, IEEE floating-point numbers use a sign/magnitude representation, where the sign bit is explicitly included in the word. Using this representation, a sign bit of 0 represents a positive number and a sign bit of 1 represents a negative number.

The Fraction Field

In general, floating-point numbers can be represented in many different ways by shifting the number to the left or right of the binary point and decreasing or increasing the exponent of the binary by a corresponding amount.

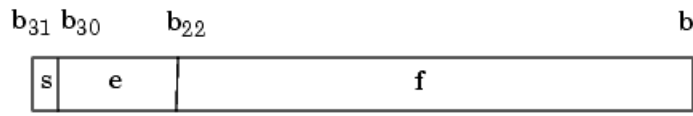
To simplify operations on these numbers, they are *normalized* in the IEEE format. A normalized binary number has a fraction of the form 1.*f* where *f* has a fixed size for a given data type. Since the leftmost fraction bit is always a 1, it is unnecessary to store this bit and is therefore implicit (or hidden). Thus, an n-bit fraction stores an n+1-bit number. The IEEE format also supports denormalized numbers, which have a fraction of the form 0.*f*. Normalized and denormalized formats are discussed in more detail in the next section.

The Exponent Field

In the IEEE format, exponent representations are biased. This means a fixed value (the bias) is subtracted from the field to get the true exponent value. For example, if the exponent field is 8 bits, then the numbers 0 through 255 are represented, and there is a bias of 127. Note that some values of the exponent are reserved for flagging `Inf` (infinity), `NaN` (not-a-number), and denormalized numbers, so the true exponent values range from -126 to 127. See the sections "Inf" on page 2-30 and "NaN" on page 2-30.

Single-Precision Format

The IEEE single-precision floating-point format is a 32-bit word divided into a 1-bit sign indicator *s*, an 8-bit biased exponent *e*, and a 23-bit fraction *f*. For more information, see "The Sign Bit" on page 2-25, "The Exponent Field" on page 2-26, and "The Fraction Field" on page 2-25. A representation of this format is given below.



The relationship between this format and the representation of real numbers is given by

$$value = \begin{cases} (-1)^s (2^{e-127})(1.f) & \text{normalized, } 0 < e < 255, \\ (-1)^s (2^{e-126})(0.f) & \text{denormalized, } e = 0,\ f > 0, \\ \text{exceptional value} & \text{otherwise.} \end{cases}$$

"Exceptional Arithmetic" on page 2-29 discusses denormalized values.

Double-Precision Format

The IEEE double-precision floating-point format is a 64-bit word divided into a 1-bit sign indicator *s*, an 11-bit biased exponent *e*, and a 52-bit fraction *f*.For more information, see "The Sign Bit" on page 2-25, "The Exponent Field" on

page 2-26, and "The Fraction Field" on page 2-25. A representation of this format is shown in the following figure.

The relationship between this format and the representation of real numbers is given by

$$value = \begin{cases} (-1)^s (2^{e-1023})(1.f) & \text{normalized, } 0 < e < 2047, \\ (-1)^s (2^{e-1022})(0.f) & \text{denormalized, } e = 0,\ f > 0, \\ \text{exceptional value} & \text{otherwise.} \end{cases}$$

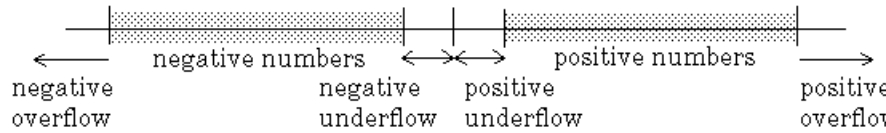
"Exceptional Arithmetic" on page 2-29 discusses denormalized values.

Range and Precision

The range of a number gives the limits of the representation while the precision gives the distance between successive numbers in the representation. The range and precision of an IEEE floating-point number depend on the specific format.

Range

The range of representable numbers for an IEEE floating-point number with f bits allocated for the fraction, e bits allocated for the exponent, and the bias of e given by $bias = 2^{(e-1)}-1$ is given below.



where

- Normalized positive numbers are defined within the range $2^{(1-bias)}$ to $(2-2^{-f})\times 2^{bias}$.
- Normalized negative numbers are defined within the range $-2^{(1-bias)}$ to $-(2-2^{-f})\times 2^{bias}$.
- Positive numbers greater than $(2-2^{-f})\times 2^{bias}$ and negative numbers greater than $-(2-2^{-f})\times 2^{bias}$ are overflows.
- Positive numbers less than $2^{(1-bias)}$ and negative numbers less than $-2^{(1-bias)}$ are either underflows or denormalized numbers.
- Zero is given by a special bit pattern, where $e = 0$ and $f = 0$.

Overflows and underflows result from exceptional arithmetic conditions. Floating-point numbers outside the defined range are always mapped to ±Inf.

Note You can use the MATLAB commands `realmin` and `realmax` to determine the dynamic range of double-precision floating-point values for your computer.

Precision

Because of a finite word size, a floating-point number is only an approximation of the "true" value. Therefore, it is important to have an understanding of the precision (or accuracy) of a floating-point result. In general, a value v with an accuracy q is specified by $v \pm q$. For IEEE floating-point numbers,

$$v = (-1)^{s}(2^{e-bias})(1.f)$$

and

$$q = 2^{-f}\times 2^{e-bias}$$

Thus, the precision is associated with the number of bits in the fraction field.

Note In the MATLAB software, floating-point relative accuracy is given by the command `eps`, which returns the distance from 1.0 to the next larger floating-point number. For a computer that supports the IEEE Standard 754, `eps` = 2^{-52} or $2.22045 \cdot 10^{-16}$.

Floating-Point Data Type Parameters

The high and low limits, exponent bias, and precision for the supported floating-point data types are given in the following table.

Data Type	Low Limit	High Limit	Exponent Bias	Precision
Single	$2^{-126} \approx 10^{-38}$	$2^{128} \approx 3 \cdot 10^{38}$	127	$2^{-23} \approx 10^{-7}$
Double	$2^{-1022} \approx 2 \cdot 10^{-308}$	$2^{1024} \approx 2 \cdot 10^{308}$	1023	$2^{-52} \approx 10^{-16}$
Nonstandard	$2^{(1 - bias)}$	$(2 - 2^{-f}) \cdot 2^{bias}$	$2^{(e - 1)} - 1$	2^{-f}

Because of the sign/magnitude representation of floating-point numbers, there are two representations of zero, one positive and one negative. For both representations $e = 0$ and $f.0 = 0.0$.

Exceptional Arithmetic

In addition to specifying a floating-point format, the IEEE Standard 754 specifies practices and procedures so that predictable results are produced independently of the hardware platform. Specifically, denormalized numbers, `Inf`, and `NaN` are defined to deal with exceptional arithmetic (underflow and overflow).

If an underflow or overflow is handled as `Inf` or `NaN`, then significant processor overhead is required to deal with this exception. Although the IEEE Standard 754 specifies practices and procedures to deal with exceptional arithmetic conditions in a consistent manner, microprocessor manufacturers might handle these conditions in ways that depart from the standard. Some of the alternative approaches, such as saturation and wrapping, are discussed in Chapter 3, “Arithmetic Operations”.

Denormalized Numbers

Denormalized numbers are used to handle cases of exponent underflow. When the exponent of the result is too small (i.e., a negative exponent with too large a magnitude), the result is denormalized by right-shifting the fraction and leaving the exponent at its minimum value. The use of denormalized numbers is also referred to as gradual underflow. Without denormalized numbers, the gap between the smallest representable nonzero number and zero is much wider than the gap between the smallest representable nonzero number and the next larger number. Gradual underflow fills that gap and reduces the impact of exponent underflow to a level comparable with roundoff among the normalized numbers. Thus, denormalized numbers provide extended range for small numbers at the expense of precision.

Inf

Arithmetic involving `Inf` (infinity) is treated as the limiting case of real arithmetic, with infinite values defined as those outside the range of representable numbers, or $-\infty \leq$ (representable numbers) $< \infty$. With the exception of the special cases discussed below (`NaN`), any arithmetic operation involving `Inf` yields `Inf`. `Inf` is represented by the largest biased exponent allowed by the format and a fraction of zero.

NaN

A `NaN` (not-a-number) is a symbolic entity encoded in floating-point format. There are two types of `NaN`: signaling and quiet. A signaling `NaN` signals an invalid operation exception. A quiet `NaN` propagates through almost every arithmetic operation without signaling an exception. The following operations result in a `NaN`: $\infty-\infty$, $-\infty+\infty$, $0\times\infty$, 0/0, and ∞/∞.

Both types of `NaN` are represented by the largest biased exponent allowed by the format and a fraction that is nonzero. The bit pattern for a quiet `NaN` is given by 0.*f* where the most significant number in *f* must be a one, while the bit pattern for a signaling `NaN` is given by 0.*f* where the most significant number in *f* must be zero and at least one of the remaining numbers must be nonzero.

3

Arithmetic Operations

Overview

When developing a dynamic system using floating-point arithmetic, you generally don't have to worry about numerical limitations since floating-point data types have high precision and range. Conversely, when working with fixed-point arithmetic, you must consider these factors when developing dynamic systems:

- **Overflow**

  Adding two sufficiently large negative or positive values can produce a result that does not fit into the representation. This will have an adverse effect on the control system.

- **Quantization**

  Fixed-point values are rounded. Therefore, the output signal to the plant and the input signal to the control system do not have the same characteristics as the ideal discrete-time signal.

- **Computational noise**

  The accumulated errors that result from the rounding of individual terms within the realization introduce noise into the control signal.

- **Limit cycles**

  In the ideal system, the output of a stable transfer function (digital filter) approaches some constant for a constant input. With quantization, limit cycles occur where the output oscillates between two values in steady state.

This chapter describes the limitations involved when arithmetic operations are performed using encoded fixed-point variables. It also provides recommendations for encoding fixed-point variables such that simulations and generated code are reasonably efficient.

Limitations on Precision

In this section...

Introduction

Computer words consist of a finite numbers of bits. This means that the binary encoding of variables is only an approximation of an arbitrarily precise real-world value. Therefore, the limitations of the binary representation automatically introduce limitations on the precision of the value. For a general discussion of range and precision, refer to "Range and Precision" on page 2-9.

The precision of a fixed-point word depends on the word size and binary point location. Extending the precision of a word can always be accomplished with more bits, but you face practical limitations with this approach. Instead, you must carefully select the data type, word size, and scaling such that numbers are accurately represented. Rounding and padding with trailing zeros are typical methods implemented on processors to deal with the precision of binary words.

Rounding

The result of any operation on a fixed-point number is typically stored in a register that is longer than the number's original format. When the result is put back into the original format, the extra bits must be disposed of. That is, the result must be *rounded*. Rounding involves going from high precision to lower precision and produces quantization errors and computational noise.

How to Choose a Rounding Mode

To choose the most suitable rounding mode for your application, you need to consider your system requirements and the properties of each rounding mode. The most important properties to consider are:

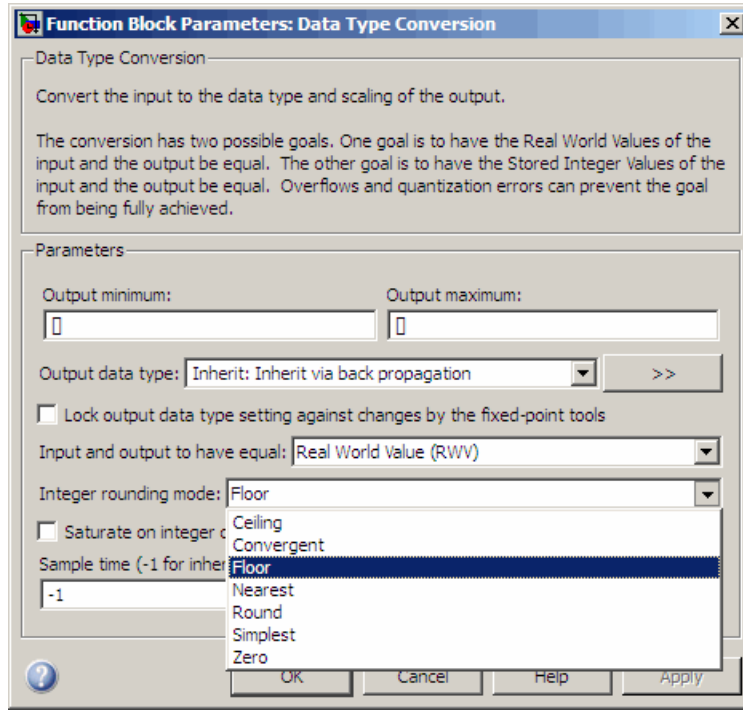
- Cost — Independent of the hardware being used, how much processing expense does the rounding method require?
- Bias — What is the expected value of the rounded values minus the original values?
- Possibility of overflow — Does the rounding method introduce the possibility of overflow?

For more information on when to use each rounding mode, see "Rounding Methods" in the *Fixed-Point Toolbox User's Guide*.

Choosing a Rounding Mode for Diagnostic Purposes. Rounding toward ceiling and rounding toward floor are sometimes useful for diagnostic purposes. For example, after a series of arithmetic operations, you may not know the exact answer because of word-size limitations, which introduce rounding. If every operation in the series is performed twice, once rounding to positive infinity and once rounding to negative infinity, you obtain an upper limit and a lower limit on the correct answer. You can then decide if the result is sufficiently accurate or if additional analysis is necessary.

Rounding Modes for Fixed-Point Simulink Blocks

Fixed-point Simulink blocks support the rounding modes shown in the expanded drop-down menu of the following dialog box.



The following table illustrates the differences between these rounding modes:

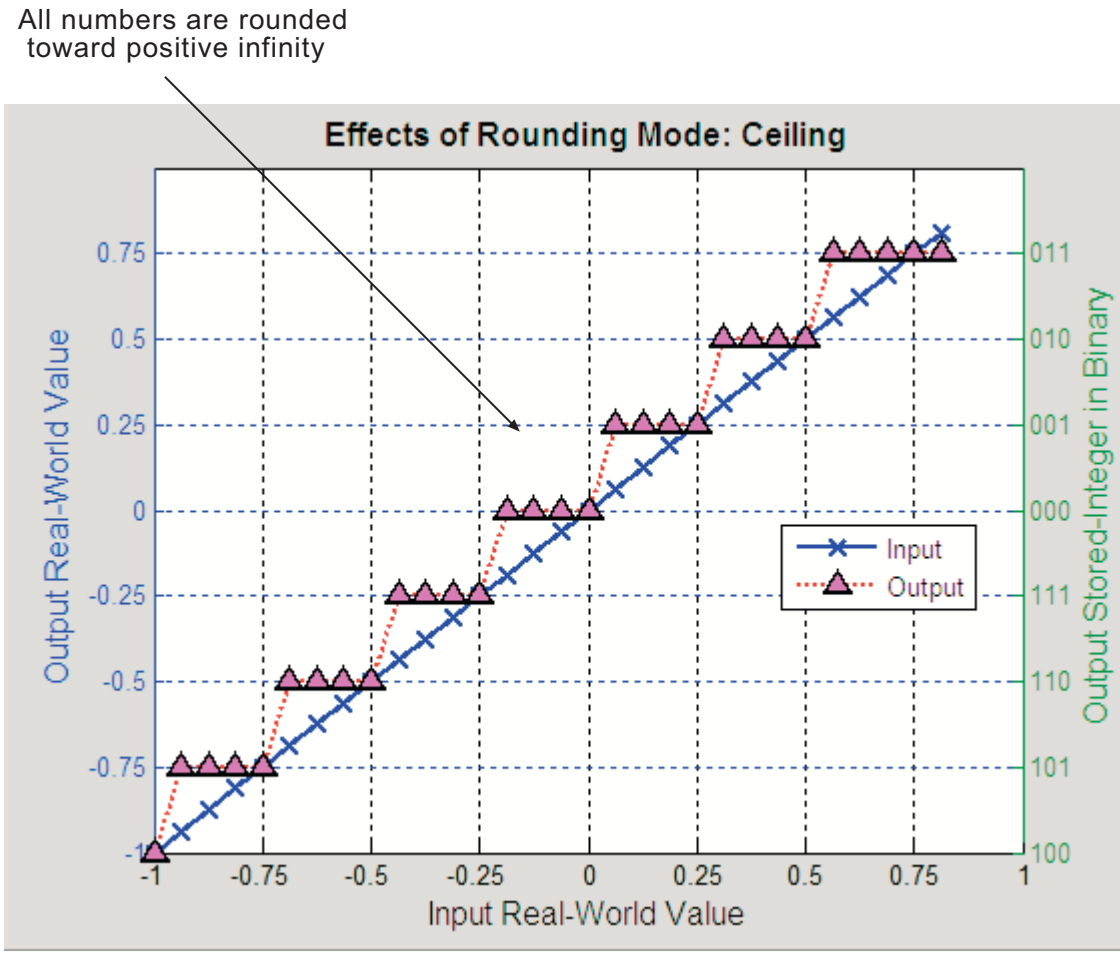
Rounding Mode	Description	Tie Handling
Ceiling	Rounds to the nearest representable number in the direction of positive infinity.	N/A
Floor	Rounds to the nearest representable number in the direction of negative infinity.	N/A
Zero	Rounds to the nearest representable number in the direction of zero.	N/A

Rounding Mode	Description	Tie Handling
Convergent	Rounds to the nearest representable number.	Ties are rounded toward the nearest even integer.
Nearest	Rounds to the nearest representable number.	Ties are rounded to the closest representable number in the direction of positive infinity.
Round	Rounds to the nearest representable number.	For positive numbers, ties are rounded toward the closest representable number in the direction of positive infinity. For negative numbers, ties are rounded toward the closest representable number in the direction of negative infinity.
Simplest	Automatically chooses between `Floor` and `Zero` to produce generated code that is as efficient as possible.	N/A

Rounding Mode: Ceiling

When you round toward ceiling, both positive and negative numbers are rounded toward positive infinity. As a result, a positive cumulative bias is introduced in the number.

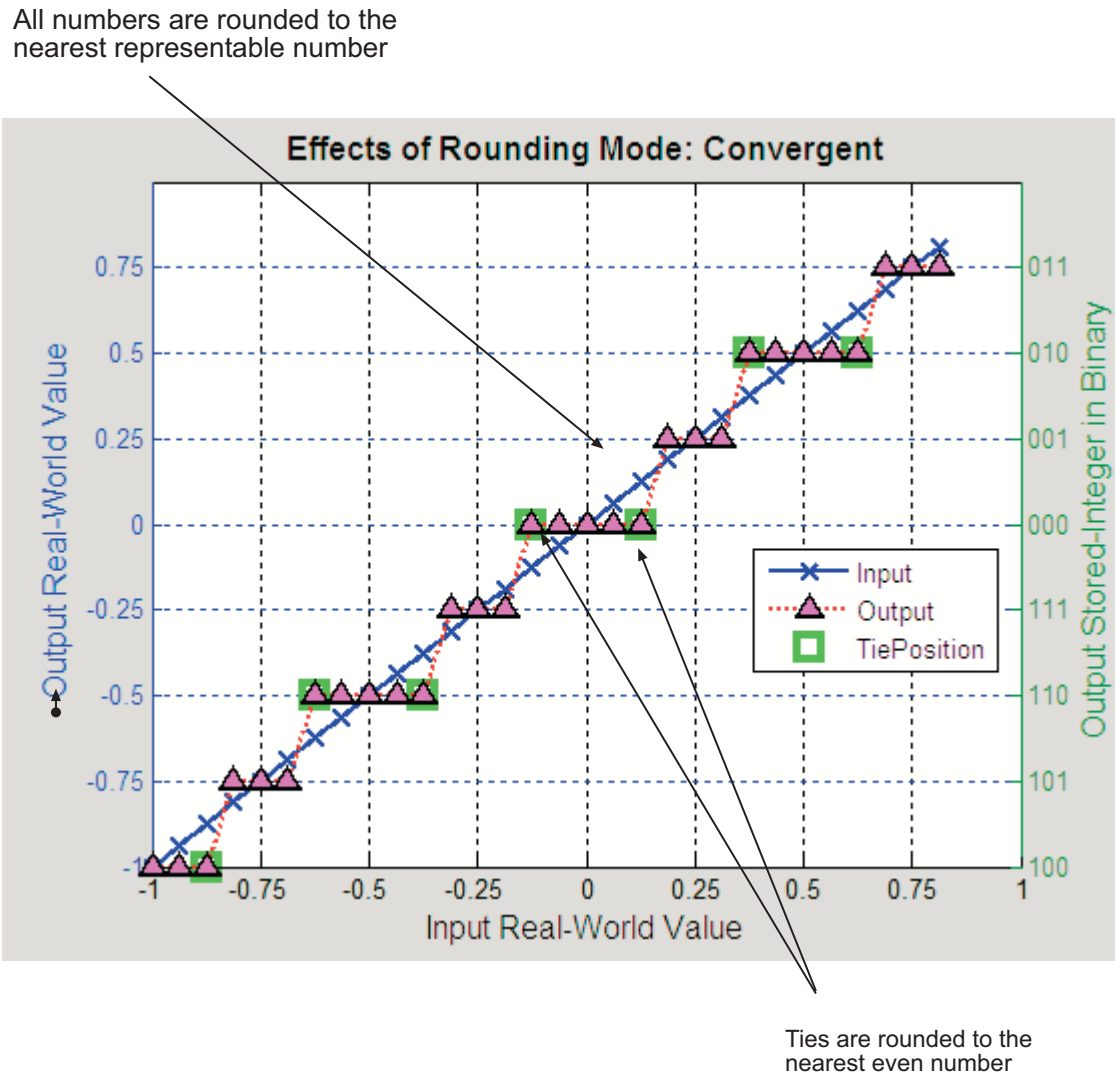
In the MATLAB software, you can round to ceiling using the `ceil` function. Rounding toward ceiling is shown in the following figure.



Rounding Mode: Convergent

`Convergent` rounds toward the nearest representable value with ties rounding toward the nearest even integer. It eliminates bias due to rounding. However, it introduces the possibility of overflow.

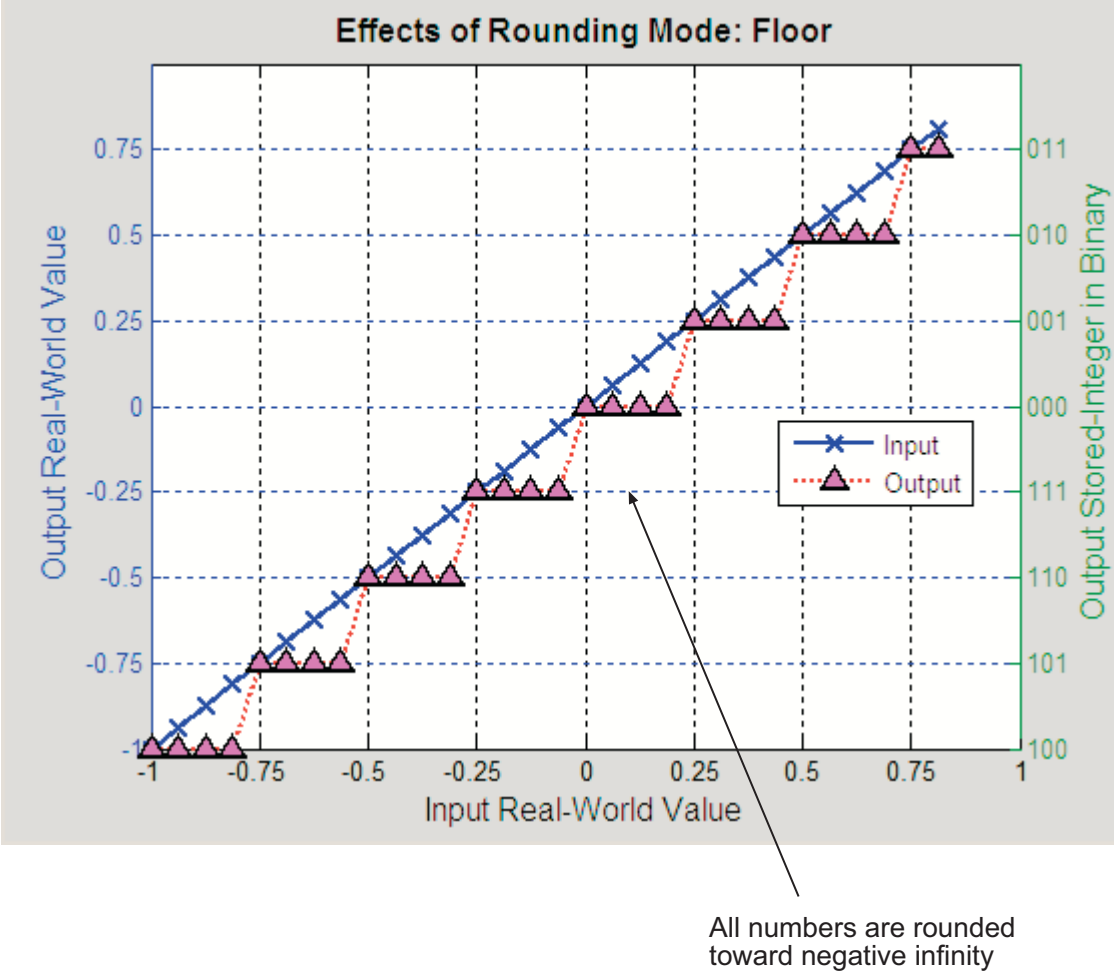
In the MATLAB software, you can perform convergent rounding using the `convergent` function. `Convergent` rounding is shown in the following figure.



Rounding Mode: Floor

When you round toward floor, both positive and negative numbers are rounded to negative infinity. As a result, a negative cumulative bias is introduced in the number.

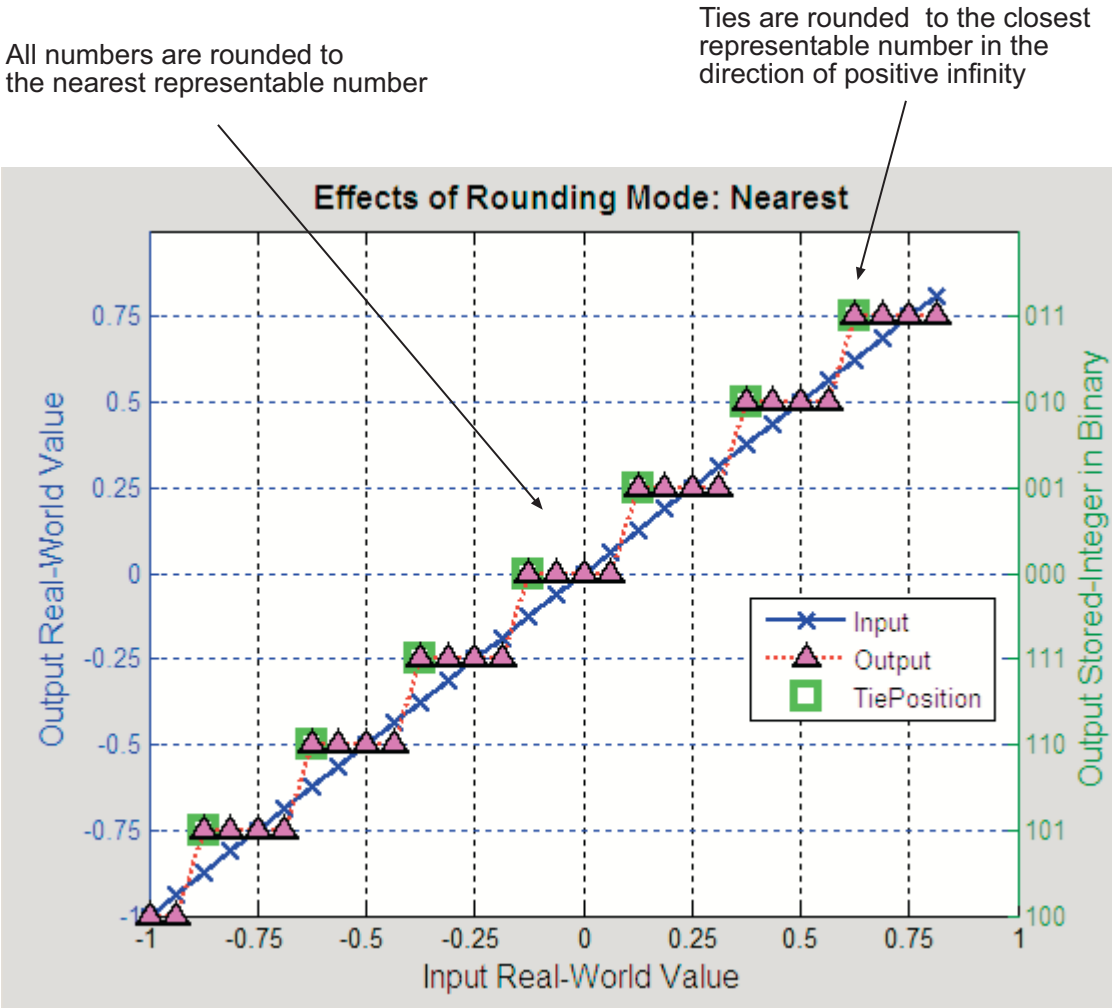
In the MATLAB software, you can round to floor using the `floor` function. Rounding toward floor is shown in the following figure.



Rounding Mode: Nearest

When you round toward nearest, the number is rounded to the nearest representable value. In the case of a tie, `nearest` rounds to the closest representable number in the direction of positive infinity.

In the Fixed-Point Toolbox software, you can round to nearest using the `nearest` function. Rounding toward nearest is shown in the following figure.



Rounding Mode: Round

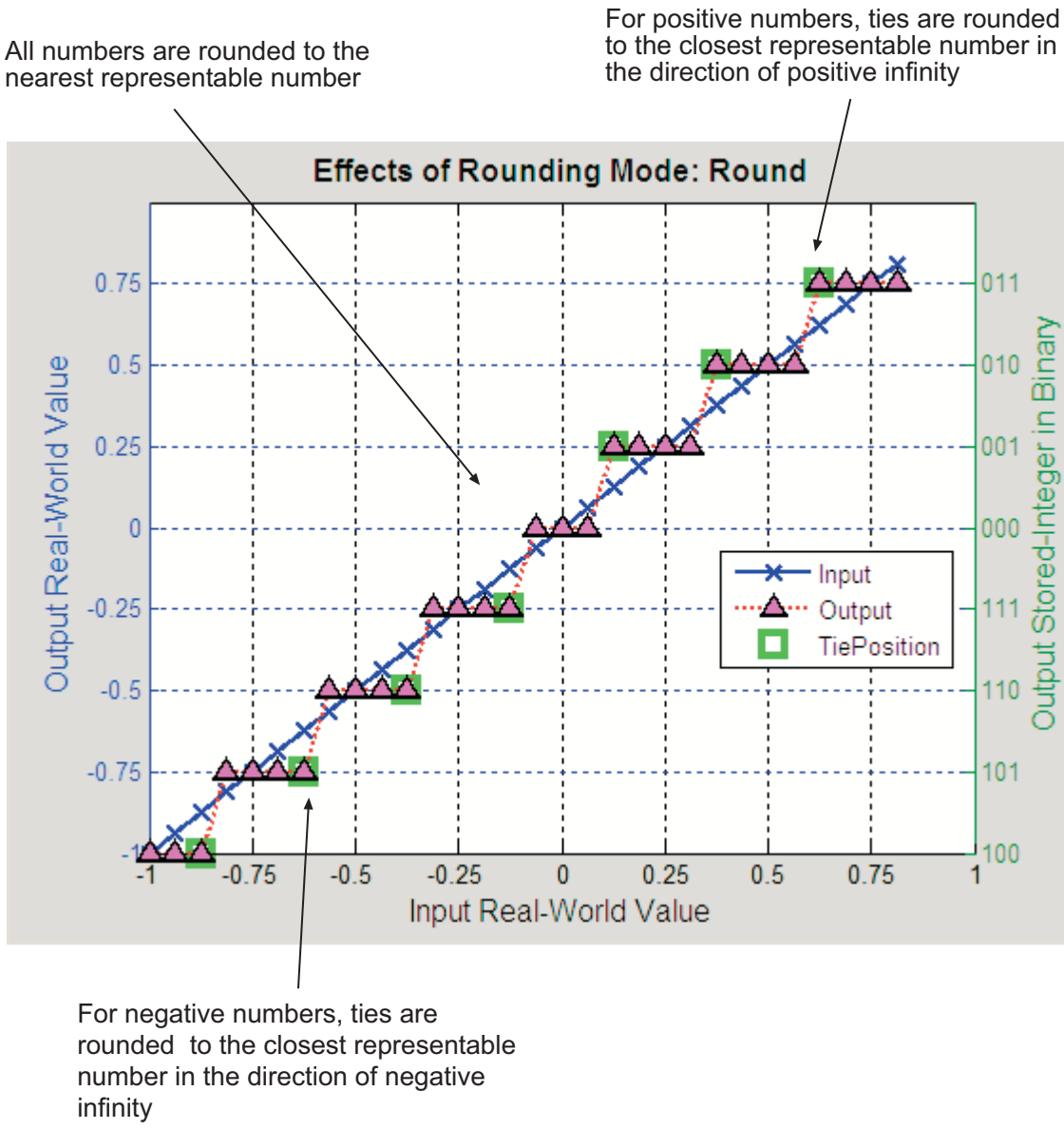
`Round` rounds to the closest representable number. In the case of a tie, it rounds:

- Positive numbers to the closest representable number in the direction of positive infinity.
- Negative numbers to the closest representable number in the direction of negative infinity.

As a result:

- A small negative bias is introduced for negative samples.
- No bias is introduced for samples with evenly distributed positive and negative values.
- A small positive bias is introduced for positive samples.

In the MATLAB software, you can perform this type of rounding using the `round` function. The rounding mode `Round` is shown in the following figure.



Rounding Mode: Simplest

The simplest rounding mode attempts to reduce or eliminate the need for extra rounding code in your generated code using a combination of techniques, discussed in the following sections:

- "Optimize Rounding for Casts" on page 3-13
- "Optimize Rounding for High-Level Arithmetic Operations" on page 3-14
- "Optimize Rounding for Intermediate Arithmetic Operations" on page 3-15

In nearly all cases, the simplest rounding mode produces the most efficient generated code. For a very specialized case of division that meets three specific criteria, round to floor might be more efficient. These three criteria are:

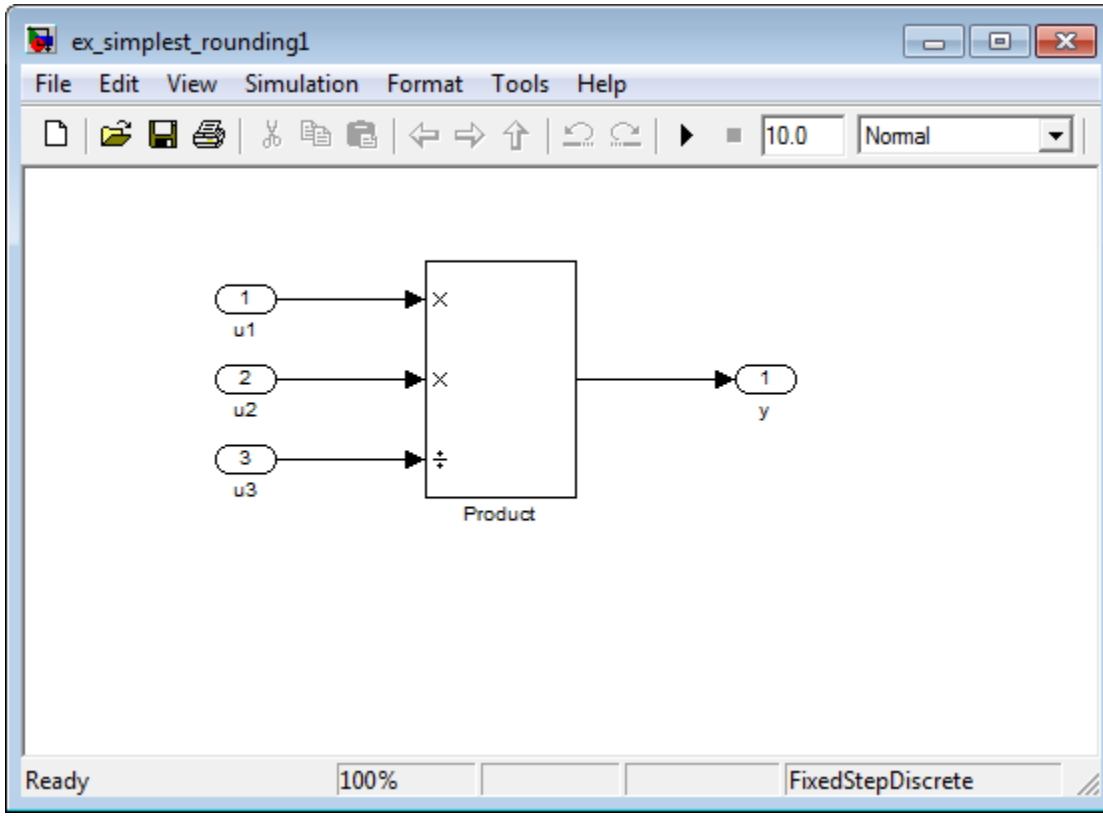
- Fixed-point/integer signed division
- Denominator is an invariant constant
- Denominator is an exact power of two

For this case, set the rounding mode to floor and the **Configuration Parameters > Hardware Implementation > Embedded Hardware > Signed integer division rounds to** parameter to describe the rounding behavior of your production target.

Optimize Rounding for Casts. The Data Type Conversion block casts a signal with one data type to another data type. When the block casts the signal to a data type with a shorter word length than the original data type, precision is lost and rounding occurs. The simplest rounding mode automatically chooses the best rounding for these cases based on the following rules:

- When casting from one integer or fixed-point data type to another, the simplest mode rounds toward floor.
- When casting from a floating-point data type to an integer or fixed-point data type, the simplest mode rounds toward zero.

Optimize Rounding for High-Level Arithmetic Operations. The simplest rounding mode chooses the best rounding for each high-level arithmetic operation. For example, consider the operation $y = u_1 \times u_2 / u_3$ implemented using a Product block:



As stated in the C standard, the most efficient rounding mode for multiplication operations is always floor. However, the C standard does not specify the rounding mode for division in cases where at least one of the operands is negative. Therefore, the most efficient rounding mode for a divide operation with signed data types can be floor or zero, depending on your production target.

The simplest rounding mode:

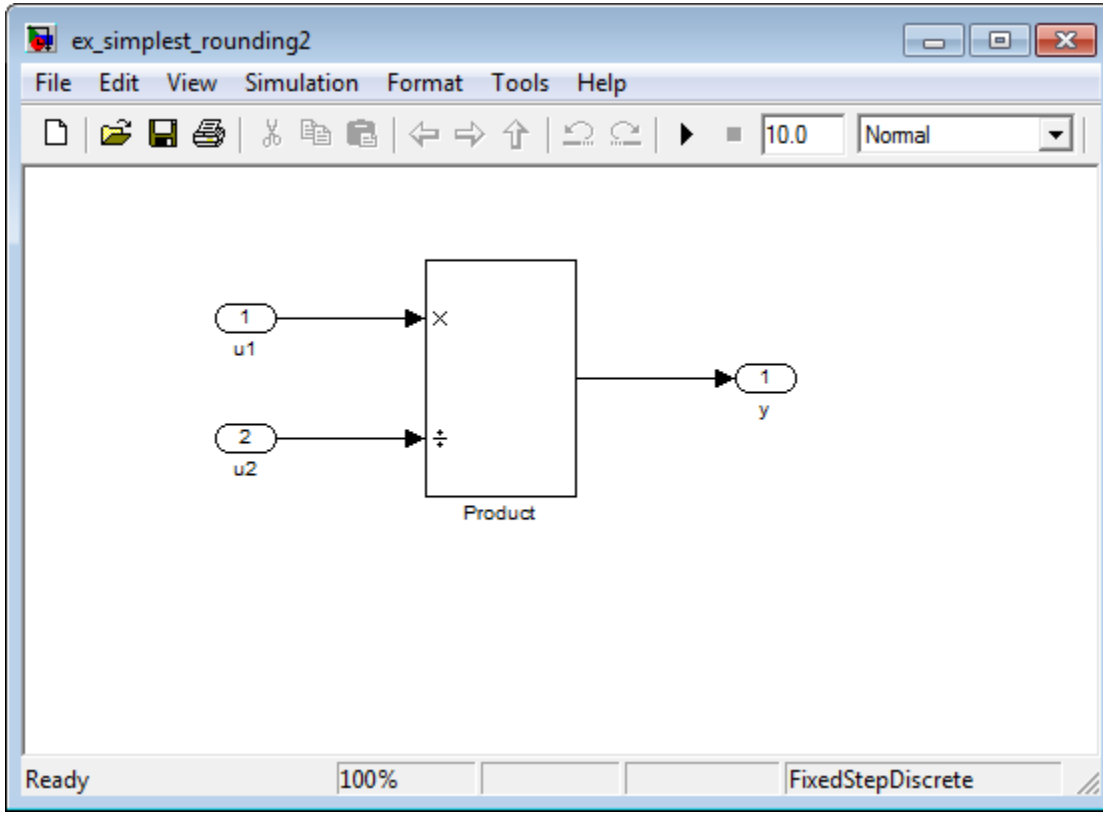
- Rounds to floor for all nondivision operations.
- Rounds to zero or floor for division, depending on the setting of the **Configuration Parameters > Hardware Implementation > Embedded Hardware > Signed integer division rounds to** parameter.

  To get the most efficient code, you must set the **Signed integer division rounds to** parameter to specify whether your production target rounds to zero or to floor for integer division. Most production targets round to zero for integer division operations. Note that `Simplest` rounding enables "mixed-mode" rounding for such cases, as it rounds to floor for multiplication and to zero for division.

  If the **Signed integer division rounds to** parameter is set to `Undefined`, the simplest rounding mode might not be able to produce the most efficient code. The simplest mode rounds to zero for division for this case, but it cannot rely on your production target to perform the rounding, because the parameter is `Undefined`. Therefore, you need additional rounding code to ensure rounding to zero behavior.

Note For signed fixed-point division where the denominator is an invariant constant power of 2, the simplest rounding mode does not generate the most efficient code. In this case, set the rounding mode to floor.

Optimize Rounding for Intermediate Arithmetic Operations. For fixed-point arithmetic with nonzero slope and bias, the simplest rounding mode also chooses the best rounding for each intermediate arithmetic operation. For example, consider the operation $y = u_1 / u_2$ implemented using a Product block, where u_1 and u_2 are fixed-point quantities:



As discussed in "Fixed-Point Numbers" on page 2-3, each fixed-point quantity is calculated using its slope, bias, and stored integer. So in this example, not only is there the high-level divide called for by the block operation, but intermediate additions and multiplies are performed:

$$y = \frac{u_1}{u_2} = \frac{S_1 Q_1 + B_1}{S_2 Q_2 + B_2}$$

The simplest rounding mode performs the best rounding for each of these operations, high-level and intermediate, to produce the most efficient code. The rules used to select the appropriate rounding for intermediate arithmetic operations are the same as those described in "Optimize Rounding for High-Level Arithmetic Operations" on page 3-14. Again, this enables mixed-mode rounding, with the most common case being round toward floor

used for additions, subtractions, and multiplies, and round toward zero used for divides.

Remember that generating the most efficient code using the simplest rounding mode requires you to set the **Configuration Parameters > Hardware Implementation > Embedded Hardware > Signed integer division rounds to** parameter to describe the rounding behavior of your production target.

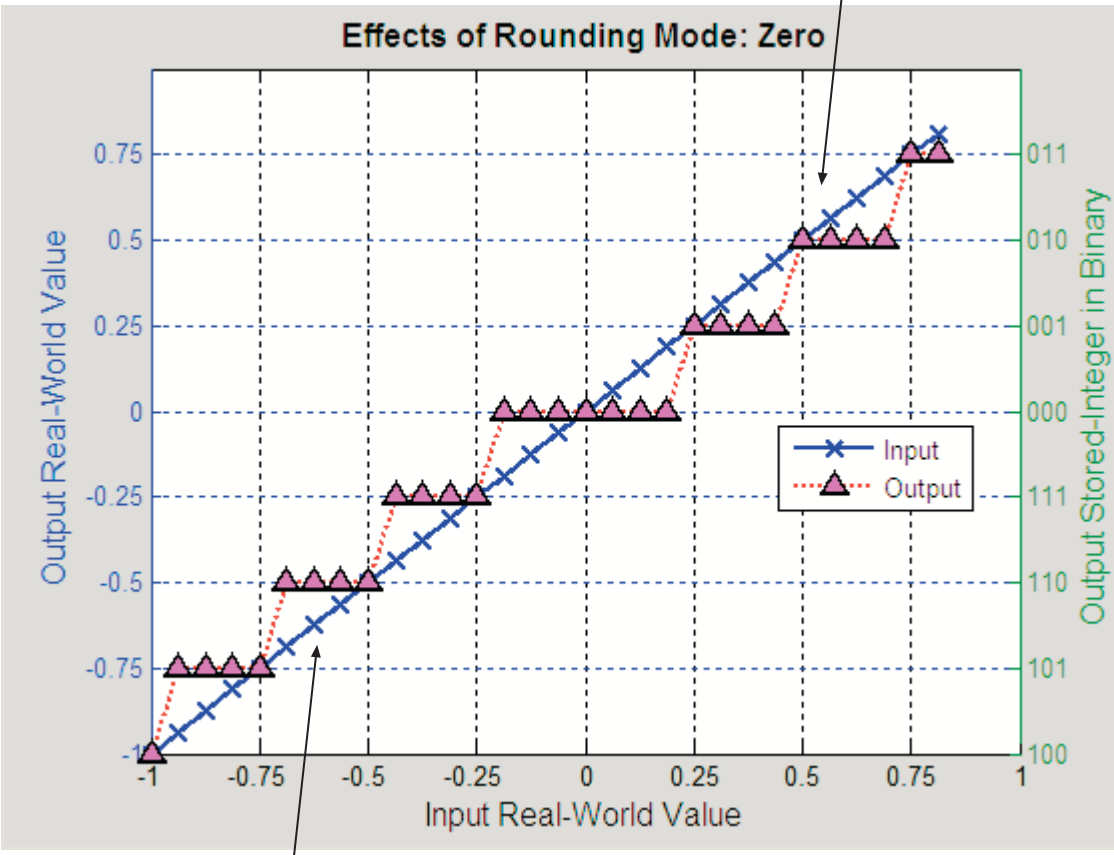
Note For signed fixed-point division where the denominator is an invariant constant power of 2, the simplest rounding mode does not generate the most efficient code. In this case, set the rounding mode to floor.

Rounding Mode: Zero

Rounding towards zero is the simplest rounding mode computationally. All digits beyond the number required are dropped. Rounding towards zero results in a number whose magnitude is always less than or equal to the more precise original value. In the MATLAB software, you can round to zero using the `fix` function.

Rounding toward zero introduces a cumulative downward bias in the result for positive numbers and a cumulative upward bias in the result for negative numbers. That is, all positive numbers are rounded to smaller positive numbers, while all negative numbers are rounded to smaller negative numbers. Rounding toward zero is shown in the following figure.

Positive numbers are rounded
to smaller positive numbers



Negative numbers are rounded
to smaller negative numbers

Example: Rounding to Zero Versus Truncation. Rounding to zero and *truncation* or *chopping* are sometimes thought to mean the same thing. However, the results produced by rounding to zero and truncation are different for unsigned and two's complement numbers. For this reason, the ambiguous term "truncation" is not used in this guide, and explicit rounding modes are used instead.

To illustrate this point, consider rounding a 5-bit unsigned number to zero by dropping (truncating) the two least significant bits. For example, the unsigned number 100.01 = 4.25 is truncated to 100 = 4. Therefore, truncating an unsigned number is equivalent to rounding to zero *or* rounding to floor.

Now consider rounding a 5-bit two's complement number by dropping the two least significant bits. At first glance, you may think truncating a two's complement number is the same as rounding to zero. For example, dropping the last two digits of -3.75 yields -3.00. However, digital hardware performing two's complement arithmetic yields a different result. Specifically, the number 100.01 = -3.75 truncates to 100 = -4, which is rounding to floor.

Padding with Trailing Zeros

Padding with trailing zeros involves extending the least significant bit (LSB) of a number with extra bits. This method involves going from low precision to higher precision.

For example, suppose two numbers are subtracted from each other. First, the exponents must be aligned, which typically involves a right shift of the number with the smaller value. In performing this shift, significant digits can "fall off" to the right. However, when the appropriate number of extra bits is appended, the precision of the result is maximized. Consider two 8-bit fixed-point numbers that are close in value and subtracted from each other:

$$1.0000000 \times 2^q - 1.1111111 \times 2^{q-1},$$

where q is an integer. To perform this operation, the exponents must be equal:

$$\begin{array}{r} 1.0000000 \times 2^q \\ \underline{-0.1111111 \times 2^q} \\ 0.0000001 \times 2^q \end{array}.$$

If the top number is padded by two zeros and the bottom number is padded with one zero, then the above equation becomes

$$\begin{array}{r} 1.000000000 \times 2^q \\ \underline{-0.111111110 \times 2^q} \\ 0.000000010 \times 2^q \end{array},$$

which produces a more precise result. An example of padding with trailing zeros in a Simulink model is illustrated in "Digital Controller Realization" on page 9-52.

Example: Limitations on Precision and Errors

Fixed-point variables have a limited precision because digital systems represent numbers with a finite number of bits. For example, suppose you must represent the real-world number 35.375 with a fixed-point

number. Using the encoding scheme described in "Scaling" on page 2-5, the representation is

$$V \approx \tilde{V} = SQ + B = 2^{-2}Q + 32,$$

where $V = 35.375$.

The two closest approximations to the real-world value are $Q = 13$ and $Q = 14$:

$$\tilde{V} = 2^{-2}(13) + 32 = 35.25,$$
$$\tilde{V} = 2^{-2}(14) + 32 = 35.50.$$

In either case, the absolute error is the same:

$$|\tilde{V} - V| = 0.125 = \frac{S}{2} = \frac{F2^E}{2}.$$

For fixed-point values within the limited range, this represents the worst-case error if round-to-nearest is used. If other rounding modes are used, the worst-case error can be twice as large:

$$|\tilde{V} - V| < F2^E.$$

Example: Maximizing Precision

Precision is limited by slope. To achieve maximum precision, you should make the slope as small as possible while keeping the range adequately large. The bias is adjusted in coordination with the slope.

Assume the maximum and minimum real-world values are given by max(V) and min(V), respectively. These limits might be known based on physical principles or engineering considerations. To maximize the precision, you must decide upon a rounding scheme and whether overflows saturate or wrap. To simplify matters, this example assumes the minimum real-world value corresponds to the minimum encoded value, and the maximum real-world value corresponds to the maximum encoded value. Using the encoding scheme described in "Scaling" on page 2-5, these values are given by

$$\max(V) = F2^E\left(\max(Q)\right) + B$$
$$\min(V) = F2^E\left(\min(Q)\right) + B.$$

Solving for the slope, you get

$$F2^E = \frac{\max(V) - \min(V)}{\max(Q) - \min(Q)} = \frac{\max(V) - \min(V)}{2^{ws} - 1}.$$

This formula is independent of rounding and overflow issues, and depends only on the word size, *ws*.

Net Slope and Net Bias Precision

What are Net Slope and Net Bias?

You can represent a fixed-point number by a general slope and bias encoding scheme

$$V \approx \tilde{V} = SQ + B,$$

where:

- V is an arbitrarily precise real-world value.
- $\tilde{V}$ is the approximate real-world value.
- Q, the stored value, is an integer that encodes V.
- $S = F2^E$ is the slope.
- B is the bias.

For a cast operation,

$$S_a Q_a + B_a = S_b Q_b + B_b$$

or

$$Q_a = \frac{S_b Q_b}{S_a} + \left(\frac{B_b - B_a}{S_a} \right),$$

where:

- $\frac{S_b}{S_a}$ is the net slope.
- $\frac{B_b - B_a}{S_a}$ is the net bias.

Detecting Net Slope and Net Bias Precision Issues

Precision issues might occur in the fixed-point constants, net slope and net bias, due to quantization errors when you convert from floating point to fixed point. These fixed-point constant precision issues can result in numerical inaccuracy in your model.

You can configure your model to alert you when fixed-point constant precision issues occur. For more information, see "How to Detect Net Slope and Net Bias Precision Issues" on page 3-25. The Simulink Fixed Point software provides the following information:

- The type of precision issue: underflow, overflow, or precision loss.
- The original value of the fixed-point constant.
- The quantized value of the fixed-point constant.
- The error in the value of the fixed-point constant.
- The block that introduced the error.

This information warns you that the outputs from this block are not accurate. If possible, change the data types in your model to fix the issue.

Fixed-Point Constant Underflow

Fixed-point constant underflow occurs when the Simulink Fixed Point software encounters a fixed-point constant whose data type does not have

enough precision to represent the ideal value of the constant, because the ideal value is too close to zero. Casting the ideal value to the fixed-point data type causes the value of the fixed-point constant to become zero. Therefore the value of the fixed-point constant differs from its ideal value.

Fixed-Point Constant Overflow

Fixed-point constant overflow occurs when the Simulink Fixed Point software converts a fixed-point constant to a data type whose range is not large enough to accommodate the ideal value of the constant with reasonable precision. The data type cannot accurately represent the ideal value because the ideal value is either too large or too small. Casting the ideal value to the fixed-point data type causes overflow. For example, suppose the ideal value is `200` and the converted data type is `int8`. Overflow occurs in this case because the maximum value that `int8` can represent is `127`.

The Simulink Fixed Point software reports an overflow error if the quantized value differs from the ideal value by more than the precision for the data type. The precision for a data type is approximately equal to the default scaling (for more information, see "Fixed-Point Data Type Parameters" on page 2-10.) Therefore, for positive values, the Simulink Fixed Point software treats errors greater than the slope as overflows. For negative values, it treats errors greater than or equal to the slope as overflows.

For example, the maximum value that `int8` can represent is `127`. The precision for `int8` is `1.0`. An ideal value of `127.3` quantizes to `127` with an absolute error of `0.3`. Although the ideal value `127.3` is greater than the maximum representable value for `int8`, the quantization error is small relative to the precision of `int8`. Therefore the Simulink Fixed Point software does not report an overflow. However, an ideal value of `128.1` does cause an overflow because the quantization error is `1.1`, which is larger than the precision for `int8`.

Note Fixed-point constant overflow differs from fixed-point constant precision loss. Precision loss occurs when the ideal fixed-point constant value is within the range of the current data type and scaling, but the software cannot represent this value exactly.

Fixed-Point Constant Precision Loss

Fixed-point constant precision loss occurs when the Simulink Fixed Point software converts a fixed-point constant to a data type without enough precision to represent the exact value of the constant. As a result, the quantized value differs from the ideal value. For an example of this behavior, see "Example: Fixed-Point Constant Precision Loss" on page 3-26.

Note Fixed-point constant precision loss differs from fixed-point constant overflow. Overflow occurs when the range of the parameter data type, that is, the maximum value that it can represent, is smaller than the ideal value of the parameter.

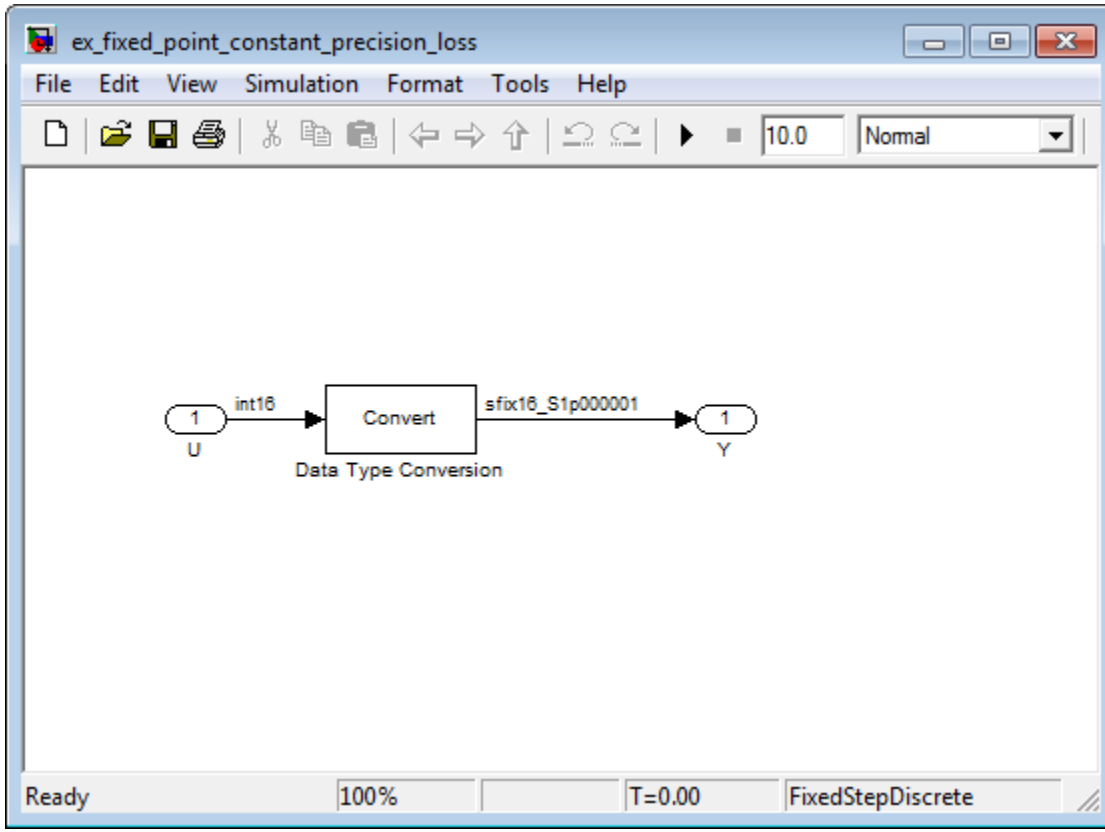
How to Detect Net Slope and Net Bias Precision Issues

To receive alerts when fixed-point constant precision issues occur, use these options available in the Simulink Configuration Parameters dialog box, on the **Diagnostics > Type Conversion** pane. Set the parameters to `warning` or `error` so that Simulink alerts you when precision issues occur.

Configuration Parameter	Specifies	Default
"Detect underflow"	Diagnostic action when a fixed-point constant underflow occurs during simulation	Does not generate a warning or error.
"Detect overflow"	Diagnostic action when a fixed-point constant overflow occurs during simulation	Does not generate a warning or error.
"Detect precision loss"	Diagnostic action when a fixed-point constant precision loss occurs during simulation	Does not generate a warning or error.

Example: Fixed-Point Constant Precision Loss

This example demonstrates how to detect fixed-point constant precision loss. The example uses the following model.



For the Data Type Conversion block in this model, the:

- Input slope, $S_U = 1$
- Output slope, $S_Y = 1.000001$
- Net slope, $S_U/S_Y = 1/1.000001$

When you simulate the model, a net slope quantization error occurs.

To set up the model and run the simulation:

1. For the Inport block, set the **Output data type** to `int16`.
2. For the Data Type Conversion block, set the **Output data type** to `fixdt(1,16, 1.000001, 0)`.
3. Set the **Diagnostics > Type Conversion > Detect precision loss** configuration parameter to `error`.
4. In your Simulink model window, select **Simulation > Start**.

   The Simulink Fixed Point software generates an error informing you that net scaling quantization caused precision loss. The message provides the following information:

   - The block that introduced the error.
   - The original value of the net slope.
   - The quantized value of the net slope.
   - The error in the value of the net slope.

Limitations on Range

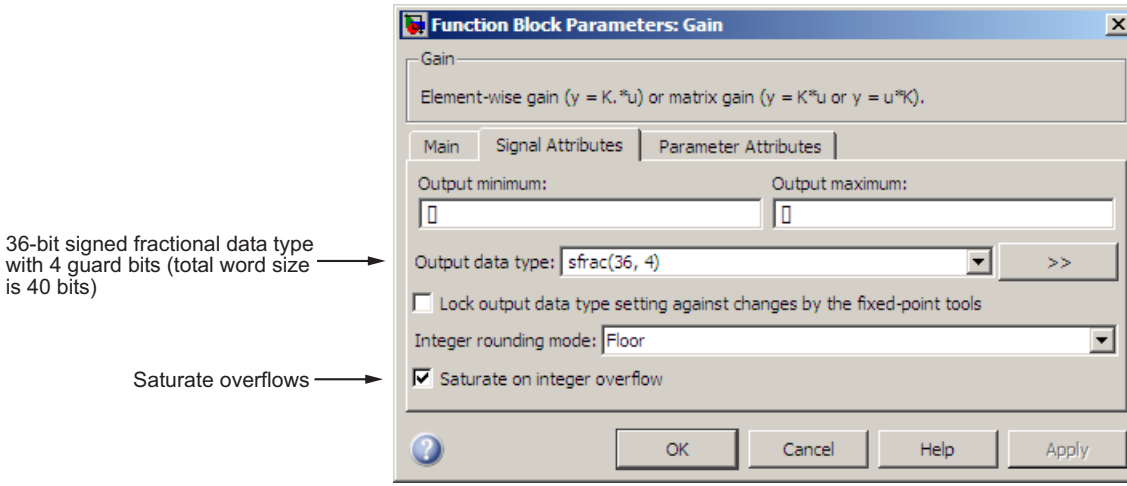
In this section...

Introduction

Limitations on the range of a fixed-point word occur for the same reason as limitations on its precision. Namely, fixed-point words have limited size. For a general discussion of range and precision, refer to "Range and Precision" on page 2-9.

In binary arithmetic, a processor might need to take an n-bit fixed-point number and store it in m bits, where $m \neq n$. If m < n, the range of the number has been reduced and an operation can produce an overflow condition. Some processors identify this condition as `Inf` or `NaN`. For other processors, especially digital signal processors (DSPs), the value *saturates* or *wraps*. If m > n, the range of the number has been extended. Extending the range of a word requires the inclusion of *guard bits*, which act to guard against potential overflow. In both cases, the range depends on the word's size and scaling.

The Simulink software supports saturation and wrapping for all fixed-point data types, while guard bits are supported only for fractional data types. As shown in the following figure, you can select saturation or wrapping for fixed-point Simulink blocks with the **Saturate on integer overflow** check box, and you can specify guard bits with the **Output data type** parameter.

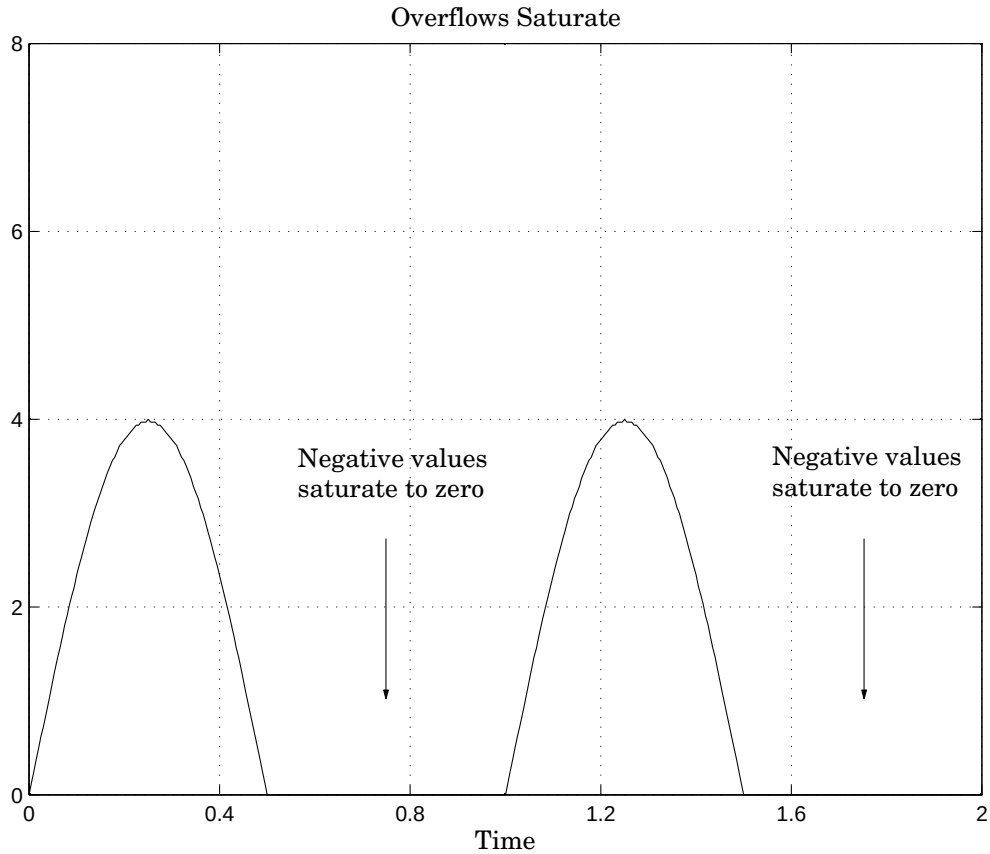


Saturation and Wrapping

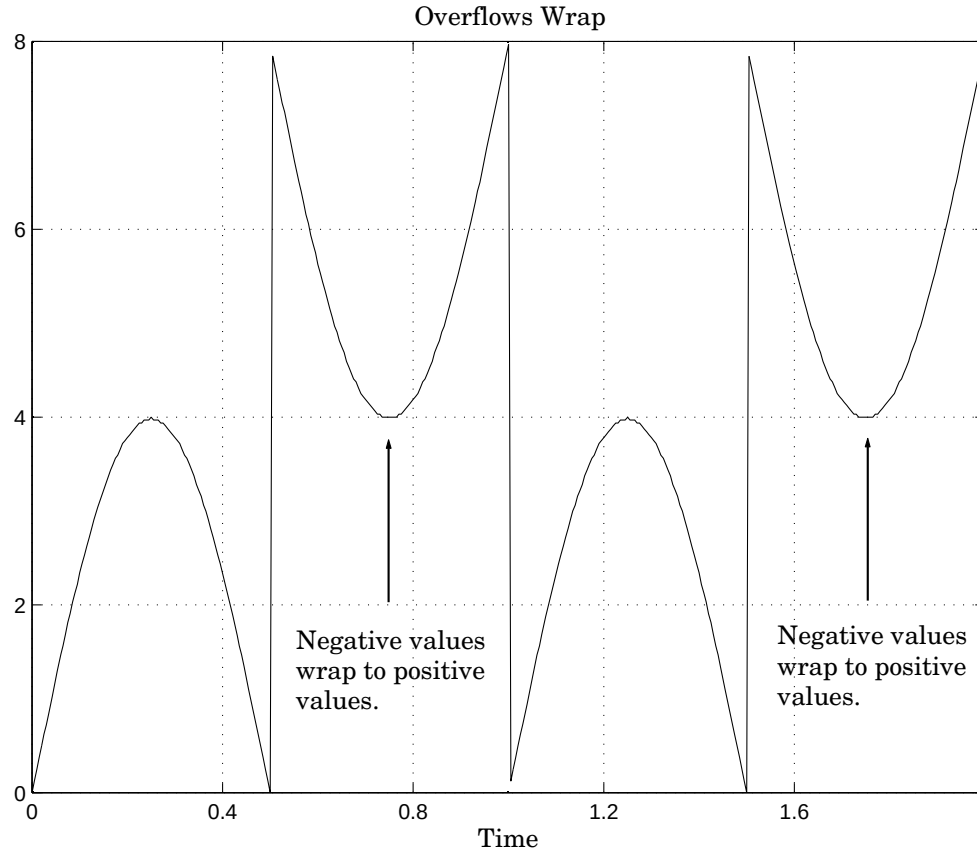
Saturation and wrapping describe a particular way that some processors deal with overflow conditions. For example, the ADSP-2100 family of processors from Analog Devices™ supports either of these modes. If a register has a saturation mode of operation, then an overflow condition is set to the maximum positive or negative value allowed. Conversely, if a register has a wrapping mode of operation, an overflow condition is set to the appropriate value within the range of the representation.

Example: Saturation and Wrapping

Consider an 8-bit unsigned word with binary-point-only scaling of 2^{-5}. Suppose this data type must represent a sine wave that ranges from -4 to 4. For values between 0 and 4, the word can represent these numbers without regard to overflow. This is not the case with negative numbers. If overflows saturate, all negative values are set to zero, which is the smallest number representable by the data type. The saturation of overflows is shown in the following figure.



If overflows wrap, all negative values are set to the appropriate positive value. The wrapping of overflows is shown in the following figure.



Note For most control applications, saturation is the safer way of dealing with fixed-point overflow. However, some processor architectures allow automatic saturation by hardware. If hardware saturation is not available, then extra software is required, resulting in larger, slower programs. This cost is justified in some designs—perhaps for safety reasons. Other designs accept wrapping to obtain the smallest, fastest software.

Guard Bits

You can eliminate the possibility of overflow by appending the appropriate number of guard bits to a binary word.

For a two's complement signed value, the guard bits are filled with either 0's or 1's depending on the value of the most significant bit (MSB). This is called *sign extension*. For example, consider a 4-bit two's complement number with value 1011. If this number is extended in range to 7 bits with sign extension, then the number becomes 1111101 and the value remains the same.

Guard bits are supported only for fractional data types. For both signed and unsigned fractionals, the guard bits lie to the left of the default binary point.

Example: Limitations on Range

Fixed-point variables have a limited range for the same reason they have limited precision—because digital systems represent numbers with a finite number of bits. As a general example, consider the case where an integer is represented as a fixed-point word of size *ws*. The range for signed and unsigned words is given by

$$\max(Q) - \min(Q),$$

where

$$\min(Q) = \begin{cases} 0 & \text{unsigned,} \\ -2^{ws-1} & \text{signed,} \end{cases}$$

$$\max(Q) = \begin{cases} 2^{ws} - 1 & \text{unsigned,} \\ 2^{ws-1} - 1 & \text{signed.} \end{cases}$$

Using the general [Slope Bias] encoding scheme described in "Scaling" on page 2-5, the approximate real-world value has the range

$$\max(\tilde{V}) - \min(\tilde{V}),$$

where

$$\min(\tilde{V}) = \begin{cases} \mathrm{B} & \text{unsigned,} \\ -F2^{E}\left(2^{ws-1}\right) + B & \text{signed,} \end{cases}$$

$$\max(\tilde{V}) = \begin{cases} F2^{E}\left(2^{ws} - 1\right) + \mathrm{B} & \text{unsigned,} \\ F2^{E}\left(2^{ws-1} - 1\right) + \mathrm{B} & \text{signed.} \end{cases}$$

If the real-world value exceeds the limited range of the approximate value, then the accuracy of the representation can become significantly worse.

Recommendations for Arithmetic and Scaling

In this section...

Introduction

The sections that follow describe the relationship between arithmetic operations and fixed-point scaling, and offer some basic recommendations that may be appropriate for your fixed-point design. For each arithmetic operation,

- The general [Slope Bias] encoding scheme described in "Scaling" on page 2-5 is used.
- The scaling of the result is automatically selected based on the scaling of the two inputs. In other words, the scaling is *inherited*.
- Scaling choices are based on
  - Minimizing the number of arithmetic operations of the result
  - Maximizing the precision of the result

  Additionally, binary-point-only scaling is presented as a special case of the general encoding scheme.

In embedded systems, the scaling of variables at the hardware interface (the ADC or DAC) is fixed. However for most other variables, the scaling is something you can choose to give the best design. When scaling fixed-point variables, it is important to remember that

- Your scaling choices depend on the particular design you are simulating.

- There is no best scaling approach. All choices have associated advantages and disadvantages. It is the goal of this section to expose these advantages and disadvantages to you.

Addition

Consider the addition of two real-world values:

$$V_a = V_b + V_c.$$

These values are represented by the general [Slope Bias] encoding scheme described in "Scaling" on page 2-5:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

In a fixed-point system, the addition of values results in finding the variable Q_a:

$$Q_a = \frac{F_b}{F_a} 2^{E_b - E_a} Q_b + \frac{F_c}{F_a} 2^{E_c - E_a} Q_c + \frac{B_b + B_c - B_a}{F_a} 2^{-E_a}.$$

This formula shows

- In general, Q_a is not computed through a simple addition of Q_b and Q_c.
- In general, there are two multiplications of a constant and a variable, two additions, and some additional bit shifting.

Inherited Scaling for Speed

In the process of finding the scaling of the sum, one reasonable goal is to simplify the calculations. Simplifying the calculations should reduce the number of operations, thereby increasing execution speed. The following choices can help to minimize the number of arithmetic operations:

- Set $B_a = B_b + B_c$. This eliminates one addition.
- Set $F_a = F_b$ or $F_a = F_c$. Either choice eliminates one of the two constant times variable multiplications.

The resulting formula is

$$Q_a = 2^{E_b - E_a} Q_b + \frac{F_c}{F_a} 2^{E_c - E_a} Q_c$$

or

$$Q_a = \frac{F_b}{F_a} 2^{E_b - E_a} Q_b + 2^{E_c - E_a} Q_c.$$

These equations appear to be equivalent. However, your choice of rounding and precision may make one choice stand out over the other. To further simplify matters, you could choose $E_a = E_c$ or $E_a = E_b$. This will eliminate some bit shifting.

Inherited Scaling for Maximum Precision

In the process of finding the scaling of the sum, one reasonable goal is maximum precision. You can determine the maximum-precision scaling if the range of the variable is known. "Example: Maximizing Precision" on page 3-21 shows that you can determine the range of a fixed-point operation from $\max(V_a)$ and $\min(V_a)$. For a summation, you can determine the range from

$$\min\left(\tilde{V}_a\right) = \min\left(\tilde{V}_b\right) + \min\left(\tilde{V}_c\right),$$
$$\max\left(\tilde{V}_a\right) = \max\left(\tilde{V}_b\right) + \max\left(\tilde{V}_c\right).$$

You can now derive the maximum-precision slope:

$$\begin{aligned} F_a 2^{E_a} &= \frac{\max\left(\tilde{V}_a\right) - \min\left(\tilde{V}_a\right)}{2^{ws_a} - 1} \\ &= \frac{F_a 2^{E_b}\left(2^{ws_b} - 1\right) + F_c 2^{E_c}\left(2^{ws_c} - 1\right)}{2^{ws_a} - 1}. \end{aligned}$$

In most cases the input and output word sizes are much greater than one, and the slope becomes

$$F_a 2^{E_a} \approx F_b 2^{E_b + ws_b - ws_a} + F_c 2^{E_c + ws_c - ws_a},$$

which depends only on the size of the input and output words. The corresponding bias is

$$B_a = \min\left(\tilde{V}_a\right) - F_a 2^{E_a} \min\left(Q_a\right).$$

The value of the bias depends on whether the inputs and output are signed or unsigned numbers.

If the inputs and output are all unsigned, then the minimum values for these variables are all zero and the bias reduces to a particularly simple form:

$$B_a = B_b + B_c.$$

If the inputs and the output are all signed, then the bias becomes

$$B_a \approx B_b + B_c + F_b 2^{E_b}\left(-2^{ws_b - 1} + 2^{ws_b - 1}\right) + F_c 2^{E_c}\left(-2^{ws_c - 1} + 2^{ws_c - 1}\right),$$
$$B_a \approx B_b + B_c.$$

Binary-Point-Only Scaling

For binary-point-only scaling, finding Q_a results in this simple expression:

$$Q_a = 2^{E_b - E_a} Q_b + 2^{E_c - E_a} Q_c.$$

This scaling choice results in only one addition and some bit shifting. The avoidance of any multiplications is a big advantage of binary-point-only scaling.

Note The subtraction of values produces results that are analogous to those produced by the addition of values.

Accumulation

The accumulation of values is closely associated with addition:

$$V_{a\_new} = V_{a\_old} + V_b.$$

Finding Q_{a_new} involves one multiplication of a constant and a variable, two additions, and some bit shifting:

$$Q_{a\_new} = Q_{a\_old} + \frac{F_b}{F_a} 2^{E_b - E_a} Q_b + \frac{B_b}{F_a} 2^{-E_a}.$$

The important difference for fixed-point implementations is that the scaling of the output is identical to the scaling of the first input.

Binary-Point-Only Scaling

For binary-point-only scaling, finding Q_{a_new} results in this simple expression:

$$Q_{a\_new} = Q_{a\_old} + 2^{E_b - E_a} Q_b.$$

This scaling option only involves one addition and some bit shifting.

Note The negative accumulation of values produces results that are analogous to those produced by the accumulation of values.

Multiplication

Consider the multiplication of two real-world values:

$$V_a = V_b V_c.$$

These values are represented by the general [Slope Bias] encoding scheme described in “Scaling” on page 2-5:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

In a fixed-point system, the multiplication of values results in finding the variable Q_a:

$$Q_a = \frac{F_b F_c}{F_a} 2^{E_b + E_c - E_a} Q_b Q_c + \frac{F_b B_c}{F_a} 2^{E_b - E_a} Q_b + \frac{F_c B_b}{F_a} 2^{E_c - E_a} Q_c + \frac{B_b B_c - B_a}{F_a} 2^{-E_a}.$$

This formula shows

- In general, Q_a is not computed through a simple multiplication of Q_b and Q_c.
- In general, there is one multiplication of a constant and two variables, two multiplications of a constant and a variable, three additions, and some additional bit shifting.

Inherited Scaling for Speed

The number of arithmetic operations can be reduced with these choices:

- Set $B_a = B_b B_c$. This eliminates one addition operation.
- Set $F_a = F_b F_c$. This simplifies the triple multiplication—certainly the most difficult part of the equation to implement.
- Set $E_a = E_b + E_c$. This eliminates some of the bit shifting.

The resulting formula is

$$Q_a = Q_b Q_c + \frac{B_c}{F_c} 2^{-E_c} Q_b + \frac{B_b}{F_b} 2^{-E_b} Q_c.$$

Inherited Scaling for Maximum Precision

You can determine the maximum-precision scaling if the range of the variable is known. "Example: Maximizing Precision" on page 3-21 shows that you can determine the range of a fixed-point operation from

$$\max\left(\tilde{V}_a\right)$$

and

$$\min\left(\tilde{V}_a\right).$$

For multiplication, you can determine the range from

$$\min\left(\tilde{V}_a\right) = \min\left(V_{LL}, V_{LH}, V_{HL}, V_{HH}\right),$$
$$\max\left(\tilde{V}_a\right) = \max\left(V_{LL}, V_{LH}, V_{HL}, V_{HH}\right),$$

where

$$V_{LL} = \min\left(\tilde{V}_b\right) \cdot \min\left(\tilde{V}_c\right),$$
$$V_{LH} = \min\left(\tilde{V}_b\right) \cdot \max\left(\tilde{V}_c\right),$$
$$V_{HL} = \max\left(\tilde{V}_b\right) \cdot \min\left(\tilde{V}_c\right),$$
$$V_{HH} = \max\left(\tilde{V}_b\right) \cdot \max\left(\tilde{V}_c\right).$$

Binary-Point-Only Scaling

For binary-point-only scaling, finding Q_a results in this simple expression:

$$Q_a = 2^{E_b + E_c - E_a} Q_b Q_c.$$

Gain

Consider the multiplication of a constant and a variable

$$V_a = K V_b,$$

where K is a constant called the gain. Since V_a results from the multiplication of a constant and a variable, finding Q_a is a simplified version of the general fixed-point multiplication formula:

$$Q_a = \left(\frac{K F_b 2^{E_b}}{F_a 2^{E_a}}\right) Q_b + \left(\frac{K B_b - B_a}{F_a 2^{E_a}}\right).$$

Note that the terms in the parentheses can be calculated offline. Therefore, there is only one multiplication of a constant and a variable and one addition.

To implement the above equation without changing it to a more complicated form, the constants need to be encoded using a binary-point-only format. For each of these constants, the range is the trivial case of only one value. Despite the trivial range, the binary point formulas for maximum precision are still valid. The maximum-precision representations are the most useful choices unless there is an overriding need to avoid any shifting. The encoding of the constants is

$$\left(\frac{KF_b 2^{E_b}}{F_a 2^{E_a}}\right) = 2^{E_X} Q_X$$

$$\left(\frac{KB_b - B_a}{F_a 2^{E_a}}\right) = 2^{E_Y} Q_Y$$

resulting in the formula

$$Q_a = 2^{E_X} Q_X Q_B + 2^{E_Y} Q_Y .$$

Inherited Scaling for Speed

The number of arithmetic operations can be reduced with these choices:

- Set $B_a = KB_b$. This eliminates one constant term.
- Set $F_a = KF_b$ and $E_a = E_b$. This sets the other constant term to unity.

  The resulting formula is simply

  $$Q_a = Q_b .$$

If the number of bits is different, then either handling potential overflows or performing sign extensions is the only possible operation involved.

Inherited Scaling for Maximum Precision

The scaling for maximum precision does not need to be different from the scaling for speed unless the output has fewer bits than the input. If this is the case, then saturation should be avoided by dividing the slope by 2 for each lost bit. This prevents saturation but causes rounding to occur.

Division

Division of values is an operation that should be avoided in fixed-point embedded systems, but it can occur in places. Therefore, consider the division of two real-world values:

$$V_a = V_b / V_c .$$

These values are represented by the general [Slope Bias] encoding scheme described in "Scaling" on page 2-5:

$$V_i = F_i 2^{E_i} Q_i + B_i .$$

In a fixed-point system, the division of values results in finding the variable Q_a:

$$Q_a = \frac{F_b 2^{E_b} Q_b + B_b}{F_c F_a 2^{E_c + E_a} Q_c + B_c F_a 2^{E_a}} - \frac{B_a}{F_a} 2^{-E_a} .$$

This formula shows

- In general, Q_a is not computed through a simple division of Q_b by Q_c.
- In general, there are two multiplications of a constant and a variable, two additions, one division of a variable by a variable, one division of a constant by a variable, and some additional bit shifting.

Inherited Scaling for Speed

The number of arithmetic operations can be reduced with these choices:

- Set $B_a = 0$. This eliminates one addition operation.

- If $B_c = 0$, then set the fractional slope $F_a = F_b/F_c$. This eliminates one constant times variable multiplication.

The resulting formula is

$$Q_a = \frac{Q_b}{Q_c} 2^{E_b - E_c - E_a} + \frac{(B_b/F_b)}{Q_c} 2^{-E_c - E_a}.$$

If $B_c \neq 0$, then no clear recommendation can be made.

Inherited Scaling for Maximum Precision

You can determine the maximum-precision scaling if the range of the variable is known. "Example: Maximizing Precision" on page 3-21 shows that you can determine the range of a fixed-point operation from

$$\max(\tilde{V}_a)$$

and

$$\min(\tilde{V}_a).$$

For division, you can determine the range from

$$\min(\tilde{V}_a) = \min(V_{LL}, V_{LH}, V_{HL}, V_{HH}),$$
$$\max(\tilde{V}_a) = \max(V_{LL}, V_{LH}, V_{HL}, V_{HH}),$$

where for nonzero denominators

$$V_{LL} = \min(\tilde{V}_b)/\min(\tilde{V}_c),$$
$$V_{LH} = \min(\tilde{V}_b)/\max(\tilde{V}_c),$$
$$V_{HL} = \max(\tilde{V}_b)/\min(\tilde{V}_c),$$
$$V_{HH} = \max(\tilde{V}_b)/\max(\tilde{V}_c).$$

Binary-Point-Only Scaling

For binary-point-only scaling, finding Q_a results in this simple expression:

$$Q_a = \frac{Q_b}{Q_c} 2^{E_b - E_c - E_a}.$$

Note For the last two formulas involving Q_a, a divide by zero and zero divided by zero are possible. In these cases, the hardware will give some default behavior but you must make sure that these default responses give meaningful results for the embedded system.

Summary

From the previous analysis of fixed-point variables scaled within the general [Slope Bias] encoding scheme, you can conclude

- Addition, subtraction, multiplication, and division can be very involved unless certain choices are made for the biases and slopes.
- Binary-point-only scaling guarantees simpler math, but generally sacrifices some precision.

Note that the previous formulas don't show the following:

- Constants and variables are represented with a finite number of bits.
- Variables are either signed or unsigned.
- Rounding and overflow handling schemes. You must make these decisions before an actual fixed-point realization is achieved.

Parameter and Signal Conversions

In this section...

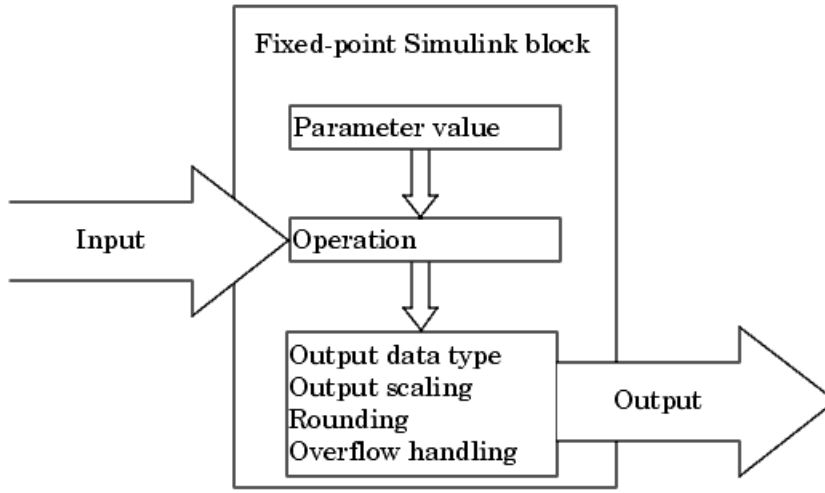
Introduction

The previous sections of this chapter, together with Chapter 2, "Data Types and Scaling" describe how data types, scaling, rounding, overflow handling, and arithmetic operations are incorporated into the Simulink software's fixed-point support. With this knowledge, you can define the output of a fixed-point model by configuring fixed-point blocks to suit your particular application.

However, to completely understand the results generated by fixed-point Simulink blocks, you must be aware of these issues:

- When numerical block parameters are converted from doubles to Simulink Fixed Point data types
- When input signals are converted from one Simulink Fixed Point data type to another (if at all)
- When arithmetic operations on input signals and parameters are performed

For example, suppose a fixed-point Simulink block performs an arithmetic operation on its input signal and a parameter, and then generates output having characteristics that are specified by the block. The following diagram illustrates how these issues are related.



The sections that follow describe parameter and signal conversions. "Rules for Arithmetic Operations" on page 3-50 discusses arithmetic operations.

Parameter Conversions

Parameters of fixed-point blocks that accept numerical values are always converted from `double` to a fixed-point data type. Parameters can be converted to the input data type, the output data type, or to a data type explicitly specified by the block. For example, the Discrete FIR Filter block converts its **Initial states** parameter to the input data type, and converts its **Numerator coefficient** parameter to a data type you explicitly specify via the block dialog box.

Parameters are always converted before any arithmetic operations are performed. Additionally, parameters are always converted *offline* using round-to-nearest and saturation. Offline conversions are discussed below.

Note Because parameters of fixed-point blocks begin as `double`, they are never precise to more than 53 bits. Therefore, if the output of your fixed-point block is longer than 53 bits, your result might be less precise than you anticipated.

Offline Conversions

An offline conversion is a conversion performed by your development platform (for example, the processor on your PC), and not by the fixed-point processor you are targeting. For example, suppose you are using a PC to develop a program to run on a fixed-point processor, and you need the fixed-point processor to compute

$$y = \left(\frac{ab}{c}\right)u = Cu$$

over and over again. If a, b, and c are constant parameters, it is inefficient for the fixed-point processor to compute ab/c every time. Instead, the PC's processor should compute ab/c offline one time, and the fixed-point processor computes only $C \cdot u$. This eliminates two costly fixed-point arithmetic operations.

Signal Conversions

Consider the conversion of a real-world value from one fixed-point data type to another. Ideally, the values before and after the conversion are equal.

$$V_a = V_b,$$

where V_b is the input value and V_a is the output value. To see how the conversion is implemented, the two ideal values are replaced by the general [Slope Bias] encoding scheme described in "Scaling" on page 2-5:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

Solving for the output data type's stored integer value, Q_a is obtained:

$$
\begin{aligned}
Q_a &= \frac{F_b}{F_a} 2^{E_b - E_a} Q_b + \frac{B_b - B_a}{F_a} 2^{-E_a} \\
&= F_s 2^{E_b - E_a} Q_b + B_{net},
\end{aligned}
$$

where F_s is the adjusted fractional slope and B_{net} is the net bias. The offline conversions and online conversions and operations are discussed below.

Offline Conversions

Both F_s and B_{net} are computed offline using round-to-nearest and saturation. B_{net} is then stored using the output data type and F_s is stored using an automatically selected data type.

Online Conversions and Operations

The remaining conversions and operations are performed *online* by the fixed-point processor, and depend on the slopes and biases for the input and output data types. The conversions and operations are given by these steps:

1. The initial value for Q_a is given by the net bias, B_{net}:

   $$Q_a = B_{net}.$$

2. The input integer value, Q_b, is multiplied by the adjusted slope, F_s:

   $$Q_{RawProduct} = F_s Q_b.$$

3. The result of step 2 is converted to the modified output data type where the slope is one and bias is zero:

   $$Q_{Temp} = convert(Q_{RawProduct}).$$

   This conversion includes any necessary bit shifting, rounding, or overflow handling.

4. The summation operation is performed:

   $$Q_a = Q_{Temp} + Q_a.$$

This summation includes any necessary overflow handling.

Streamlining Simulations and Generated Code

Note that the maximum number of conversions and operations is performed when the slopes and biases of the input signal and output signal differ (are mismatched). If the scaling of these signals is identical (matched), the number of operations is reduced from the worst (most inefficient) case. For example, when an input has the same fractional slope and bias as the output, only step 3 is required:

$$Q_a = convert(Q_b).$$

Exclusive use of binary-point-only scaling for both input signals and output signals is a common way to eliminate mismatched slopes and biases, and results in the most efficient simulations and generated code.

Rules for Arithmetic Operations

In this section...

Introduction

Fixed-point arithmetic refers to how signed or unsigned binary words are operated on. The simplicity of fixed-point arithmetic functions such as addition and subtraction allows for cost-effective hardware implementations.

The sections that follow describe the rules that the Simulink software follows when arithmetic operations are performed on inputs and parameters. These rules are organized into four groups based on the operations involved: addition and subtraction, multiplication, division, and shifts. For each of these four groups, the rules for performing the specified operation are presented with an example using the rules.

Note For information about calculations using Fixed-Point Toolbox software, see the *Fixed-Point Toolbox User's Guide*.

Computational Units

The core architecture of many processors contains several computational units including arithmetic logic units (ALUs), multiply and accumulate units (MACs), and shifters. These computational units process the binary data directly and provide support for arithmetic computations of varying precision. The ALU performs a standard set of arithmetic and logic operations as well as division. The MAC performs multiply, multiply/add, and multiply/subtract

operations. The shifter performs logical and arithmetic shifts, normalization, denormalization, and other operations.

Addition and Subtraction

Addition is the most common arithmetic operation a processor performs. When two n-bit numbers are added together, it is always possible to produce a result with n + 1 nonzero digits due to a carry from the leftmost digit. For two's complement addition of two numbers, there are three cases to consider:

- If both numbers are positive and the result of their addition has a sign bit of 1, then overflow has occurred; otherwise the result is correct.
- If both numbers are negative and the sign of the result is 0, then overflow has occurred; otherwise the result is correct.
- If the numbers are of unlike sign, overflow cannot occur and the result is always correct.

Fixed-Point Simulink Blocks Summation Process

Consider the summation of two numbers. Ideally, the real-world values obey the equation

$$V_a = \pm V_b \pm V_c,$$

where V_b and V_c are the input values and V_b is the output value. To see how the summation is actually implemented, the three ideal values should be replaced by the general [Slope Bias] encoding scheme described in "Scaling" on page 2-5:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

The equation in "Addition" on page 3-35 gives the solution of the resulting equation for the stored integer, Q_a. Using shorthand notation, that equation becomes

$$Q_a = \pm F_{sb} 2^{E_b - E_a} Q_b \pm F_{sc} 2^{E_c - E_a} Q_c + B_{net},$$

where F_{sb} and F_{sc} are the adjusted fractional slopes and B_{net} is the net bias. The offline conversions and online conversions and operations are discussed below.

Offline Conversions. F_{sb}, F_{sc}, and B_{net} are computed offline using round-to-nearest and saturation. Furthermore, B_{net} is stored using the output data type.

Online Conversions and Operations. The remaining operations are performed online by the fixed-point processor, and depend on the slopes and biases for the input and output data types. The worst (most inefficient) case occurs when the slopes and biases are mismatched. The worst-case conversions and operations are given by these steps:

1. The initial value for Q_a is given by the net bias, B_{net}:

   $$Q_a = B_{net}.$$

2. The first input integer value, Q_b, is multiplied by the adjusted slope, F_{sb}:

   $$Q_{RawProduct} = F_{sb} Q_b.$$

3. The previous product is converted to the modified output data type where the slope is one and the bias is zero:

   $$Q_{Temp} = convert\left(Q_{RawProduct}\right).$$

   This conversion includes any necessary bit shifting, rounding, or overflow handling.

4. The summation operation is performed:

   $$Q_a = Q_a \pm Q_{Temp}.$$

   This summation includes any necessary overflow handling.

5. Steps 2 to 4 are repeated for every number to be summed.

It is important to note that bit shifting, rounding, and overflow handling are applied to the intermediate steps (3 and 4) and not to the overall sum.

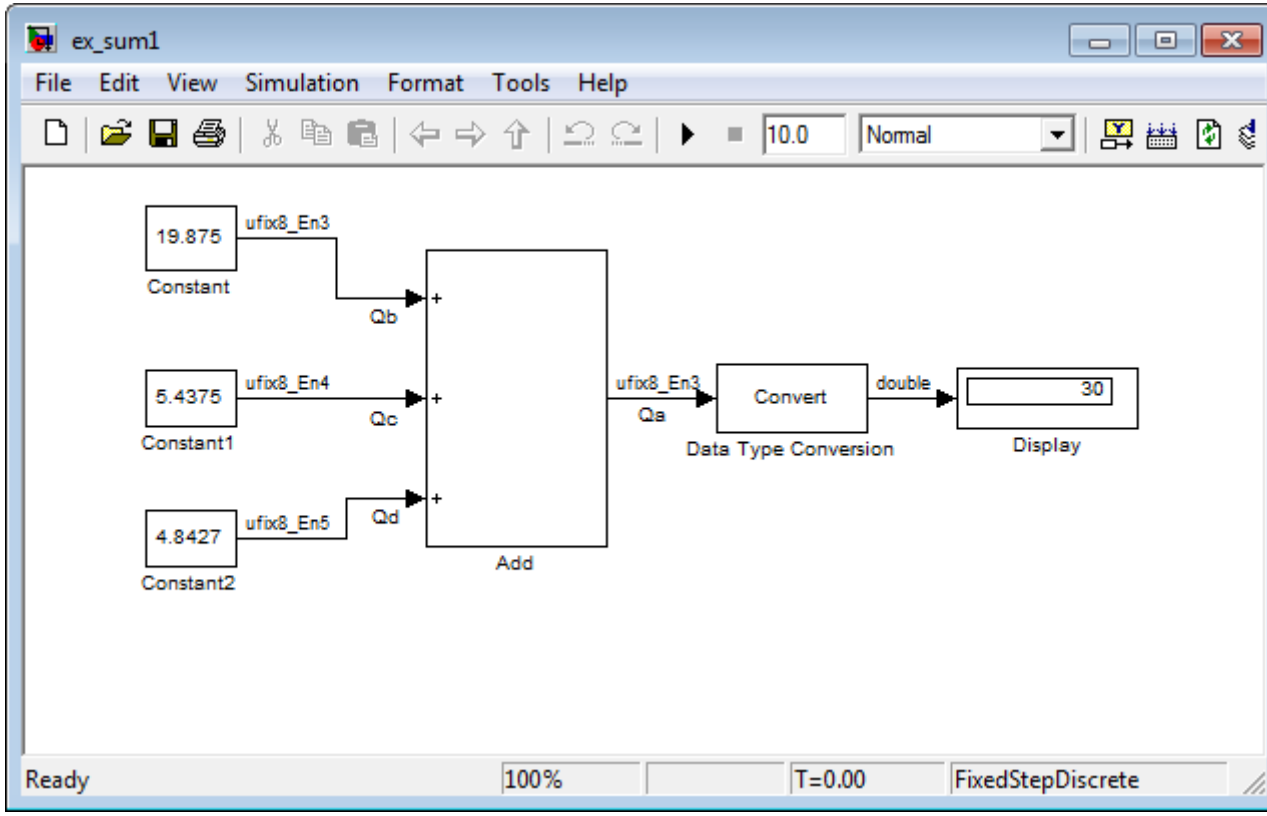
Streamlining Simulations and Generated Code

If the scaling of the input and output signals is matched, the number of summation operations is reduced from the worst (most inefficient) case. For example, when an input has the same fractional slope as the output, step 2 reduces to multiplication by one and can be eliminated. Trivial steps in the summation process are eliminated for both simulation and code generation. Exclusive use of binary-point-only scaling for both input signals and output signals is a common way to eliminate mismatched slopes and biases, and results in the most efficient simulations and generated code.

Example: The Summation Process

Suppose you want to sum three numbers. Each of these numbers is represented by an 8-bit word, and each has a different binary-point-only scaling. Additionally, the output is restricted to an 8-bit word with binary-point-only scaling of 2^{-3}.

The summation is shown in the following model for the input values 19.875, 5.4375, and 4.84375.



Applying the rules from the previous section, the sum follows these steps:

1 Because the biases are matched, the initial value of Q_a is trivial:

$$Q_a = 00000.000.$$

2 The first number to be summed (19.875) has a fractional slope that matches the output fractional slope. Furthermore, the binary points and storage types are identical, so the conversion is trivial:

$$Q_b = 10011.111,$$
$$Q_{Temp} = Q_b.$$

3 The summation operation is performed:

$$Q_a = Q_a + Q_{Temp} = 10011.111.$$

4 The second number to be summed (5.4375) has a fractional slope that matches the output fractional slope, so a slope adjustment is not needed. The storage data types also match, but the difference in binary points requires that both the bits and the binary point be shifted one place to the right:

$$Q_c = 0101.0111,$$
$$Q_{Temp} = convert(Q_c)$$
$$Q_{Temp} = 00101.011.$$

Note that a loss in precision of one bit occurs, with the resulting value of Q_{Temp} determined by the rounding mode. For this example, round-to-floor is used. Overflow cannot occur in this case because the bits and binary point are both shifted to the right.

5 The summation operation is performed:

$$Q_a = Q_a + Q_{Temp}$$
$$= \begin{array}{r} 10011.111 \\ +00101.011 \\ \hline 11001.010 \end{array} = 25.250.$$

Note that overflow did not occur, but it is possible for this operation.

6 The third number to be summed (4.84375) has a fractional slope that matches the output fractional slope, so a slope adjustment is not needed. The storage data types also match, but the difference in binary points requires that both the bits and the binary point be shifted two places to the right:

$$Q_d = 100.11011,$$
$$Q_{Temp} = convert(Q_d)$$
$$Q_{Temp} = 00100.110.$$

Note that a loss in precision of two bit occurs, with the resulting value of Q_{Temp} determined by the rounding mode. For this example, round-to-floor is used. Overflow cannot occur in this case because the bits and binary point are both shifted to the right.

7 The summation operation is performed:

$$Q_a = Q_a + Q_{Temp}$$
$$= \begin{array}{r} 11001.010 \\ +00100.110 \\ \hline 11110.000 \end{array} = 30.000.$$

Note that overflow did not occur, but it is possible for this operation.

As shown here, the result of step 7 differs from the ideal sum:

$$= \begin{array}{r} 10011.111 \\ 0101.0111 \\ +100.11011 \\ \hline 11110.001 \end{array} = 30.125.$$

Blocks that perform addition and subtraction include the Sum, Gain, and Discrete FIR Filter blocks.

Multiplication

The multiplication of an n-bit binary number with an m-bit binary number results in a product that is up to m + n bits in length for both signed and unsigned words. Most processors perform n-bit by n-bit multiplication and produce a 2n-bit result (double bits) assuming there is no overflow condition.

Fixed-Point Simulink Blocks Multiplication Process

Consider the multiplication of two numbers. Ideally, the real-world values obey the equation

$$V_a = V_b V_c.$$

where V_b and V_c are the input values and V_a is the output value. To see how the multiplication is actually implemented, the three ideal values should be replaced by the general [Slope Bias] encoding scheme described in "Scaling" on page 2-5:

$$V_i = F_i 2^{E_i} Q_i + B_i.$$

The solution of the resulting equation for the output stored integer, Q_a, is given below:

$$Q_a = \frac{F_b F_c}{F_a} 2^{E_b + E_c - E_a} Q_b Q_c + \frac{F_b B_c}{F_a} 2^{E_b - E_a} Q_b + \frac{F_c B_b}{F_a} 2^{E_c - E_a} Q_c + \frac{B_b B_c - B_a}{F_a} 2^{-E_a}.$$

Multiplication with Nonzero Biases and Mismatched Fractional Slopes. The worst-case implementation of the above equation occurs when the slopes and biases of the input and output signals are mismatched. In such cases, several low-level integer operations are required to carry out the high-level multiplication (or division). Implementation choices made about these low-level computations can affect the computational efficiency, rounding errors, and overflow.

In Simulink blocks, the actual multiplication or division operation is always performed on fixed-point variables that have zero biases. If an input has nonzero bias, it is converted to a representation that has binary-point-only scaling before the operation. If the result is to have nonzero bias, the operation is first performed with temporary variables that have binary-point-only scaling. The result is then converted to the data type and scaling of the final output.

If both the inputs and the output have nonzero biases, then the operation is broken down as follows:

$$V_{1Temp} = V_1,$$
$$V_{2Temp} = V_2,$$
$$V_{3Temp} = V_{1Temp} V_{2Temp},$$
$$V_3 = V_{3Temp},$$

where

$$V_{1Temp} = 2^{E_{1Temp}} Q_{1Temp},$$
$$V_{2Temp} = 2^{E_{2Temp}} Q_{2Temp},$$
$$V_{3Temp} = 2^{E_{3Temp}} Q_{3Temp}.$$

These equations show that the temporary variables have binary-point-only scaling. However, the equations do not indicate the signedness, word lengths, or values of the fixed exponent of these variables. The Simulink software assigns these properties to the temporary variables based on the following goals:

- Represent the original value without overflow.

  The data type and scaling of the original value define a maximum and minimum real-world value:

  $$V_{Max} = F2^E Q_{MaxInteger} + B,$$

  $$V_{Min} = F2^E Q_{MinInteger} + B.$$

  The data type and scaling of the temporary value must be able to represent this range without overflow. Precision loss is possible, but overflow is never allowed.

- Use a data type that leads to efficient operations.

  This goal is relative to the target that you will use for production deployment of your design. For example, suppose that you will implement

the design on a 16-bit fixed-point processor that provides a 32-bit `long`, 16-bit `int`, and 8-bit `short` or `char`. For such a target, preserving efficiency means that no more than 32 bits are used, and the smaller sizes of 8 or 16 bits are used if they are sufficient to maintain precision.

- Maintain precision.

  Ideally, every possible value defined by the original data type and scaling is represented perfectly by the temporary variable. However, this can require more bits than is efficient. Bits are discarded, resulting in a loss of precision, to the extent required to preserve efficiency.

For example, consider the following, assuming a 16-bit microprocessor target:

$$V_{Original} = Q_{Original} + -43.25,$$

where $Q_{Original}$ is an 8-bit, unsigned data type. For this data type,

$$Q_{MaxInteger} = 225,$$
$$Q_{MinInteger} = 0,$$

so

$$V_{Max} = 211.75,$$
$$V_{Min} = -43.25.$$

The minimum possible value is negative, so the temporary variable must be a signed integer data type. The original variable has a slope of 1, but the bias is expressed with greater precision with two digits after the binary point. To get full precision, the fixed exponent of the temporary variable has to be -2 or less. The Simulink software selects the least possible precision, which is generally the most efficient, unless overflow issues arise. For a scaling of 2^{-2}, selecting signed 16-bit or signed 32-bit avoids overflow. For efficiency, the Simulink software selects the smaller choice of 16 bits. If the original variable is an input, then the equations to convert to the temporary variable are

$$\text{uint8\_T} \quad Q_{Original},$$
$$\text{uint16\_T} \quad Q_{Temp},$$
$$Q_{Temp} = \left((\text{uint16\_T})Q_{Original} \square \ 2\right) - 173.$$

Multiplication with Zero Biases and Mismatched Fractional Slopes. When the biases are zero and the fractional slopes are mismatched, the implementation reduces to

$$Q_a = \frac{F_b F_c}{F_a} 2^{E_b + E_c - E_a} Q_b Q_c.$$

Offline Conversions

The quantity

$$F_{Net} = \frac{F_b F_c}{F_a}$$

is calculated offline using round-to-nearest and saturation. F_{Net} is stored using a fixed-point data type of the form

$$2^{E_{Net}} Q_{Net},$$

where E_{Net} and Q_{Net} are selected automatically to best represent F_{Net}.

Online Conversions and Operations

1. The integer values Q_b and Q_c are multiplied:

   $$Q_{RawProduct} = Q_b Q_c.$$

   To maintain the full precision of the product, the binary point of $Q_{RawProduct}$ is given by the sum of the binary points of Q_b and Q_c.

2. The previous product is converted to the output data type:

$$Q_{Temp} = convert(Q_{RawProduct}).$$

This conversion includes any necessary bit shifting, rounding, or overflow handling. "Signal Conversions" on page 3-47 discusses conversions.

3 The multiplication

$$Q_{2RawProduct} = Q_{Temp} Q_{Net}$$

is performed.

4 The previous product is converted to the output data type:

$$Q_a = convert(Q_{2RawProduct}).$$

This conversion includes any necessary bit shifting, rounding, or overflow handling. "Signal Conversions" on page 3-47 discusses conversions.

5 Steps 1 through 4 are repeated for each additional number to be multiplied.

Multiplication with Zero Biases and Matching Fractional Slopes. When the biases are zero and the fractional slopes match, the implementation reduces to

$$Q_a = 2^{E_b + E_c - E_a} Q_b Q_c.$$

Offline Conversions

No offline conversions are performed.

Online Conversions and Operations

1 The integer values Q_b and Q_c are multiplied:

$$Q_{RawProduct} = Q_b Q_c.$$

To maintain the full precision of the product, the binary point of $Q_{RawProduct}$ is given by the sum of the binary points of Q_b and Q_c.

2. The previous product is converted to the output data type:

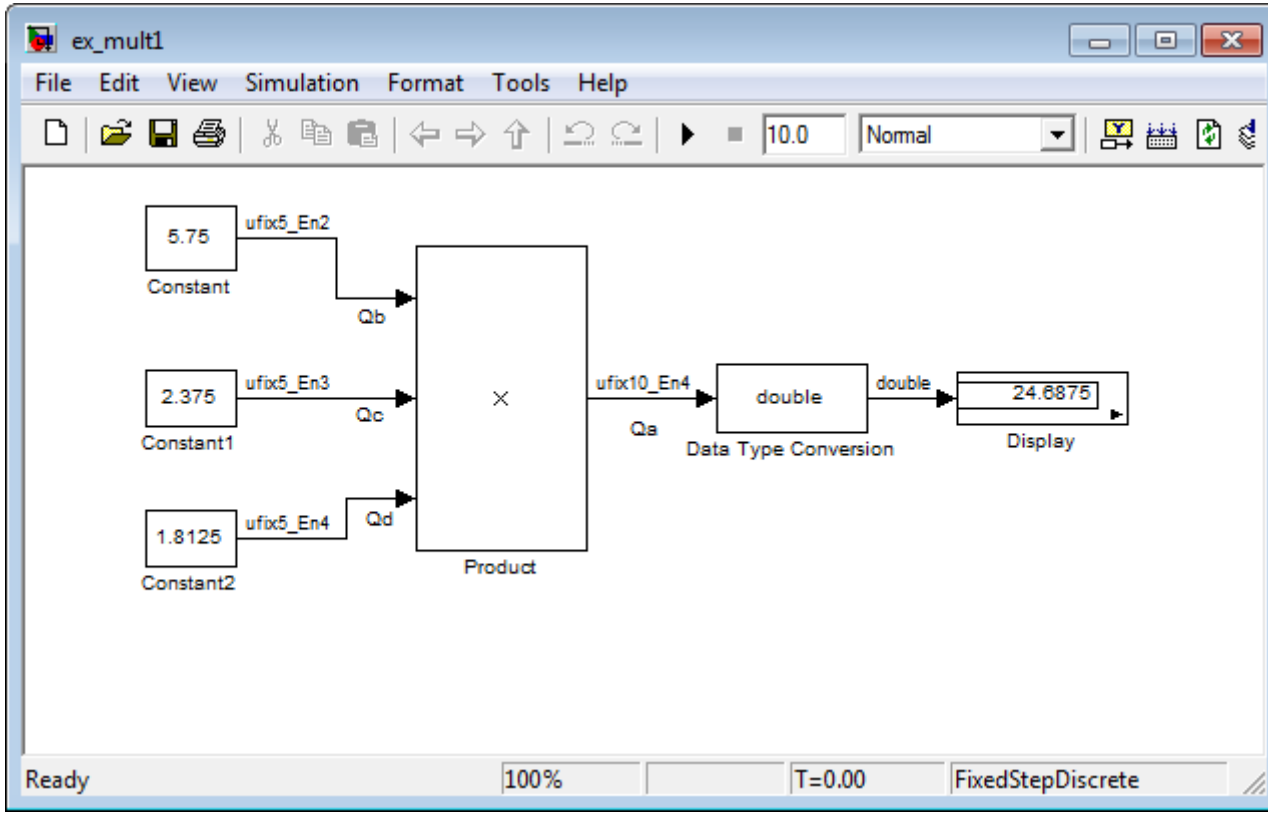
$$Q_a = convert(Q_{RawProduct}).$$

   This conversion includes any necessary bit shifting, rounding, or overflow handling. "Signal Conversions" on page 3-47 discusses conversions.

3. Steps 1 and 2 are repeated for each additional number to be multiplied.

Example: The Multiplication Process

Suppose you want to multiply three numbers. Each of these numbers is represented by a 5-bit word, and each has a different binary-point-only scaling. Additionally, the output is restricted to a 10-bit word with binary-point-only scaling of 2^{-4}. The multiplication is shown in the following model for the input values 5.75, 2.375, and 1.8125.



Applying the rules from the previous section, the multiplication follows these steps:

1 The first two numbers (5.75 and 2.375) are multiplied:

$$
\begin{array}{rl}
Q_{RawProduct} = & 101.11 \\
& \underline{\times 10.011} \\
& 101.11 \cdot 2^{-3} \\
& 101.11 \cdot 2^{-2} \\
& \underline{+101.11 \cdot 2^{1}} \\
& 01101.10101 = 13.65625.
\end{array}
$$

Note that the binary point of the product is given by the sum of the binary points of the multiplied numbers.

2 The result of step 1 is converted to the output data type:

$$\begin{aligned} Q_{Temp} &= convert\left(Q_{RawProduct}\right) \\ &= 001101.1010 = 13.6250. \end{aligned}$$

"Signal Conversions" on page 3-47 discusses conversions. Note that a loss in precision of one bit occurs, with the resulting value of Q_{Temp} determined by the rounding mode. For this example, round-to-floor is used. Furthermore, overflow did not occur but is possible for this operation.

3 The result of step 2 and the third number (1.8125) are multiplied:

$$Q_{RawProduct} = \begin{array}{r} 01101.1010 \\ \times 1.1101 \\ \hline 1101.1010 \cdot 2^{-4} \\ 1101.1010 \cdot 2^{-2} \\ 1101.1010 \cdot 2^{-1} \\ +1101.1010 \cdot 2^{0} \\ \hline 0011000.10110010 \end{array} = 24.6953125.$$

Note that the binary point of the product is given by the sum of the binary points of the multiplied numbers.

4 The product is converted to the output data type:

$$\begin{aligned} Q_a &= convert\left(Q_{RawProduct}\right) \\ &= 011000.1011 = 24.6875. \end{aligned}$$

"Signal Conversions" on page 3-47 discusses conversions. Note that a loss in precision of 4 bits occurred, with the resulting value of Q_{Temp} determined by the rounding mode. For this example, round-to-floor is used. Furthermore, overflow did not occur but is possible for this operation.

Blocks that perform multiplication include the Product, Discrete FIR Filter, and Gain blocks.

Division

This section discusses the division of quantities with zero bias.

Note When any input to a division calculation has nonzero bias, the operations performed exactly match those for multiplication described in "Multiplication with Nonzero Biases and Mismatched Fractional Slopes" on page 3-57.

Fixed-Point Simulink Blocks Division Process

Consider the division of two numbers. Ideally, the real-world values obey the equation

$$V_a = V_b / V_c ,$$

where V_b and V_c are the input values and V_a is the output value. To see how the division is actually implemented, the three ideal values should be replaced by the general [Slope Bias] encoding scheme described in "Scaling" on page 2-5:

$$V_i = F_i 2^{E_i} Q_i + B_i .$$

For the case where the slope adjustment factors are one and the biases are zero for all signals, the solution of the resulting equation for the output stored integer, Q_a, is given by the following equation:

$$Q_a = 2^{E_b - E_c - E_a} \left(Q_b / Q_c \right).$$

This equation involves an integer division and some bit shifts. If $E_a > E_b - E_c$, then any bit shifts are to the right and the implementation is simple. However, if $E_a < E_b - E_c$, then the bit shifts are to the left and the implementation can be more complicated. The essential issue is that the output has more precision

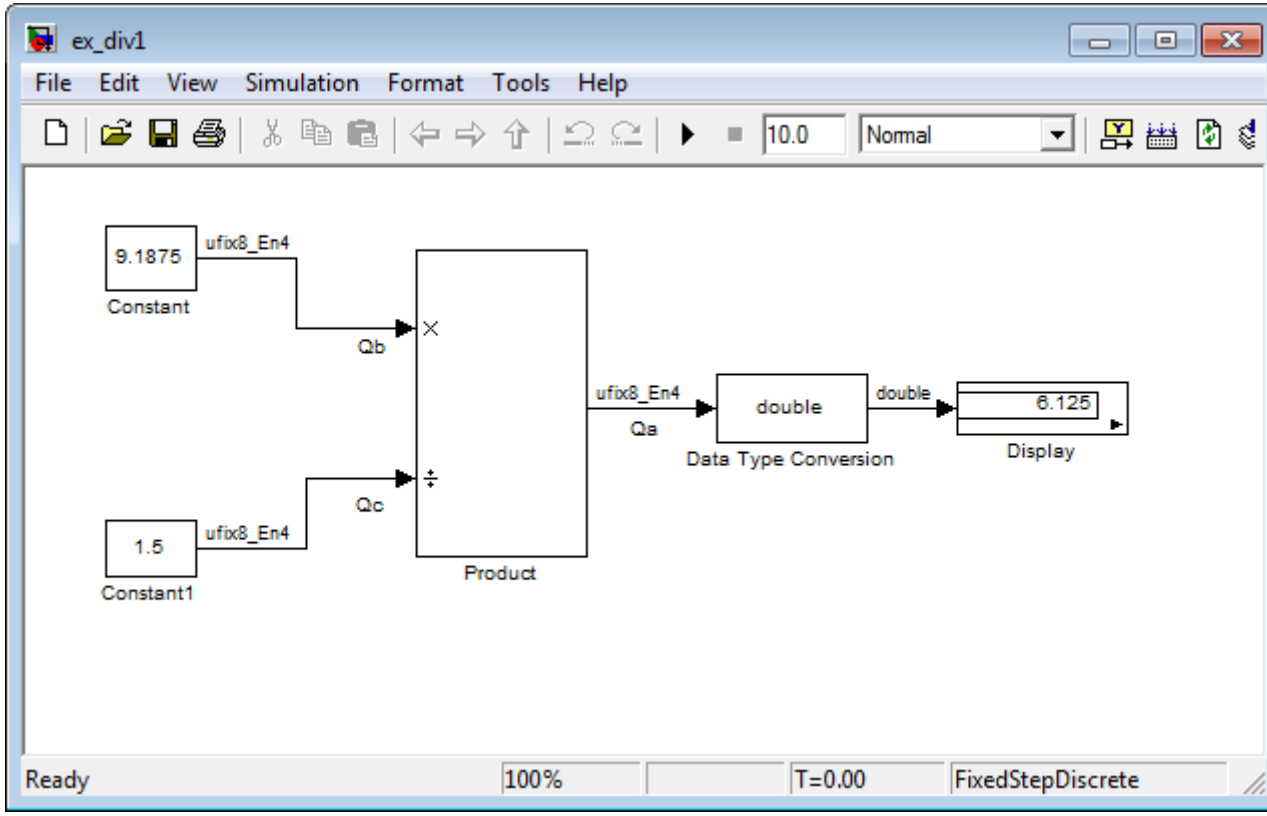
than the integer division provides. To get full precision, a *fractional* division is needed. The C programming language provides access to integer division only for fixed-point data types. Depending on the size of the numerator, you can obtain some of the fractional bits by performing a shift prior to the integer division. In the worst case, it might be necessary to resort to repeated subtractions in software.

In general, division of values is an operation that should be avoided in fixed-point embedded systems. Division where the output has more precision than the integer division (i.e., $E_a < E_b - E_c$) should be used with even greater reluctance.

Example: The Division Process

Suppose you want to divide two numbers. Each of these numbers is represented by an 8-bit word, and each has a binary-point-only scaling of 2^{-4}. Additionally, the output is restricted to an 8-bit word with binary-point-only scaling of 2^{-4}.

The division of 9.1875 by 1.5000 is shown in the following model.



For this example,

$$
\begin{aligned}
Q_a &= 2^{-4-(-4)-(-4)}\left(Q_b/Q_c\right) \\
&= 2^4\left(Q_b/Q_c\right).
\end{aligned}
$$

Assuming a large data type was available, this could be implemented as

$$
Q_a = \frac{\left(2^4 Q_b\right)}{Q_c},
$$

where the numerator uses the larger data type. If a larger data type was not available, integer division combined with four repeated subtractions would be used. Both approaches produce the same result, with the former being more efficient.

Shifts

Nearly all microprocessors and digital signal processors support well-defined *bit-shift* (or simply *shift*) operations for integers. For example, consider the 8-bit unsigned integer 00110101. The results of a 2-bit shift to the left and a 2-bit shift to the right are shown in the following table.

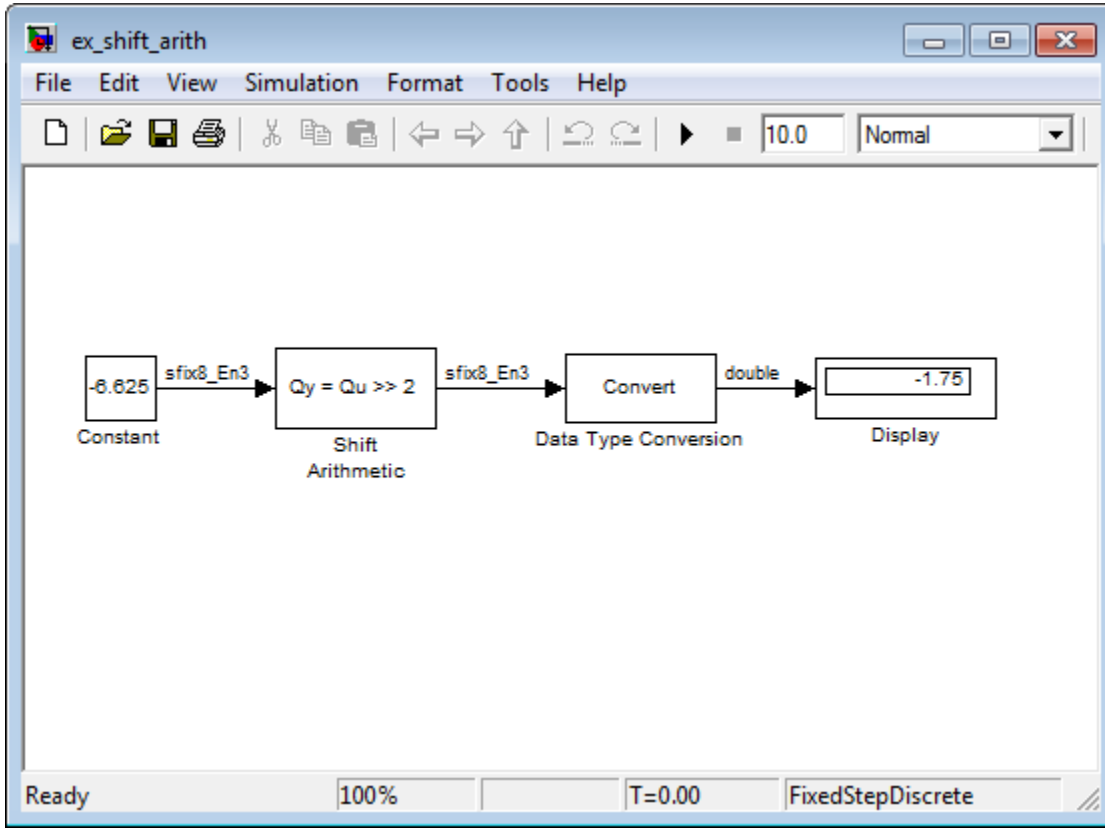
Shift Operation	Binary Value	Decimal Value
No shift (original number)	00110101	53
Shift left by 2 bits	11010100	212
Shift right by 2 bits	00001101	13

You can perform a shift using the Simulink Shift Arithmetic block. Use this block to perform a bit shift, a binary point shift, or both. See the documentation for the Shift Arithmetic block in the *Simulink Reference* for more information on performing bit and binary point shifts.

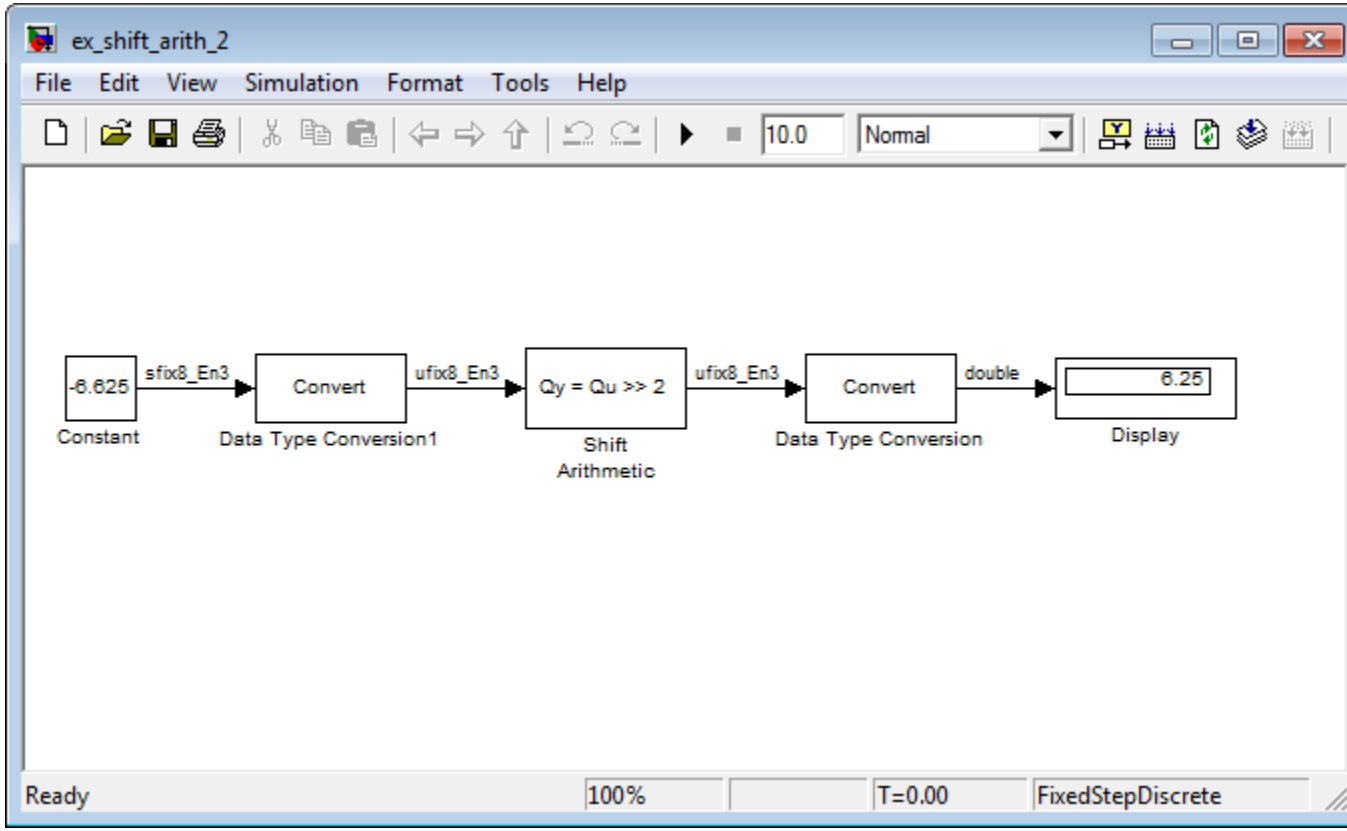
Shifting Bits to the Right

The special case of shifting bits to the right requires consideration of the treatment of the leftmost bit, which can contain sign information. A shift to the right can be classified either as a *logical* shift right or an *arithmetic* shift right. For a logical shift right, a 0 is incorporated into the most significant bit for each bit shift. For an arithmetic shift right, the most significant bit is recycled for each bit shift.

The Shift Arithmetic block performs an arithmetic shift right and, therefore, recycles the most significant bit for each bit shift right. For example, given the fixed-point number 11001.011 (-6.625), a bit shift two places to the right with the binary point unmoved yields the number 11110.010 (-1.75), as shown in the model below:



To perform a logical shift right on a signed number using the Shift Arithmetic block, use the Data Type Conversion block to cast the number as an unsigned number of equivalent length and scaling, as shown in the following model. The model shows that the fixed-point signed number 11001.001 (-6.625) becomes 00110.010 (6.25).


ex_shift_arith_2
File Edit View Simulation Format Tools Help
10.0
Normal
-6.625
Constant
sfix8_En3
Convert
Data Type Conversion1
ufix8_En3
Qy = Qu >> 2
Shift
Arithmetic
ufix8_En3
Convert
Data Type Conversion
double
6.25
Display
Ready
100%
T=0.00
FixedStepDiscrete

Example: Conversions and Arithmetic Operations

This example uses the Discrete FIR Filter block to illustrate when parameters are converted from a double to a fixed-point number, when the input data type is converted to the output data type, and when the rules for addition, subtraction, and multiplication are applied. For details about conversions and operations, refer to "Parameter and Signal Conversions" on page 3-45 and "Rules for Arithmetic Operations" on page 3-50.

Note If a block can perform all four arithmetic operations, then the rules for multiplication and division are applied first. The Discrete FIR Filter block is an example of this.

Suppose you configure the Discrete FIR Filter block for two outputs, where the first output is given by

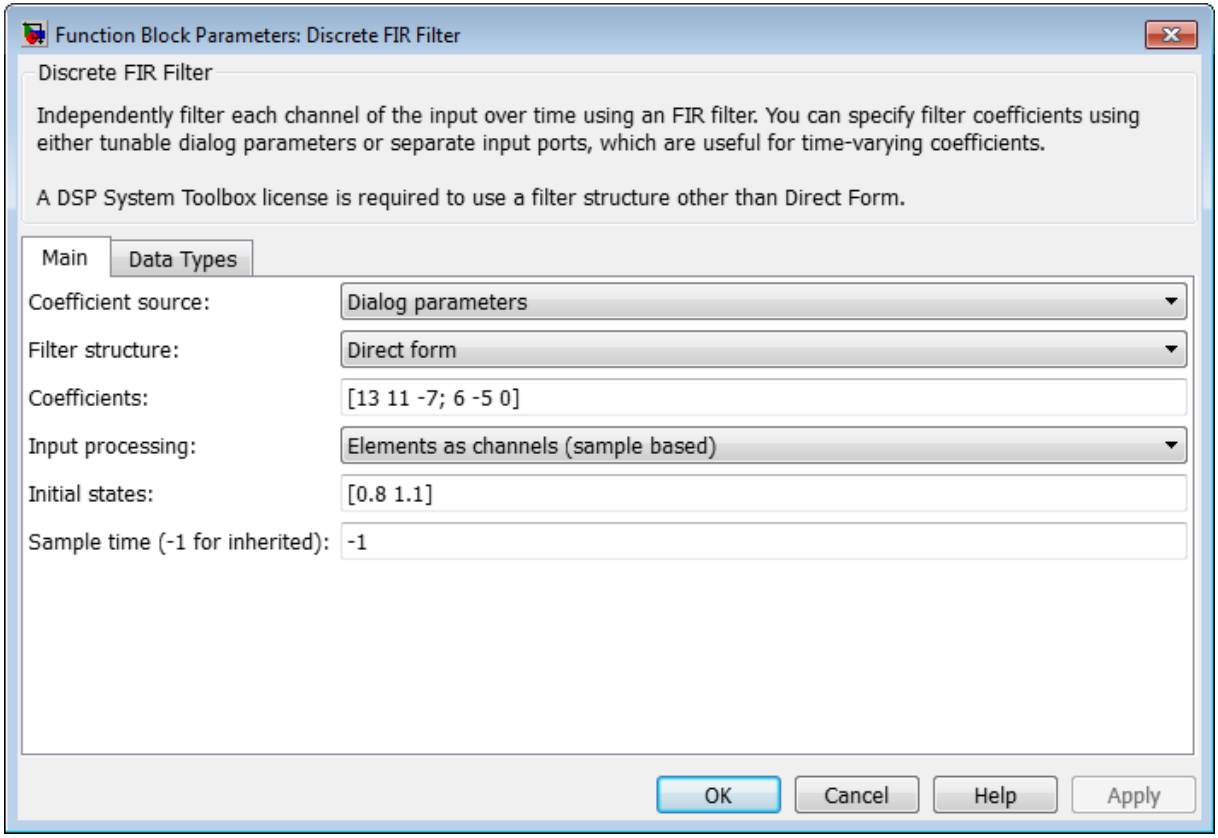
$$y_1(k) = 13 \cdot u(k) + 11 \cdot u(k-1) - 7 \cdot u(k-2),$$

and the second output is given by

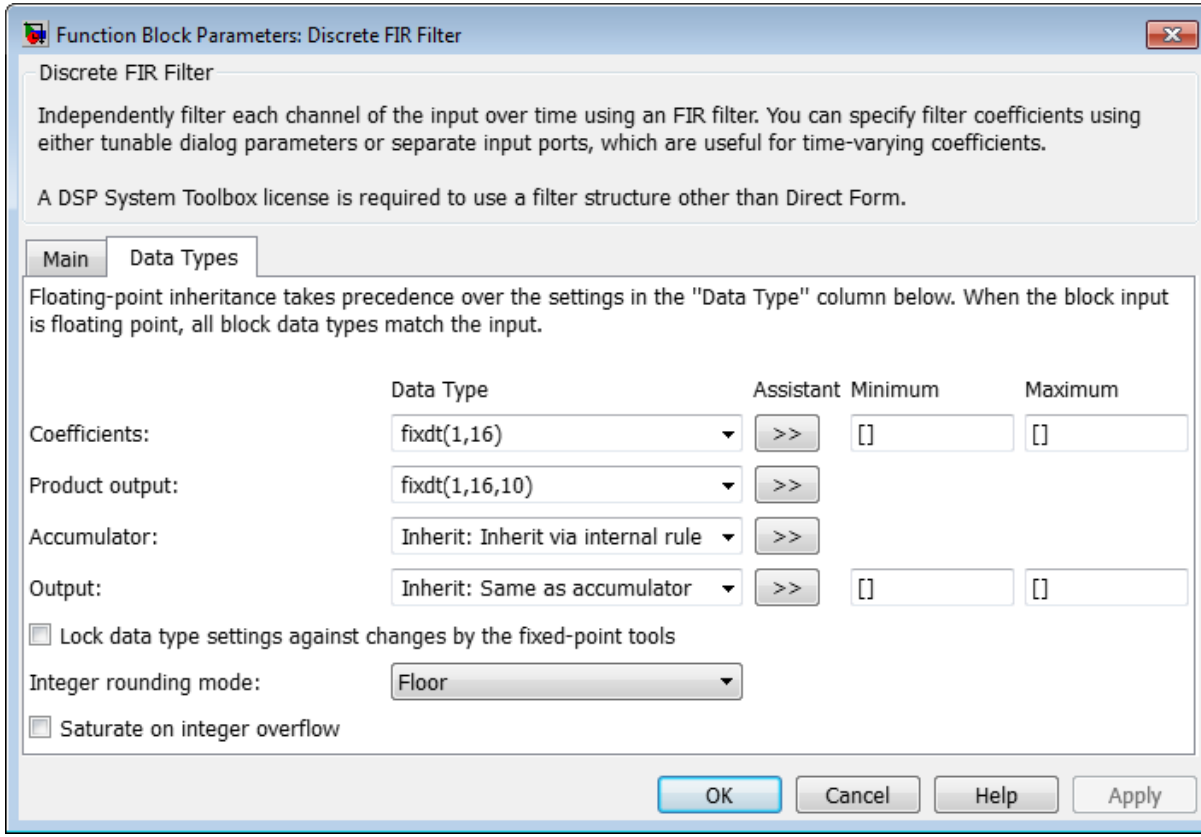
$$y_2(k) = 6 \cdot u(k) - 5 \cdot u(k-1).$$

Additionally, the initial values of $u(k-1)$ and $u(k-2)$ are given by 0.8 and 1.1, respectively, and all inputs, parameters, and outputs have binary-point-only scaling.

To configure the Discrete FIR Filter block for this situation, on the **Main** pane of its dialog box, you must specify the **Coefficients** parameter as `[13 11 -7; 6 -5 0]` and the **Initial states** parameter as `[0.8 1.1]`, as shown here.



Similarly, configure the options on the **Data Types** pane of the block dialog box to appear as follows:



The Discrete FIR Filter block performs parameter conversions and block operations in the following order:

1. The **Coefficients** parameter is converted offline from doubles to the **Coefficients** data type using round-to-nearest and saturation.

   The **Initial states** parameter is converted offline from doubles to the input data type using round-to-nearest and saturation.

2. The coefficients and inputs are multiplied together for the initial time step for both outputs. For $y_1(0)$, the operations $13 \cdot u(0)$, $11 \cdot 0.8$, and $-7 \cdot 1.1$ are performed, while for $y_2(0)$, the operations $6 \cdot u(0)$ and $-5 \cdot 0.8$ are performed.

The results of these operations are stored as **Product output**.

3. The sum is carried out in **Accumulator**. The final summation result is then converted to **Output**.

4. Steps 2 and 3 repeat for subsequent time steps.

4

Realization Structures

Overview

In this section...

Introduction

This chapter investigates how you can realize fixed-point digital filters using Simulink blocks and the Simulink Fixed Point software.

The Simulink Fixed Point software addresses the needs of the control system, signal processing, and other fields where algorithms are implemented on fixed-point hardware. In signal processing, a digital filter is a computational algorithm that converts a sequence of input numbers to a sequence of output numbers. The algorithm is designed such that the output signal meets frequency-domain or time-domain constraints (desirable frequency components are passed, undesirable components are rejected).

In general terms, a discrete transfer function controller is a form of a digital filter. However, a digital controller can contain nonlinear functions such as lookup tables in addition to a discrete transfer function. This guide uses the term *digital filter* when referring to discrete transfer functions.

Note To design and implement a wide variety of floating-point and fixed-point filters suitable for use in signal processing applications and for deployment on DSP chips, use the DSP System Toolbox software.

Realizations and Data Types

In an ideal world, where numbers, calculations, and storage of states have infinite precision and range, there are virtually an infinite number of realizations for the same system. In theory, these realizations are all identical.

In the more realistic world of double-precision numbers, calculations, and storage of states, small nonlinearities are introduced by the finite precision and range of floating-point data types. Therefore, each realization of a given

system produces different results. In most cases however, these differences are small.

In the world of fixed-point numbers, where precision and range are limited, the differences in the realization results can be very large. Therefore, you must carefully select the data type, word size, and scaling for each realization element such that results are accurately represented. To assist you with this selection, design rules for modeling dynamic systems with fixed-point math are provided in “Targeting an Embedded Processor” on page 4-4.

Targeting an Embedded Processor

In this section...

Introduction

The sections that follow describe issues that often arise when targeting a fixed-point design for use on an embedded processor, such as some general assumptions about integer sizes and operations available on embedded processors. These assumptions lead to design issues and design rules that might be useful for your specific fixed-point design.

Size Assumptions

Embedded processors are typically characterized by a particular bit size. For example, the terms "8-bit micro," "32-bit micro," or "16-bit DSP" are common. It is generally safe to assume that the processor is predominantly geared to processing integers of the specified bit size. Integers of the specified bit size are referred to as the *base data type*. Additionally, the processor typically provides some support for integers that are twice as wide as the base data type. Integers consisting of double bits are referred to as the *accumulator data type*. For example a 16-bit micro has a 16-bit base data type and a 32-bit accumulator data type.

Although other data types may be supported by the embedded processor, this section describes only the base and accumulator data types.

Operation Assumptions

The embedded processor operations discussed in this section are limited to the needs of a basic simulation diagram. Basic simulations use multiplication, addition, subtraction, and delays. Fixed-point models also need shifts to do scaling conversions. For all these operations, the embedded processor

should have native instructions that allow the base data type as inputs. For accumulator-type inputs, the processor typically supports addition, subtraction, and delay (storage/retrieval from memory), but not multiplication.

Multiplication is typically not supported for accumulator-type inputs because of complexity and size issues. A difficulty with multiplication is that the output needs to be twice as big as the inputs for full precision. For example, multiplying two 16-bit numbers requires a 32-bit output for full precision. The need to handle the outputs from a multiplication operation is one of the reasons embedded processors include accumulator-type support. However, if multiplication of accumulator-type inputs is also supported, then there is a need to support a data type that is twice as big as the accumulator type. To restrict this additional complexity, multiplication is typically not supported for inputs of the accumulator type.

Design Rules

The important design rules that you should be aware of when modeling dynamic systems with fixed-point math follow.

Design Rule 1: Only Multiply Base Data Types

It is best to multiply only inputs of the base data type. Embedded processors typically provide an instruction for the multiplication of base-type inputs, but not for the multiplication of accumulator-type inputs. If necessary, you can combine several instructions to handle multiplication of accumulator-type inputs. However, this can lead to large, slow embedded code.

You can insert blocks to convert inputs from the accumulator type to the base type prior to Product or Gain blocks, if necessary.

Design Rule 2: Delays Should Use the Base Data Type

There are two general reasons why a Unit Delay should use only base-type numbers:

- The Unit Delay essentially stores a variable's value to RAM and, one time step later, retrieves that value from RAM. Because the value must be in memory from one time step to the next, the RAM must be exclusively dedicated to the variable and can't be shared or used for another purpose.

Using accumulator-type numbers instead of the base data type doubles the RAM requirements, which can significantly increase the cost of the embedded system.

- The Unit Delay typically feeds into a Gain block. The multiplication design rule requires that the input (the unit delay signal) use the base data type.

Design Rule 3: Temporary Variables Can Use the Accumulator Data Type

Except for unit delay signals, most signals are not needed from one time step to the next. This means that the signal values can be temporarily stored in shared and reused memory. This shared and reused memory can be RAM or it can simply be registers in the CPU. In either case, storing the value as an accumulator data type is not much more costly than storing it as a base data type.

Design Rule 4: Summation Can Use the Accumulator Data Type

Addition and subtraction can use the accumulator data type if there is justification. The typical justification is reducing the buildup of errors due to roundoff or overflow.

For example, a common filter operation is a weighted sum of several variables. Multiplying a variable by a weight naturally produces a product of the accumulator type. Before summing, each product can be converted back to the base data type. This approach introduces round-off error into each part of the sum.

Alternatively, the products can be summed using the accumulator data type, and the final sum can be converted to the base data type. Round-off error is introduced in just one point and the precision is generally better. The cost of doing an addition or subtraction using accumulator-type numbers is slightly more expensive, but if there is justification, it is usually worth the cost.

Canonical Forms

In this section...

Introduction

The Simulink Fixed Point software does not attempt to standardize on one particular fixed-point digital filter design method. For example, you can produce a design in continuous time and then obtain an "equivalent" discrete-time digital filter using one of many transformation methods. Alternatively, you can design digital filters directly in discrete time. After you obtain a digital filter, it can be realized for fixed-point hardware using any number of canonical forms. Typical canonical forms are the direct form, series form, and parallel form, each of which is outlined in the sections that follow.

For a given digital filter, the canonical forms describe a set of fundamental operations for the processor. Because there are an infinite number of ways to realize a given digital filter, you must make the best realization on a per-system basis. The canonical forms presented in this chapter optimize the implementation with respect to some factor, such as minimum number of delay elements.

In general, when choosing a realization method, you must take these factors into consideration:

- **Cost**

  The cost of the realization might rely on minimal code and data size.

- **Timing constraints**

  Real-time systems must complete their compute cycle within a fixed amount of time. Some realizations might yield faster execution speed on different processors.

- **Output signal quality**

  The limited range and precision of the binary words used to represent real-world numbers will introduce errors. Some realizations are more sensitive to these errors than others.

The Simulink Fixed Point software allows you to evaluate various digital filter realization methods in a simulation environment. Following the development cycle outlined in "The Development Cycle" on page 1-18, you can fine-tune the realizations with the goal of reducing the cost (code and data size) or increasing signal quality. After you have achieved the desired performance, you can use the Simulink Coder product to generate rapid prototyping C code and evaluate its performance with respect to your system's real-time timing constraints. You can then modify the model based upon feedback from the rapid prototyping system.

The presentation of the various realization structures takes into account that a summing junction is a fundamental operator, thus you may find that the structures presented here look different from those in the fixed-point filter design literature. For each realization form, an example is provided using the transfer function shown here:

$$
\begin{aligned}
H_{ex}(z) &= \frac{1+2.2z^{-1}+1.85z^{-2}+0.5z^{-3}}{1-0.5z^{-1}+0.84z^{-2}+0.09z^{-3}} \\
&= \frac{\left(1+0.5z^{-1}\right)\left(1+1.7z^{-1}+z^{-2}\right)}{\left(1+0.1z^{-1}\right)\left(1-0.6z^{-1}+0.9z^{-2}\right)} \\
&= 5.5556 - \frac{3.4639}{1+0.1z^{-1}} + \frac{-1.0916+3.0086z^{-1}}{1-0.6z^{-1}+0.9z^{-2}}.
\end{aligned}
$$

Direct Form II

In general, a direct form realization refers to a structure where the coefficients of the transfer function appear directly as Gain blocks. The direct form II realization method is presented as using the minimal number of delay elements, which is equal to n, the order of the transfer function denominator.

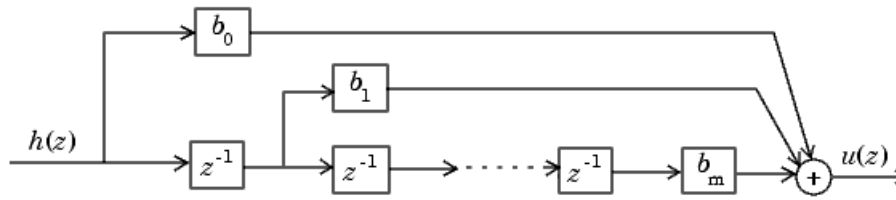
The canonical direct form II is presented as "Standard Programming" in *Discrete-Time Control Systems* by Ogata. It is known as the "Control

Canonical Form" in *Digital Control of Dynamic Systems* by Franklin, Powell, and Workman.

You can derive the canonical direct form II realization by writing the discrete-time transfer function with input $e(z)$ and output $u(z)$ as

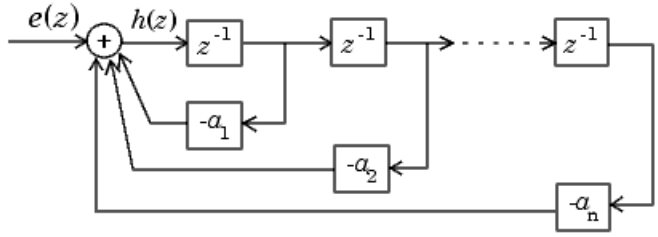
$$\begin{aligned}\frac{u(z)}{e(z)} &= \frac{u(z)}{h(z)} \cdot \frac{h(z)}{e(z)} \\ &= \underbrace{\left(b_0 + b_1 z^{-1} + \ldots + b_m z^{-m}\right)}_{\frac{u(z)}{h(z)}} \underbrace{\frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} \ldots + a_n z^{-n}}}_{\frac{h(z)}{e(z)}}.\end{aligned}$$

The block diagram for $u(z)/h(z)$ follows.



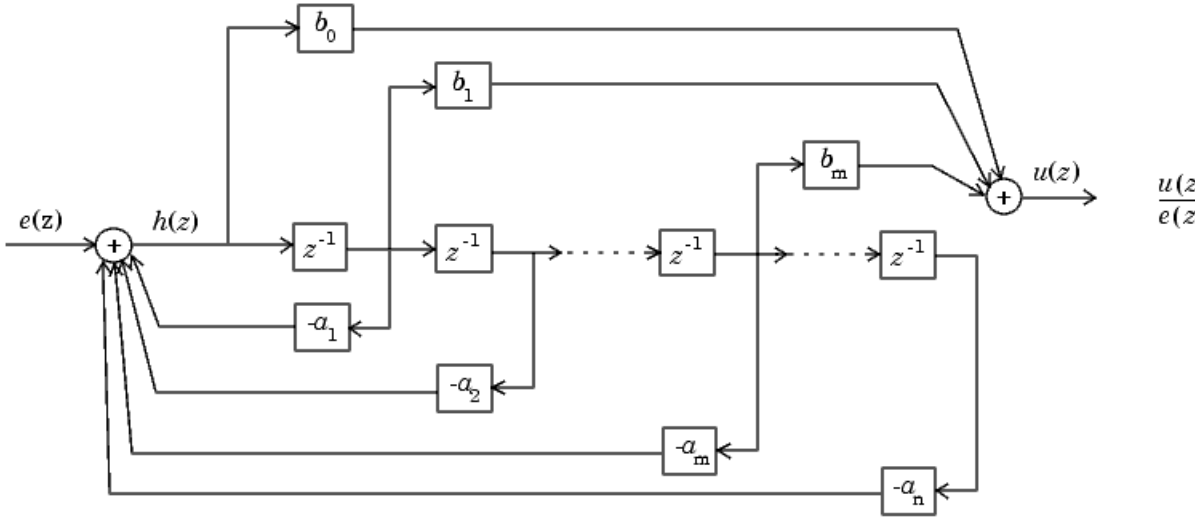
$$\frac{u(z)}{h(z)} = b_0 + b_1 z^{-1} + \ldots + b_m z^{-m}$$

The block diagrams for $h(z)/e(z)$ follow.



$$\frac{h(z)}{e(z)} = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_n z^{-n}}$$

Combining these two block diagrams yields the direct form II diagram shown in the following figure. Notice that the feedforward part (top of block diagram) contains the numerator coefficients and the feedback part (bottom of block diagram) contains the denominator coefficients.



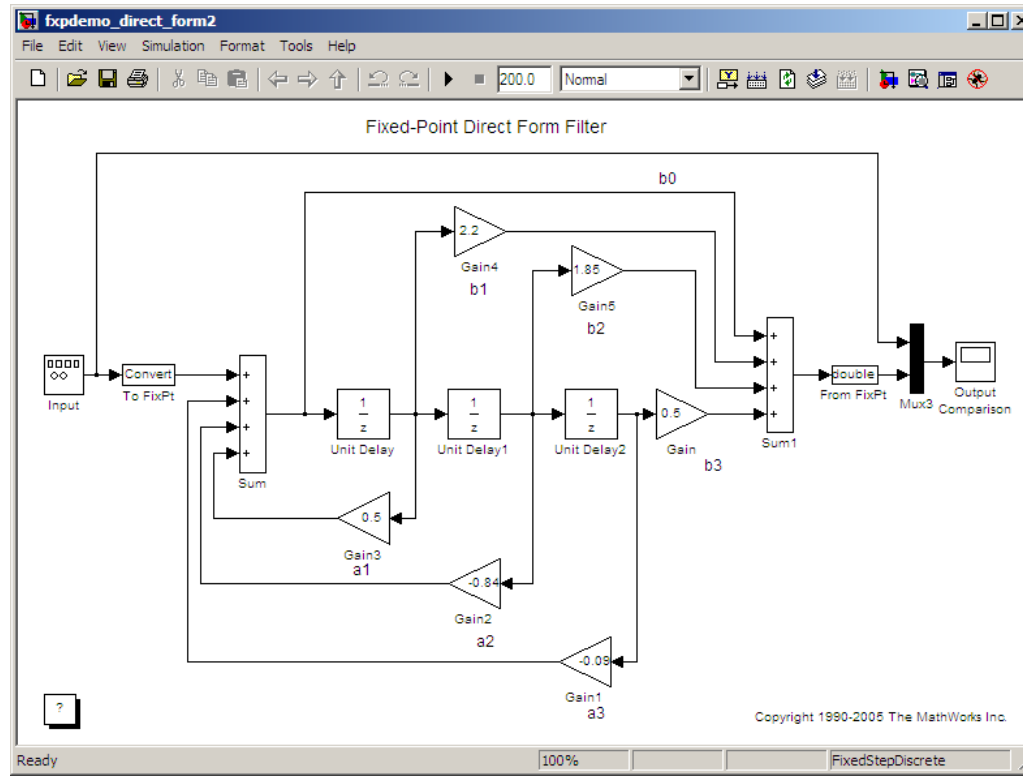
The direct form II example transfer function is given by

$$H_{ex}(z) = \frac{1 + 2.2z^{-1} + 1.85z^{-2} + 0.5z^{-3}}{1 - 0.5z^{-1} + 0.84z^{-2} + 0.09z^{-3}}.$$

The realization of $H_{ex}(z)$ using fixed-point Simulink blocks is shown in the following figure. You can display this model by typing

```
fxpdemo_direct_form2
```

at the MATLAB command line.

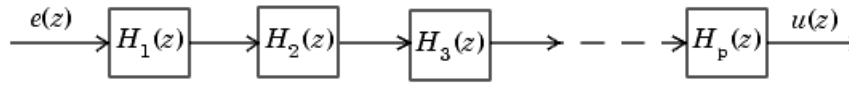


Series Cascade Form

In the canonical series cascade form, the transfer function $H(z)$ is written as a product of first-order and second-order transfer functions:

$$H_i(z) = \frac{u(z)}{e(z)} = H_1(z) \cdot H_2(z) \cdot H_3(z) \ldots H_p(z).$$

This equation yields the canonical series cascade form.



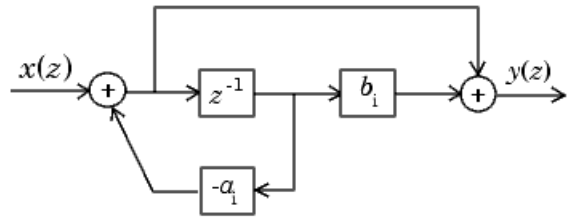
Factoring $H(z)$ into $H_i(z)$ where i = 1,2,3,...,p can be done in a number of ways. Using the poles and zeros of $H(z)$, you can obtain $H_i(z)$ by grouping pairs of conjugate complex poles and pairs of conjugate complex zeros to produce second-order transfer functions, or by grouping real poles and real zeros to produce either first-order or second-order transfer functions. You could also group two real zeros with a pair of conjugate complex poles or vice versa. Since there are many ways to obtain $H_i(z)$, you should compare the various groupings to see which produces the best results for the transfer function under consideration.

For example, one factorization of $H(z)$ might be

$$\begin{aligned} H(z) &= H_1(z) H_2(z) \ldots H_p(z) \\ &= \prod_{i=1}^{j} \frac{1 + b_i z^{-1}}{1 + a_i z^{-1}} \prod_{i=j+1}^{p} \frac{1 + e_i z^{-1} + f_i z^{-2}}{1 + c_i z^{-1} + d_i z^{-2}}. \end{aligned}$$

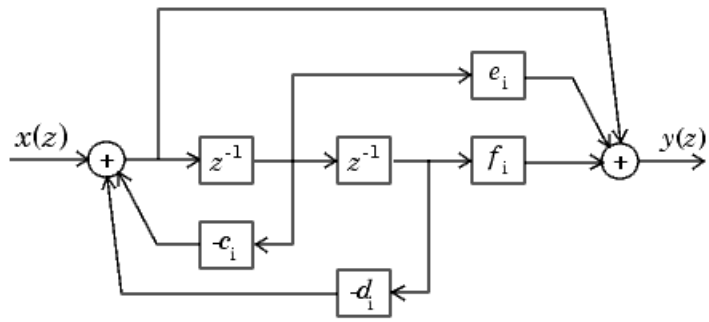
You must also take into consideration that the ordering of the individual $H_i(z)$'s will lead to systems with different numerical characteristics. You might want to try various orderings for a given set of $H_i(z)$'s to determine which gives the best numerical characteristics.

The first-order diagram for $H(z)$ follows.



$$\frac{y(z)}{x(z)} = \frac{1 + b_i z^{-1}}{1 + a_i z^{-1}}$$

The second-order diagram for $H(z)$ follows.



$$\frac{y(z)}{x(z)} = \frac{1 + e_i z^{-1} + f_i z^{-2}}{1 + c_i z^{-1} + d_i z^{-2}}$$

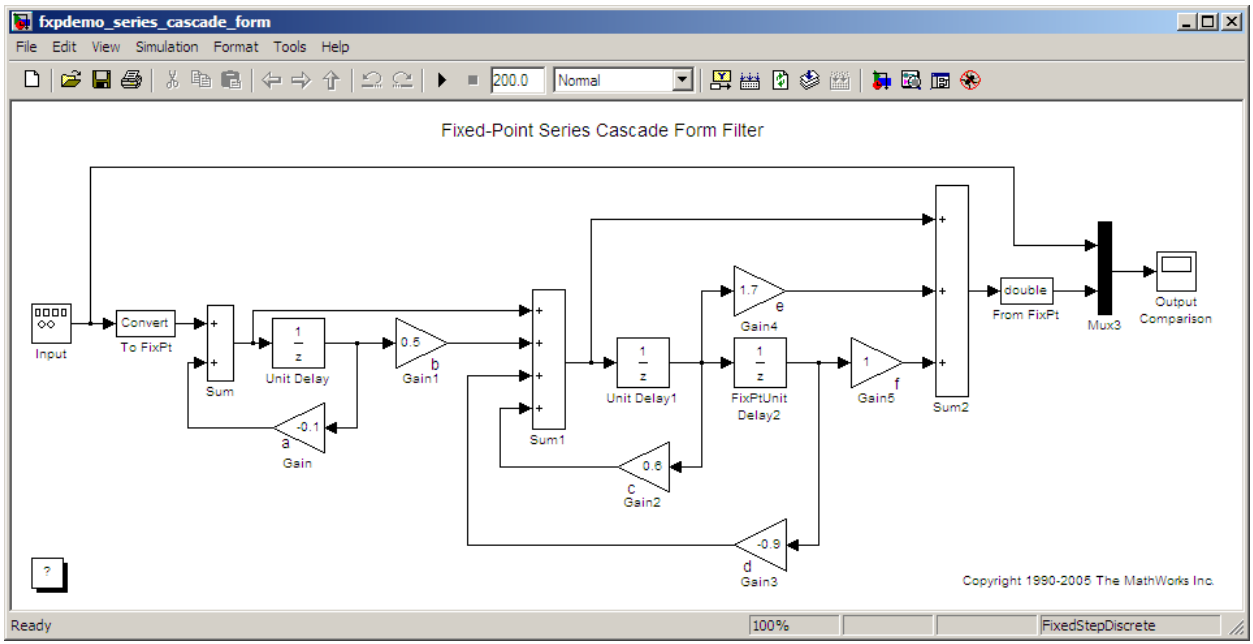
The series cascade form example transfer function is given by

$$H_{ex}(z) = \frac{(1+0.5z^{-1})(1+1.7z^{-1}+z^{-2})}{(1+0.1z^{-1})(1-0.6z^{-1}+0.9z^{-2})}.$$

The realization of $H_{ex}(z)$ using fixed-point Simulink blocks is shown in the following figure. You can display this model by typing

```
fxpdemo_series_cascade_form
```

at the MATLAB command line.

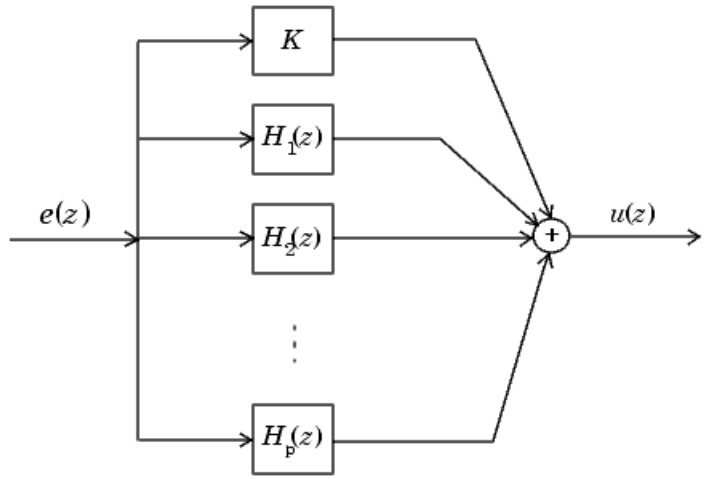


Parallel Form

In the canonical parallel form, the transfer function $H(z)$ is expanded into partial fractions. $H(z)$ is then realized as a sum of a constant, first-order, and second-order transfer functions, as shown:

$$H_i(z) = \frac{u(z)}{e(z)} = K + H_1(z) + H_2(z) + \ldots + H_p(z).$$

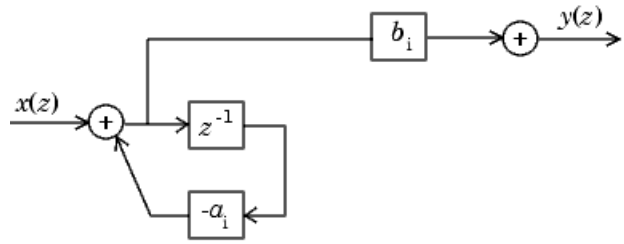
This expansion, where K is a constant and the $H_i(z)$ are the first- and second-order transfer functions, follows.



As in the series canonical form, there is no unique description for the first-order and second-order transfer function. Because of the nature of the Sum block, the ordering of the individual filters doesn't matter. However, because of the constant K, you can choose the first-order and second-order transfer functions such that their forms are simpler than those for the series cascade form described in the preceding section. This is done by expanding $H(z)$ as

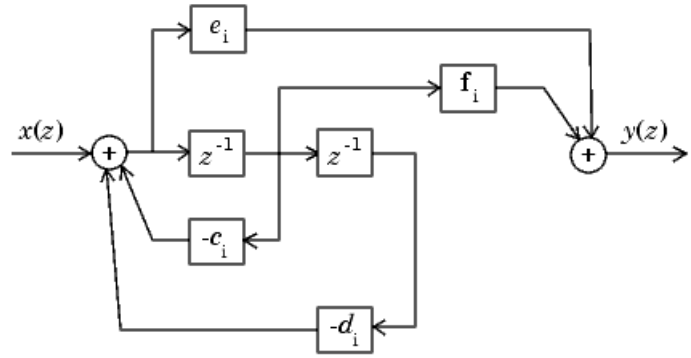
$$\begin{aligned} H(z) &= K + \sum_{i=1}^{j} H_i(z) + \sum_{i=j+1}^{p} H_i(z) \\ &= K + \sum_{i=1}^{j} \frac{b_i}{1 + a_i z^{-1}} + \sum_{i=j+1}^{p} \frac{e_i + f_i z^{-1}}{1 + c_i z^{-1} + d_i z^{-2}}. \end{aligned}$$

The first-order diagram for $H(z)$ follows.



$$\frac{y(z)}{x(z)} = \frac{b_i}{1 + a_i z^{-1}}$$

The second-order diagram for $H(z)$ follows.



$$\frac{y(z)}{x(z)} = \frac{e_i + f_i z^{-1}}{1 + c_i z^{-1} + d_i z^{-2}}$$

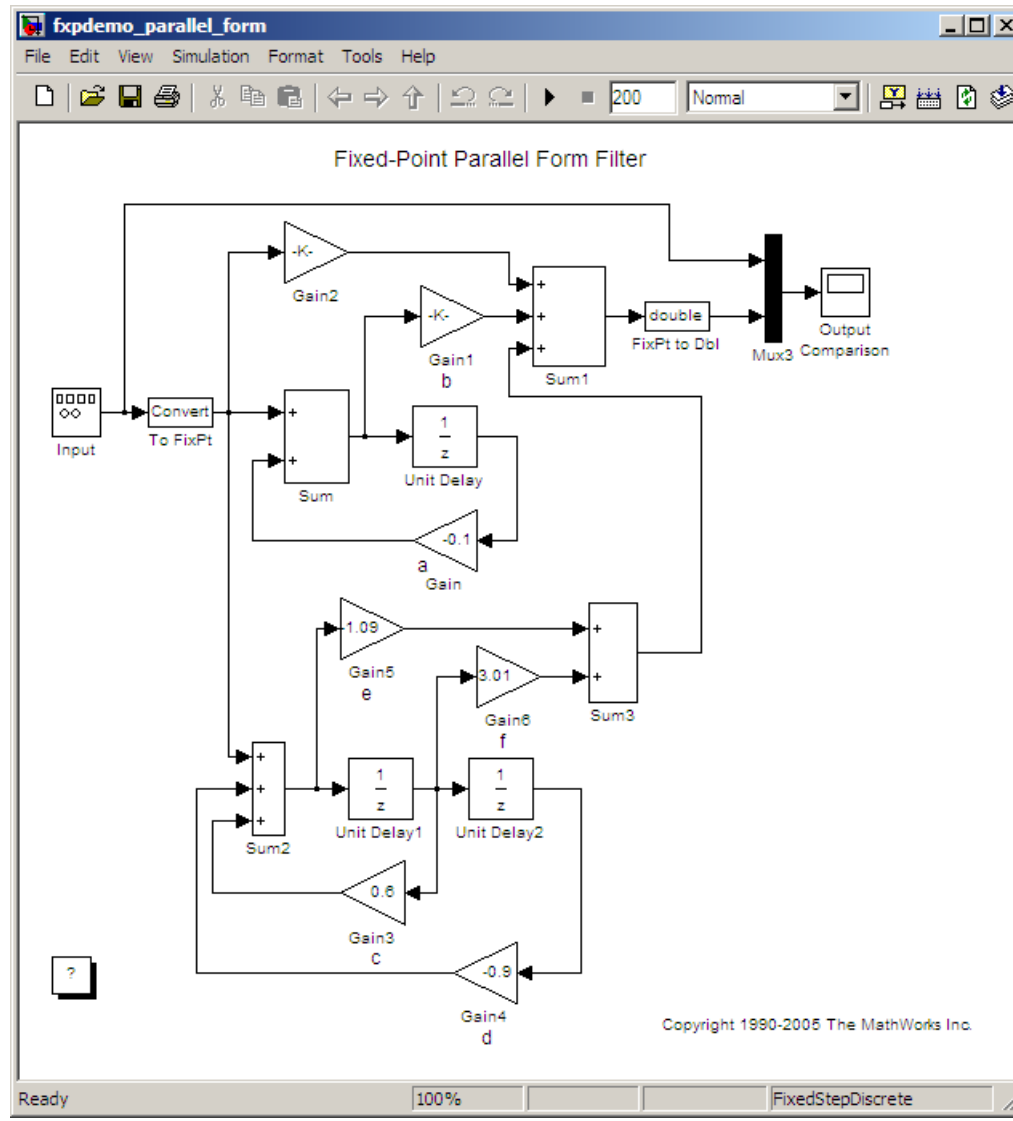
The parallel form example transfer function is given by

$$H_{ex}(z) = 5.5556 - \frac{3.4639}{1 + 0.1z^{-1}} + \frac{-1.0916 + 3.0086z^{-1}}{1 - 0.6z^{-1} + 0.9z^{-2}}.$$

The realization of $H_{ex}(z)$ using fixed-point Simulink blocks is shown in the following figure. You can display this model by typing

```
fxpdemo_parallel_form
```

at the MATLAB command line.


fxpdemo_parallel_form
File Edit View Simulation Format Tools Help
200
Normal
Fixed-Point Parallel Form Filter
-K-
Gain2
-K-
Gain1
b
double
FixPt to Dbl
Sum1
Mux3
Output
Comparison
Input
Convert
To FixPt
Sum
1
z
Unit Delay
-0.1
a
Gain
1.09
Gain5
e
3.01
Gain6
f
Sum3
Sum2
Unit Delay1
Unit Delay2
0.6
Gain3
c
-0.9
Gain4
d
?
Copyright 1990-2005 The MathWorks Inc.
Ready
100%
FixedStepDiscrete

5

Fixed-Point Advisor

Working with the Fixed-Point Advisor

In this section...

Introduction to the Fixed-Point Advisor

The Fixed-Point Advisor provides a set of tasks to help you prepare a model for conversion from a floating-point model or subsystem to an equivalent fixed-point representation. After preparing the model for conversion, use the Fixed-Point Tool to obtain initial scaling and then refine the scaling.

Best Practices for Using the Fixed-Point Advisor

Start with a Known Working Model

Check that update diagram succeeds for your model before using the Fixed-Point Advisor. (To update diagram, press **Ctrl+D**.) If update diagram fails, fix the failure in your model before you start converting your model.

Back Up Your Model

Always back up your Simulink model before using the Fixed-Point Advisor.

This practice provides a fallback in case of error and a baseline for testing and validation.

Convert Small Models

The Fixed-Point Advisor is intended to assist in converting small models. Larger models can result in long processing times.

Convert Subsystems

Convert subsystems within your model, rather than the entire model. This practice saves time and unnecessary conversions.

Specify Short Simulation Run Times

Specifying small simulation run times reduces task processing times. You can change the simulation run time in the Configuration Parameters dialog box. See "Start time" and "Stop time" in the *Simulink Reference* for more information.

Make Small Changes to Your Model

Make small changes to your model so that you can identify where errors are introduced accidentally.

Isolate the System Under Conversion

If you encounter data type propagation issues with a particular subsystem, isolate this subsystem by placing Data Type Conversion blocks on the inputs and outputs of the system. The Data Type Conversion block converts an input signal of any Simulink software data type to the data type and scaling you specify for its **Output data type** parameter. This practice enables you to continue converting the rest of your model.

The ultimate goal is to replace all blocks that do not support fixed-point data types. You must eventually replace blocks that you isolated with Data Type Conversion blocks with blocks that do support fixed-point data types.

Use Lock Output Data Type Setting When Necessary

You can prevent the Fixed-Point Advisor replacing the current data type. Use the **Lock output data type setting against changes by the fixed-point tools** parameter available on many blocks. The default setting allows replacement. Use this setting when:

- You already know the fixed-point data types you want to use for a particular block.

  For example, the block is modeling a real-world component. Set up the block to allow for known hardware limitations, such as restricting outputs to integer values.

  Specify the output data type of the block explicitly and select **Lock output data type setting against changes by the fixed-point tools**.

- You are debugging a model and know that a particular block accepts only certain data types.

  Specify the output data type of upstream blocks explicitly and select **Lock output data type setting against changes by the fixed-point tools**.

Save Simulink Signal Objects Before Closing Your Model

The Fixed-Point Advisor proposes data types for Simulink signal objects in your model. However, it does not automatically save Simulink signal objects. To preserve changes, save the Simulink signal objects in your workspace and model before closing the model.

Save Restore Point

Consider saving a restore point before making changes to your model that might cause subsequent update diagram failure. For example, before applying proposed data types in task 3.1. For more information, see "Saving a Restore Point" on page 5-11.

Models That Might Cause Data Type Propagation Errors

The Fixed-Point Advisor is not aware of all potential scaling issues and might propose data types that cause subsequent data propagation errors. To ensure that you can recover your original data type settings, back up your model

before using the Fixed-Point Advisor. For more information about backing up your model and other best practices for using the Fixed-Point Advisor, see "Best Practices for Using the Fixed-Point Advisor" on page 5-2.

The following models are likely to cause data type propagation issues:

Model Uses...	Fixed-Point Advisor Behavior	Data Type Propagation Issue
Buses	Not aware of the minimum, maximum, data type, and initial value information for bus objects and does not include them in autoscaling.	Fixed-Point Advisor might propose data types that are inconsistent with the data types for the bus object.
Simulink parameter objects	Does not consider any data type information for Simulink parameter objects.	Fixed-Point Advisor might propose data types that are inconsistent with the data types for the parameter object.

Model Uses...	Fixed-Point Advisor Behavior	Data Type Propagation Issue
User-defined S-functions	Not aware of the operation of user-defined S-functions.	• The user-defined S-function only accepts certain input data types. The Fixed-Point Advisor is not aware of this requirement and proposes a different data type upstream of the S-function. Update diagram fails on the model due to data type mismatch errors. • The user-defined S-function specifies certain output data types. The Fixed-Point Advisor is not aware of this requirement and does not use it when autoscaling. Therefore it might propose data types that are inconsistent with the data types for the S-function.
User-defined masked subsystems	Has no knowledge of the masked subsystem workspace and therefore cannot take this into account when autoscaling.	Fixed-Point Advisor might propose data types that are inconsistent with the requirements of the masked subsystem, particularly if the subsystem uses mask initialization. The proposed data types might cause data type mismatch errors or overflows.

Model Uses...	Fixed-Point Advisor Behavior	Data Type Propagation Issue
Linked subsystems	Does not include linked subsystems when converting your model.	Data type mismatch errors might occur at the linked subsystem boundaries.
MATLAB Function blocks	Does not propose data types for MATLAB Function blocks.	Fixed-Point Advisor might propose data types that are inconsistent with the requirements of the MATLAB Function blocks. The proposed data types might cause data type mismatch errors or overflows.
Model reference	Does not propose data types for referenced models.	Data type propagation errors might occur at the referenced model's boundaries.
Blocks whose output is always floating-point for floating-point inputs regardless of their output data type setting. For example, Discrete Filter block and many DSP System Toolbox blocks.	Might not propose data types for these blocks as they do not allow you to set the output data type to `double` or `single`.	Date type propagation errors might occur because the Fixed-Point Advisor is unable to lock down the output data type of these blocks.

Running the Fixed-Point Advisor

1. Open a model.
2. Start the Fixed-Point Advisor by:
   - Typing `fpcadvisor('model_name/subsystem_name')` at the MATLAB command line

- Selecting a subsystem and, from the Tools menu, selecting **Fixed-Point > Fixed-Point Tool** to open the Fixed-Point Tool. In the Fixed-Point Tool **Fixed-point preparation for selected system** pane, click **Fixed-Point Advisor**.
- Right-clicking a subsystem block and, from the subsystem context menu, selecting **Fixed-Point > Fixed-Point Tool** to open the Fixed-Point Tool. In the Fixed-Point Tool **Fixed-point preparation for selected system** pane, click **Fixed-Point Advisor**.

The Fixed-Point Advisor window opens.

3 Select the Fixed-Point Advisor folder in the left pane.

4 Run the advisor by:

- Selecting **Run to Failure** from the Run menu
- Right-clicking the Fixed-Point Advisor folder and selecting **Run to Failure** from the folder context menu

The Fixed-Point Advisor runs the tasks in order until a task fails. A waitbar is displayed while each task runs.

5 Review the results. If a task fails because input parameters are not specified, select an **Input Parameter**. Then continue running to failure by right-clicking the task and selecting **Continue** from the context menu. If the task fails for a different reason, fix the task as described in "Fixing a Task Failure" on page 5-8.

Fixing a Task Failure

Tasks fail when there is a step for you to take to convert your model from floating-point to fixed-point. For more information on why a specific task fails, see the Chapter 12, "Fixed-Point Advisor Reference".

You can fix a failure using three different methods:

- Fix the failure by following the instructions in the Analysis Result box. Use this method to fix failures individually. See "Manually Fixing Failures" on page 5-9

- Fix the failure using the Action box. Use this method to automatically fix all failures. See "Automatically Fixing Failures" on page 5-9.
- Fix the failure using the Model Advisor Results Explorer. Use this method to batch fix failures. See "Batch Fixing Failures" on page 5-10

Note A warning result is meant for your information. You can choose to fix the reported issue, or to move on to the next task.

Manually Fixing Failures

All checks have an Analysis Result box that describes the recommended actions to manually fix failures.

To manually fix warnings or failures within a task:

1. Optionally, save a restore point so you can undo the changes that you make. For more information, see "Saving a Restore Point" on page 5-11.
2. In the Analysis Result box, review the recommended actions. Use the information to make changes to your model.
3. To verify that the task now passes, in the Analysis box, click **Run This Task**.

Automatically Fixing Failures

You can automatically fix failures using the Action box. The action box applies all of the recommended actions listed in the Analysis Result box.

Caution You should review the Analysis Result box prior to automatically fixing failures to ensure that you want to apply all of the recommended actions.

Automatically fix all failures within a task using the following steps:

1. Optionally, save a restore point so you can undo the changes that you make. For more information, see "Saving a Restore Point" on page 5-11.

2 In the Action box, click **Modify All**.

   The Action Result box displays a table of changes.

3 To verify that the task now passes, in the Analysis box, click **Run This Task**.

Batch Fixing Failures

If a task fails and you want to explore the results and make batch changes, use the following steps.

1 Optionally, save a restore point so you can undo the changes that you make. For more information, see "Saving a Restore Point" on page 5-11.

2 In the Analysis box, click **Explore Result**.

   The Model Advisor Result Explorer dialog box opens.

3 Use the Model Advisor Result Explorer to modify block parameters.

4 When you have finished making changes, in the Fixed-Point Advisor window, click **Run This Task** to see if the changes you made results in the task passing. Continue fixing failures and rerunning the task until the task passes.

Restore Points

The Fixed-Point Advisor provides a model and data restore point capability for reverting changes that you made in response to advice from the Fixed-Point Advisor. A restore point is a snapshot in time of the model, base workspace, and Fixed-Point Advisor.

Caution A restore point saves only the current working model, base workspace variables, and Fixed-Point Advisor tree. It does not save other items, such as libraries and referenced submodels.

To learn how to save a restore point, see "Saving a Restore Point" on page 5-11.

To learn how to load a restore point, see "Loading a Restore Point" on page 5-12.

Saving a Restore Point

When to Save a Restore Point

Consider saving a restore point:

- Before applying changes to your model that might cause update diagram failure. For example, before applying proposed data types in task 3.1.
- Before attempting to fix failures.

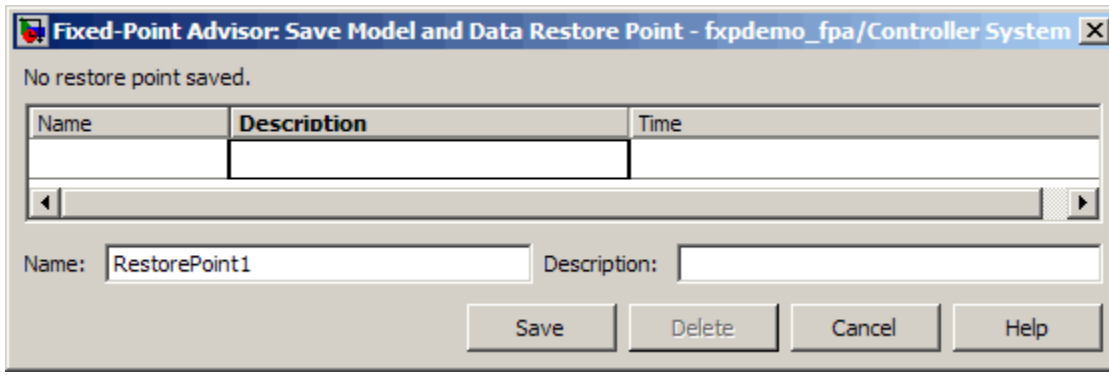
How to Save a Restore Point

You can save a restore point and give it a name and optional description, or allow the Fixed Point Advisor to automatically name the restore point for you.

To save a restore point with a name and optional description:

1. From the main menu, select **File > Save Restore Point As**.

   The **Save Model and Data Restore Point** dialog box opens.



2. In the **Name** field, enter a name for the restore point.

3. In the **Description** field, you can optionally add a description to help you identify the restore point.

4. Click **Save**.

   The Fixed Point Advisor saves a restore point of the current model, base workspace, and Fixed Point Advisor status.

Note To quickly save a restore point, go to **File > Save Restore Point**. The Fixed Advisor saves a restore point with the name `autosaven`, where `n` is the sequential number of the restore point. If you use this method, you cannot change the name of, or add a description to, the restore point.

Loading a Restore Point

When to Load a Restore Point

Load a restore point when:

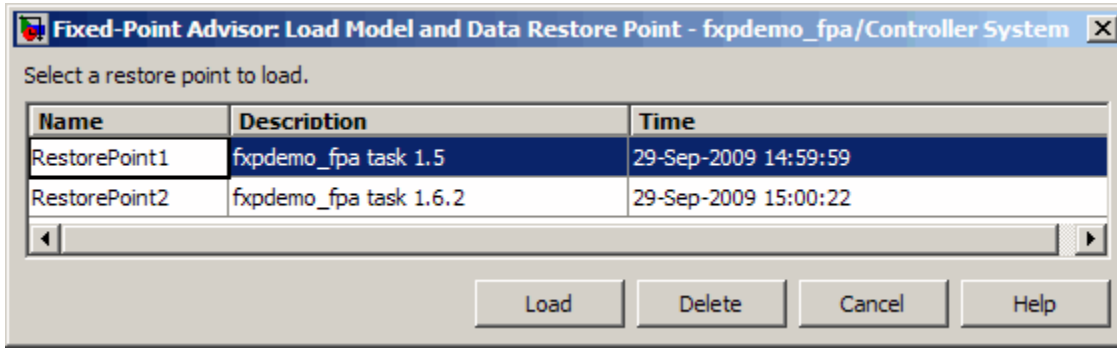
- A task fails and you cannot continue the conversion. In this case, load a restore point saved earlier in the run to avoid having to rerun all the previous tasks.
- You want to revert changes you made in response to advice from the Fixed-Point Advisor.

How to Load a Restore Point

To load a restore point:

1. Go to **File > Load Restore Point**.

   The **Load Model and Data Restore Point** dialog box opens.



2 Select the restore point that you want.

3 Click **Load**.

The Model Advisor issues a warning that the restoration will overwrite the current model and workspace.

4 Click **Load** to load the restore point you selected.

The Fixed Point Advisor reverts the model, base workspace, and Fixed Point Advisor status.

Converting a Model from Floating- to Fixed-Point Using Simulation Data

In this section...

About This Example

This example steps you through using the Fixed-Point Advisor to prepare the `fxpdemo_fpa` model for conversion from using floating-point data types to using fixed-point data types. This example shows you how to:

- Set model-wide configuration options.
- Set block-specific parameters.
- Obtain an initial fixed-point data types for the model.
- Validate the fixed-point data types against the floating-point model.

Starting the Preparation

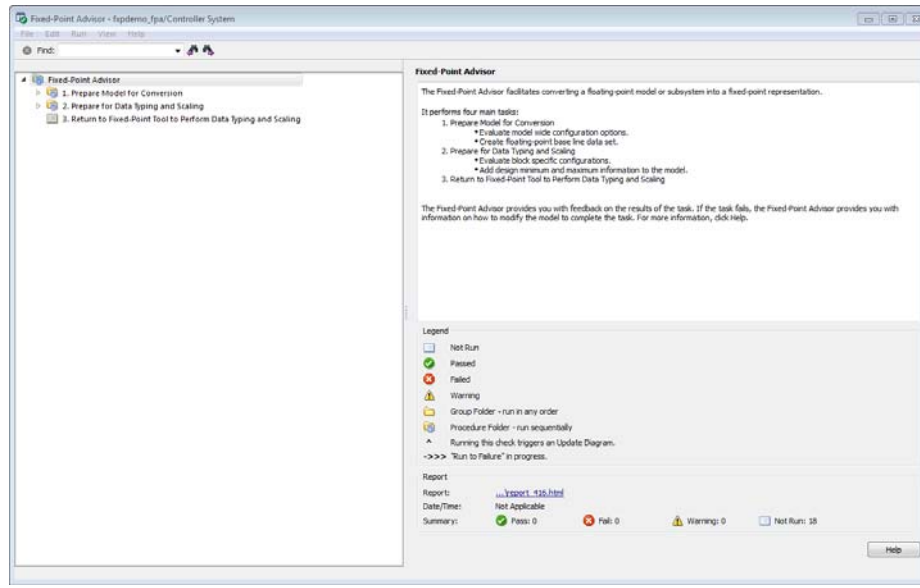
1. Open the model, at the command line, enter: `fxpdemo_fpa`.

2 To start the conversion:

   a Right-click **Controller System** and, from the subsystem context menu, select **Fixed-Point Tool**.

   b In the Fixed-Point Tool **Fixed-point preparation for selected system** pane, click the **Fixed-Point Advisor** button.

   The Fixed-Point Advisor opens for the subsystem `Controller System`.



Prepare Model for Conversion

First, validate model-wide settings and create reference simulation data.

1 For the purpose of this tutorial, run the tasks in the Fixed-Point Advisor **Prepare Model for Conversion** folder one at a time. In the left pane, select **Verify model simulation settings** and, in the right pane, click **Run This Task**.

   This task validates that model simulation settings allow signal logging and disables data type override to facilitate conversion to fixed point. These settings ensure that fixed-point data can be logged in downstream tasks.

The task passes.

2 Select and run **Verify update diagram status**.

Your model must be able to successfully complete an update diagram action to run the checks in the Fixed-Point Advisor.

The task passes.

3 Select and run **Address unsupported blocks**.

This task identifies blocks that do not support fixed-point data types. The Fixed-Point Advisor cannot convert these blocks. To complete the conversion of your model, replace these blocks with Simulink built-in blocks that do support fixed-point data types. If a replacement block is not available, you can temporarily isolate the unsupported block with Data Type Conversion blocks.

The task fails because the model contains a block that does not support fixed-point data types.

4 Fix the failure by replacing the TrigFcn block with the provided replacement:

a Click the **Preview** link to view the replacement block.

b Click the link to the original block and view its settings.

c Double-click the replacement block and verify its settings match the settings of the original block.

Note If the settings on the replacement block differ from the settings on the original block, set up the replacement block to match the original block.

---

d In the Controller System subsystem, right-click the original TrigFcn block. From the context menu, select `Replace with Lookup Table`.

The Fixed-Point Advisor replaces the original block.

e In the Fixed-Point Advisor, rerun the task. The task passes.

5. Select and run **Set up signal logging**. Because you are using simulation minimum and maximum data, you must specify at least one signal to use in analysis and comparison in downstream checks. You should log, at minimum, the unique input and output signals.

   The task runs and the Fixed-Point Advisor warns that signal logging is not specified for any signals.

6. Because you want to propose data types based on simulation data, fix the warning:

   a. Click the **Explore Result** button.

   b. In the Model Advisor Result Explorer, select the signals you want to log and select the **EnableLogging** check box.

      For this tutorial, log the signals connected to the Inport and Outport blocks:

      - `Ctr_in`
      - `Ctr_out`

      Tip

      - The menus at the top of the Model Advisor Result Explorer allow you to change the information displayed in the center pane.
      - You can edit properties in the Model Advisor Result Explorer. For more information, see "The Model Explorer: Overview" in the Simulink documentation.
      - When you update a property value in the center pane, the Signal Properties (right) pane updates to contain the same information.

   a. Close the Model Advisor Result Explorer.

   b. In the Fixed-Point Advisor, rerun the task.

      The task passes because signal logging is enabled for at least one signal.

7 Select and run **Create simulation reference data**. The Fixed-Point Advisor simulates the model using the current solver settings, and creates and archives reference signal data to use for analysis and comparison in later conversion tasks.

Tip If the simulation is set up to have a long simulation time, after starting this task, you can stop the simulation by selecting the waitbar and then pressing **Ctrl+C**. This allows you to change the simulation time and continue without having to wait for the long simulation to complete.

The task runs and the Fixed-Point Advisor warns that logging is not enabled.

8 To fix the failure, in the Action box click **Modify All**.

The **Modify All** action configures the model to the settings recommended in the Analysis Result. The Action Result box displays a table of changes.

Caution You should review the Analysis Result box prior to automatically fixing failures to ensure that you want to apply all the recommended actions.

---

9 Click the **Run This Task** button.

The task passes and the tool stores the results in a run named `FPA_Reference`.

Tip You can view these results in the Fixed-Point Tool **Contents** pane.

---

10 Open the **Verify Fixed-Point Conversion Guidelines** folder. Select and run **Check model configuration data validity diagnostic parameters settings**. This task verifies that the **Configuration Parameters** > **Diagnostics** > **Data Validity** > **Parameters** options are all set to

`warning`. If these options are set to `error`, the model update diagram action fails in later tasks.

The task passes.

11 Select and run **Implement logic signals as Boolean data**. This task verifies that **Configuration Parameters** > **Optimization** > **Implement logic signals as Boolean data** is selected. If it is cleared, the code generated in downstream checks is not optimized.

The task passes.

12 Select and run **Check for proper bus usage**. This task identifies:

- Mux blocks that are bus creators
- Bus signals that the top-level model treats as vectors

Note This is a Simulink check. For more information, see "Check for proper bus usage" in the Simulink documentation.

The task passes.

13 Select and run **Simulation range checking**. This tasks verifies that the **Configuration Parameters > Diagnostics > Simulation range checking** option is not set to `none`. A warning is displayed because **Simulation range checking** is currently set to `none`. The recommended setting is `warning` so that warnings are generated when signals exceed the specified minimum or maximum values.

14 Fix the warning by applying the recommended setting using the **Modify All** button and rerun the task.

The task passes.

15 Select and run **Check for implicit signal resolution**. This task checks for models that use implicit signal resolution. To use the Fixed-Point Advisor for Simulink signal object scaling, turn off implicit signal resolution by setting the **Diagnostics > Data Validity > Signal resolution** property in the Configuration Parameters dialog box to `Explicit only` and enforce resolution for each of the signals and states that currently resolve successfully. For more information, see "Signal resolution" in the Simulink documentation.

The task passes because the model contains no Simulink signal objects.

The run to failure action has completed for the **Prepare Model for Conversion** folder. At this point, you can review the results report found at the folder level, or continue to the next folder.

Prepare for Data Typing and Scaling

This folder contains tasks that set the block configuration options and set output minimum and maximum values for blocks. The block settings from this task simplify the initial scaling. The optimal block configuration is achieved in later stages. The tasks in this folder prepare the model for scaling by the Fixed-Point Tool.

1. Right-click **Prepare for Data Typing and Scaling** and select **Run to Failure**.

   The Fixed-Point Advisor runs the **Review locked data type settings** task. This task identifies blocks that have their data type settings locked down, which excludes them from autoscaling.

   The task passes because it finds no blocks with locked scaling.

2. The Fixed-Point Advisor runs the **Remove output data type inheritance** task. This task identifies blocks with the **Output data type** property set to `Inherit`. Inherited data types might lead to data type propagation errors.

   The task fails because some blocks in the model have inherited output data types.

3. Fix the failure using the **Modify All** button to explicitly configure the output data types to the recommended values, and rerun the task.

   The task passes.

4. Continue running to failure. **Relax input data type settings** runs. This task identifies blocks with input data type constraints that might lead to data type propagation errors.

   The task passes because all blocks have flexible input data types.

5. **Verify Stateflow charts that have strong data typing with Simulink** runs. This task verifies that all Stateflow charts are configured to have strong data typing with Simulink I/O.

   The task passes because the model does not have any Stateflow charts.

6 **Remove redundant specification between signal objects and blocks** runs. This task identifies and removes redundant data type specification originating from blocks and Simulink signal objects.

   The task passes because the model contains no resolved Simulink signal objects.

7 **Verify hardware selection** runs. This task identifies the hardware device information in the **Hardware Implementation** pane of the Configuration Parameters dialog box.

   The task fails because the **Configuration Parameters > Hardware Implementation** option does not provide values for the **Device vendor** and **Device type** parameters.

8 Fix the failure:

   a Click the *Hardware Implementation Device settings* link.

   b In the Configuration Parameters dialog box **Hardware Implementation** pane, change:

      - **Device vendor** to `Generic`
      - **Device type** to `32-bit Embedded Processor`

   c Click **OK** to apply the settings.

      The changes are applied and the Configuration Parameters dialog box closes.

9 In the Fixed-Point Advisor window, rerun the task.

   The task fails because you must specify a default data type for floating-point data types that is suitable for the chosen hardware.

10 Fix the failure by setting **Default data type for all floating-point signals** to `int16`.

   The software uses this default data type for all output signals. The Fixed-Point Advisor suggests the `Same as embedded hardware integer` setting, which is `int32`. But, because the model performs many multiplications and you want the product to fit into `int32`, use `int16`.

11 Rerun the task.

The task passes.

12 Select and run **Specify block minimum and maximum values**.

The Fixed-Point Advisor warns you that you have not specified any minimum and maximum values. Ideally, you should specify block output and parameter minimum and maximum values for, at minimum, the Inport blocks in the system. You can specify the minimum and maximum values for any block in this step. Typically, they are determined during the design process based on the system you are creating.

13 Fix the warning by specifying minimum and maximum values for Inport blocks:

a Click the **Explore Result** button.

The Model Advisor Result Explorer opens, displaying the Inport blocks that do not have an output minimum and maximum specified.

b In the Model Advisor Result Explorer center pane, select `Ctr_in`. For the purpose of this tutorial, you want to specify the output minimum and maximum values for this block. Set **OutMin** to `-5` and set **OutMax** to `5`.

c Close the Model Advisor Result Explorer.

d In the Fixed-Point Advisor, rerun the task.

The task passes because minimum and maximum values are specified for all Inport blocks.

e For the purpose of this tutorial, do not specify other minimum and maximum values for other blocks. You can review the results report found at the folder level.

14 Select and run **Return to the Fixed-Point Tool to perform data typing and scaling**.

The Fixed-Point Advisor closes and returns you to the Fixed-Point Tool.

15 In the Fixed-Point Tool **Contents** pane, examine the results for the simulation reference run. One of the TrigFcn block outputs overflowed multiple times, indicating that the fixed-point settings on this block are

not suitable for the input range. To refine the fixed-point data types, first run the model with a global override of the fixed-point data types using double-precision numbers to avoid quantization effects. This provides a floating-point benchmark that represents the ideal output. Then, propose new data types based on these "ideal" results.

Propose Data Types Based on the Simulation Reference Run

Now, use the Fixed-Point Tool to propose fixed-point data types based on the simulation reference (`FPA_Reference`) run.

1. In the Fixed-Point Tool:
   a. Click the **Propose fraction lengths** button

.

b Because you are proposing data types based on fixed-point results, the tool issues a warning. In the warning dialog box, click **Yes**.

The Fixed-Point Tool proposes new data types for objects in the model and updates the results in the **Contents** pane.

2 In the Fixed-Point Tool, set the **Column View** to `Autoscaling with Simulation Min/Max View` to display information relevant to the proposal. The tool displays the proposed scaling in the **ProposedDT** column in the **Contents** pane.

To accommodate the full simulation range, the Fixed-Point Tool proposes new data types for some blocks in the model. Because the `TrigFcn` block is a linked library, the tool does not propose new data types for this block.

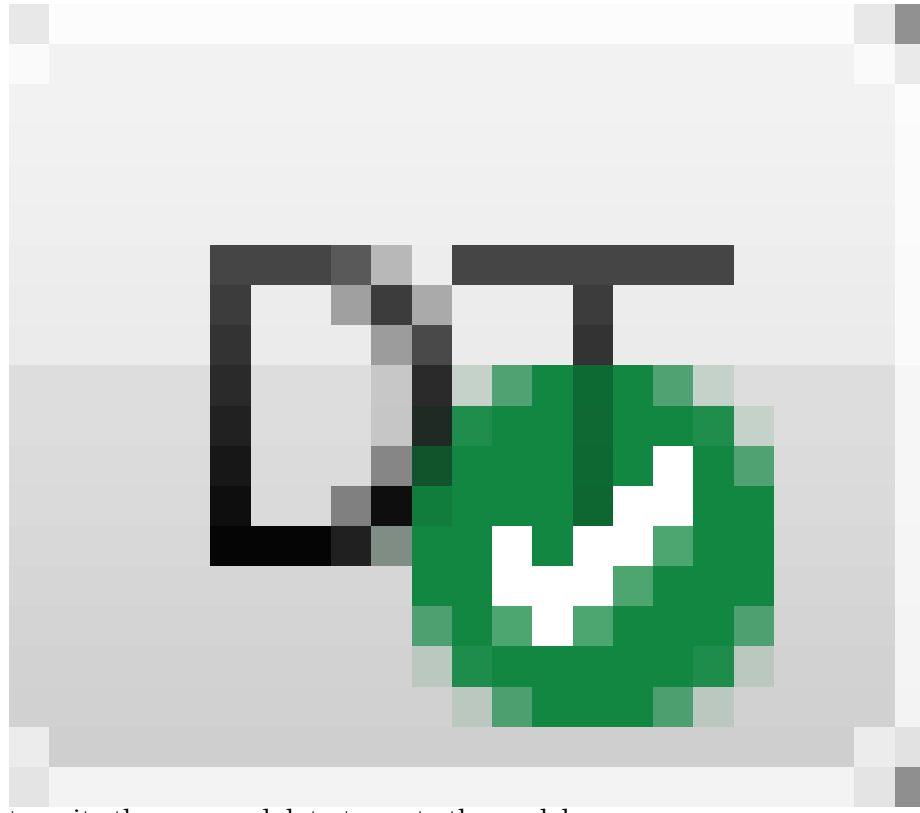
3 Examine the results to resolve any conflicts and to ensure that you want to accept the proposed data type for each result.

In the Fixed-Point Tool toolbar, select **Show > Conflicts with proposed data types**.

The Fixed-Point Tool detected no conflicts, so you are ready to apply the new data types as described in "Apply the New Fixed-Point Data Types" on page 7-24.

Apply the New Fixed-Point Data Types

1 Click **Apply accepted fraction lengths**



to write the proposed data types to the model.

2 In the Fixed-Point Tool toolbar, select **Show > All results**.

The tool has set all the specified data types to the proposed types.

Simulate Using New Fixed-Point Settings

1 In the **Shortcuts to set up runs** pane, click the **Model-wide no override and full instrumentation** button to use the locally specified data type settings.

2. In the Fixed-Point Tool **Model Hierarchy** pane, select the Controller Subsystem.

3. Click **Simulate** to run the simulation.

   The Simulink software simulates using the new scaling that you applied in the previous step and stores the results in the `NoOverride` run.

4. Examine the results. Because the tool did not propose new data types for the `TrigFcn` block, this block still overflows.

6

Fixed-Point Tool

Overview of the Fixed-Point Tool

In this section...

Introduction to the Fixed-Point Tool

The Fixed-Point Tool is a graphical user interface that automates the task of specifying fixed-point data types in a model. The tool collects range data for model objects, either from design minimum and maximum values that objects specify explicitly, from logged minimum and maximum values that occur during simulation, or from minimum and maximum values derived using range analysis. Based on these values, the tool proposes fixed-point scaling that maximizes precision and covers the range. The tool allows you to review the data type proposals and then apply them selectively to objects in your model.

Fixed-Point Tool Capability	For More Information
Deriving range information based on specified design range	Chapter 10, "Range Analysis"
Proposing data types based on simulation data	"Automatic Data Typing Using Simulation Data" on page 9-11
Proposing data types based on derived ranges	"Automatic Data Typing Using Derived Minimum and Maximum Values" on page 9-30
Proposing data types based on simulation data from multiple runs	"Proposing Data Types For a Model Using Results from Multiple Simulations" on page 9-73
Debugging fixed-point models	"Debugging a Fixed-Point Model" on page 6-12

Opening the Fixed-Point Tool

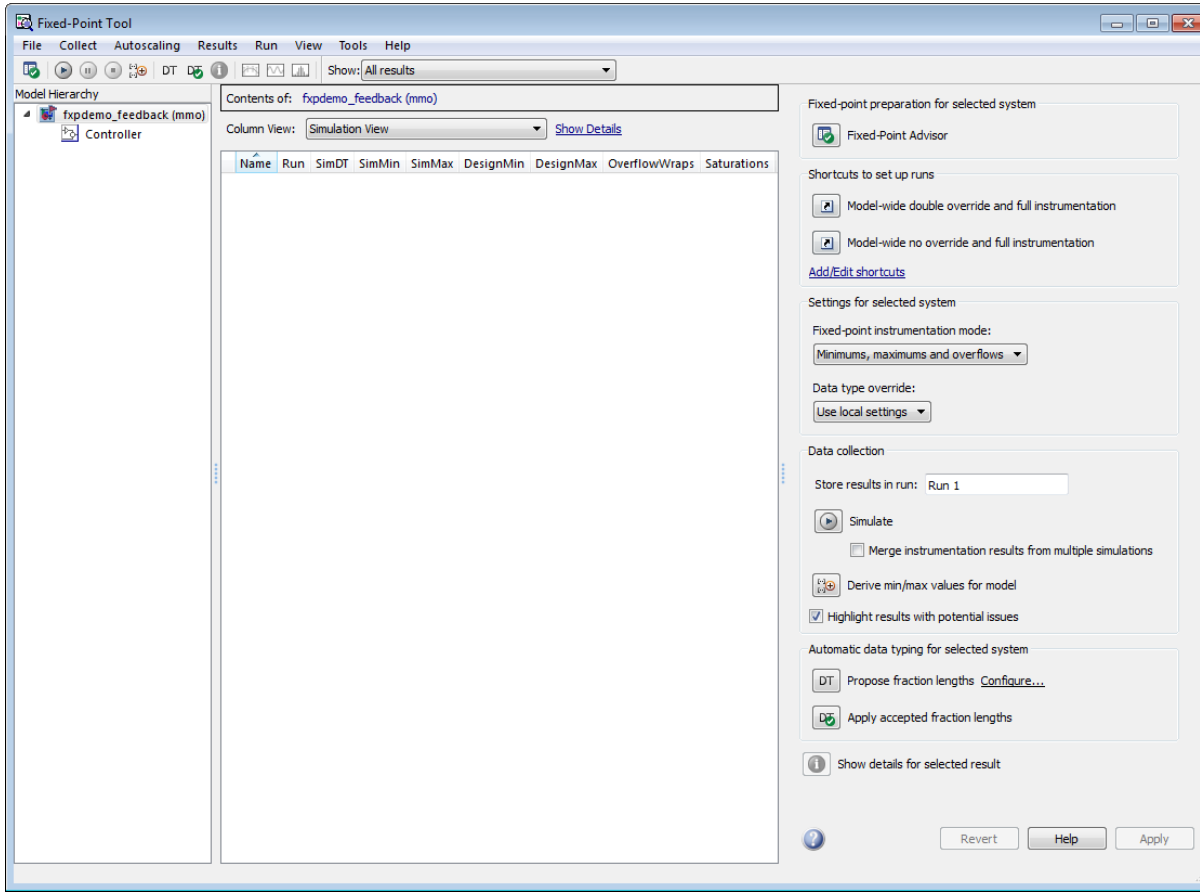
Use any of the following methods to open the Fixed-Point Tool:

- From the Simulink **Tools** menu, select **Fixed-Point Tool**.
- From a model's context (right-click) menu, select **Fixed-Point Tool**.
- From a subsystem's context (right-click) menu, select **Fixed-Point Tool**.

Alternatively, you can use the `fxptdlg` function to open the tool programmatically. See `fxptdlg` in the *Simulink Reference* for more information.

Understanding the Interface

When you first open the Fixed-Point Tool, it appears as shown here.



The Fixed-Point Tool contains the following components:

- **Model Hierarchy** pane — Displays a tree-structured view of the Simulink model hierarchy.
- **Contents** pane — Displays a tabular view of objects that log fixed-point data in a system or subsystem.
- Dialog pane — Displays parameters for specifying particular attributes of a system or subsystem, such as its data type override and fixed-point instrumentation mode.
- **Main** toolbar — Provides buttons that execute commonly used Fixed-Point Tool commands.
- **Shortcut Editor** — Provides ability to configure shortcuts that set up the run name as well as model-wide data type override and instrumentation settings prior to simulation or range derivation. Open the **Shortcut Editor** by clicking the Dialog Pane **Add/Edit shortcuts** link. For more information, see "Configuring Runs with the Shortcut Editor" on page 6-7.

For more information about each of these components, see the documentation for the `fxptdlg` function in the *Simulink Reference*.

Run Management

In this section...

Managing Runs with the Shortcut Editor

The Fixed-Point Tool supports multiple runs. Each run uses one set of model settings to simulate the model or to derive or propose data types. You can easily switch between different run setups using shortcuts. You can:

- Store multiple runs.
- Specify custom run names.
- Propose data types based on the results in any run.
- Apply data type proposals based on any run.
- Compare the results of any two runs.
- Rename runs directly in the Fixed-Point Tool **Contents** pane.

Managing Runs Manually

In addition to using shortcuts, you can manually manage runs using the following settings:

- In the **Data collection** pane, **Store results in run**.

  Provide a new run name before a simulation or collecting derived minimum and maximum values so that you do not overwrite existing runs.

- In the **Settings for selected system** pane:
  - **Fixed-point instrumentation mode**
  - **Data type override**
  - **Data type override applies to**

Configuring Runs with the Shortcut Editor

You can use shortcuts to configure the run name as well as model-wide data type override and instrumentation settings prior to simulation. The Fixed-Point Tool provides:

- Frequently used factory default shortcuts, such as **Model-wide double override and full instrumentation**, which sets up your model so that you can override all fixed-point data types with double-precision numbers and log the simulation minimum and maximum values and overflows.
- The ability to add and edit custom shortcuts. The shortcuts are saved with the model so that you need only define them once and then reuse them multiple times. Use the Shortcut Editor to create or edit shortcuts and to add and organize shortcut buttons in the Fixed-Point Tool **Shortcuts to set up runs** pane.

Why Use Shortcuts?

Shortcuts provide a quick and easy method to set up a run with pre-configured data type override and fixed-point instrumentation settings prior to simulation or range derivation. You can also associate a run name with each shortcut. When you apply a shortcut, you change the data type override and fixed-point instrumentation settings of multiple systems in your hierarchy at once.

Shortcuts:

- Simplify the workflow. For example, you can collect a floating-point baseline in a clearly named run.
- Provide the ability to configure data type override and instrumentation settings on multiple subsystems in the model hierarchy at the same time. This capability is especially useful for models that have a complicated hierarchy.
- Are a convenient way to store frequently used settings and reuse them. This capability is particularly useful when switching between different settings during debugging.
- Provide a way to store the original fixed-point instrumentation and data type override settings for the model. Preserving these settings in a shortcut provides a fallback in case of failure and a baseline for testing and validation.

When to Use Shortcuts

To do this	Use
Autoscale your entire model	The factory default shortcuts. These defaults provide an efficient way to override the model with floating-point data types or remove existing data type overrides. For example, see "Proposing Fraction Lengths for a Feedback Controller Using Simulation Range Data" on page 9-55.
Debug a model	Shortcuts to switch between different data type override and fixed-point instrumentation modes. For example, see "Debugging a Fixed-Point Model" on page 6-12.

To do this	Use
Manage the settings on multiple systems in a model. For example, if you are converting your model to fixed point one subsystem at a time.	The Shortcut Editor to define your own shortcuts so that you can switch between different settings without manually changing individual settings each time.
Capture the initial settings of the model before making any changes to it.	The Shortcut Editor to capture the model settings and save them in a named run. For more information, see "How to Capture Current Model Settings Using the Shortcut Editor" on page 6-11.

Adding Shortcuts

1. On the Fixed-Point Tool **Shortcuts to set up runs** pane, click **Add/Edit shortcuts**.

2. For each subsystem that you want to specify a shortcut for, In the Shortcut Editor **Model Hierarchy** pane, select the subsystem:

   a. Enter the shortcut name in the **Name of shortcut** field.

      By default, if **Allow modification of run name** is selected, the software sets the **Run name** to the shortcut name. You can manually override the name.

   b. Edit the shortcut properties. See "Editing Shortcuts" on page 6-9.

      For example, see "Setting Up Shortcuts" on page 6-15.

Editing Shortcuts

1. On the Fixed-Point Tool **Shortcuts to set up runs** pane, click **Add/Edit shortcuts**.

2. In the Shortcut Editor, from the **Name of shortcut** list, select the shortcut that you want to edit.

The editor displays the run name, fixed-point instrumentation settings, and data type override settings defined by the shortcut.

Note You cannot modify the factory default shortcuts.

---

3. If you do not want this shortcut to modify the existing fixed-point instrumentation settings on the model, clear **Allow modification of fixed-point instrumentation settings**.

4. If you do not want this shortcut to modify the existing data type override settings on the model, clear **Allow modification of data type override settings**.

5. If you do not want this shortcut to modify the run name on the model, clear **Allow modification of run name**.

6. If you want to modify the shortcut for a subsystem:

   a. Select the subsystem.

   b. If applicable, set the **Fixed-point instrumentation mode** that you want the Fixed-Point Tool to use when you apply this shortcut.

   c. If applicable, set the **Data type override** mode that you want the Fixed-Point Tool to use when you apply this shortcut.

   d. If applicable, set the **Run name** that you want the Fixed-Point Tool to use when you apply this shortcut.

   e. Click **Apply**.

7. Repeat the previous step until you have modified the shortcut for the subsystems of interest.

8. Optionally, if you want the Fixed-Point Tool to display a button for this new shortcut, use the right arrow control to move the shortcut to the list of shortcuts to display. You can change the order of the shortcut buttons by using the up and down arrow controls.

9. Save the model to store the shortcut with the model.

Deleting Shortcuts

To delete a shortcut permanently from a model:

1. On the Fixed-Point Tool **Shortcuts to set up runs** pane, click **Add/Edit shortcuts**.
2. On the Shortcut Editor **Manage shortcuts** pane, in the **Shortcuts** table, select the shortcut that you want to delete.
3. Click the **Delete selected shortcut** button, .

How to Capture Current Model Settings Using the Shortcut Editor

1. On the Fixed-Point Tool **Shortcuts to set up runs** pane, click **Add/Edit shortcuts**.
2. In the Shortcut Editor, create a new shortcut, for example, `Initial subsystem settings`.

   By default, if **Allow modification of run name** is selected, the software sets the **Run name** to the shortcut name. You can manually override the name.
3. Verify that **Allow modification of fixed-point instrumentation settings** and **Allow modification of data type override settings** are selected.
4. Click **Capture system settings**.

   The software sets the **Fixed-point instrumentation mode**, **Data type override**, and, if appropriate, **Data type override applies to**, settings to the settings for the systems in the model hierarchy.
5. Click **Apply**.
6. Save the model to store the shortcut with the model.

Debugging a Fixed-Point Model

In this section...

Learning Objectives

This example shows how to:

- Identify which parts of a model cause numeric problems.

  The current fixed-point settings on this model cause overflows. You debug the model by overriding the fixed-point settings on one subsystem at a time and simulating the model to determine how these fixed-point settings affect the model behavior.

- Create and use shortcuts to set up fixed-point instrumentation and data type override settings for different runs.

  Because you need to optimize the model for two different inputs, you switch several times between different data type override and fixed-point instrumentation settings. Using shortcuts facilitates changing these settings.

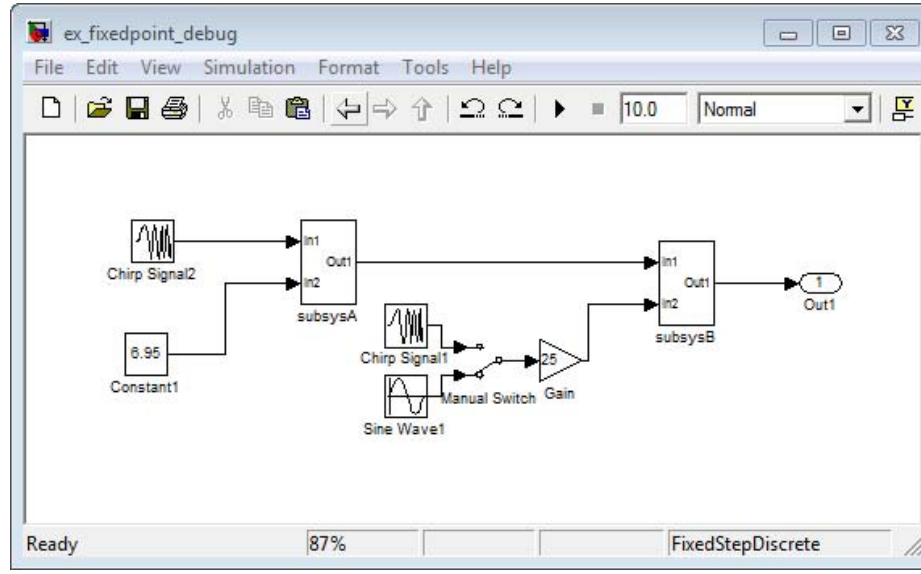
- Autoscale the model over the complete simulation range for both inputs.

Initial System Behavior

Initially, the input to the Gain block is a sine wave of amplitude 7. Simulate the model using local system settings and with logging enabled to see if any overflows or saturations occur.

1 Open the `ex_fixedpoint_debug` model. At the MATLAB command line, enter:

```
run(docpath(fullfile(docroot,'toolbox','fixpoint','examples','ex_fixedpoint_debug.mdl')))
```



2 From the model **Tools** menu, select **Fixed-Point Tool**.

3 In the Fixed-Point Tool, set up a shortcut for the initial system settings:

   a On the **Shortcuts to set up runs** pane, click **Add/Edit shortcuts**.

   b In the Shortcut Editor:

      i In the **Model Hierarchy** pane, select `subsysA>Math1`.

      ii In the **Name of shortcut field**, enter `Setting A`.

      The editor sets the **Run name** for this shortcut to `Setting A`.

      iii Set **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`.

      iv Set **Data type override** to `Use local settings`.

v Click **Apply**.

vi In the **Model Hierarchy** pane, select `subsysA>Math2` and repeat steps (iii) to (v).

vii In the **Manage shortcuts** pane, under **Shortcuts**, select `Setting A` then click the right arrow control to move this shortcut to the list of shortcuts displayed in the Fixed-Point Tool.

4 Use this shortcut to set up a run and then use the settings to simulate the model.

a In the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug`.

b In the **Shortcuts to set up runs** pane, click **Setting A**.

c Click the **Simulate** button, 

The Simulink software simulates the model using the fixed-point instrumentation and data type settings specified in **Setting A**. Afterward, the Fixed-Point Tool displays in its **Contents** pane the simulation results for each block that logged fixed-point data. The tool stores the results in the run named **Setting A**. The Fixed-Point tool highlights `subsysB/Math2/Add1:Output` in red to indicate that there is an issue with this result. The **OverflowWraps** column for this result shows that the block overflowed 51 times, which indicates a poor guess for its scaling.

Debugging the Model Behavior

To debug the model, first simulate the model using local settings on the subsystem `Math1` while overriding the fixed-point settings on `Math2` with doubles. Simulating subsystem `Math2` with doubles override avoids quantization effects for this subsystem. If overflows occur, you can deduce that there are problems with the fixed-point settings in subsystem `Math1`.

Next, simulate using local settings on `Math2` and doubles override on `Math1`. If overflows occur for this simulation, you can deduce that there are problems with the fixed-point settings for subsystem `Math2`.

Setting Up Shortcuts

1 Use the Shortcut Editor to create the following new shortcuts.

Shortcut Name	Subsystem	Fixed-point instrumentation mode	Data type override	Data type override applies to
Setting B	Math1	MinMaxAndOverflow	Use local settings	N/A
	Math2	MinMaxAndOverflow	Double	All numeric types
Setting C	Math1	MinMaxAndOverflow	Double	All numeric types
	Math2	MinMaxAndOverflow	Use local settings	N/A

2 In the **Manage shortcuts** pane, add `Setting B` and `Setting C` to the list of buttons to display in the Fixed-Point Tool.

Running the Model to Test Subsystem Math1 Settings

Simulate with original fixed-point settings on `Math1` while overriding the fixed-point settings with doubles on `Math2`.

1 In the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug`.

2 In the **Shortcuts to set up runs** pane, click **Setting B** to override fixed-point settings on `Math2`.

3 Click the **Simulate** button.

The Simulink software simulates the model using the fixed-point instrumentation and data type settings specified in **Setting B**, using fixed-point settings for `Math1` and overriding the fixed-point settings for `Math2`. No overflows occur, which indicates that the settings on `Math1` are not causing the overflows.

Running the Model to Test Subsystem Math2 Settings

Simulate with original fixed-point settings on `Math2` while overriding the fixed-point settings with doubles on `Math1`.

1. In the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug`.
2. In the **Shortcuts to set up runs** pane, click **Setting C** to override the fixed-point settings on `Math1`.
3. Click the **Simulate** button.

   The Simulink software simulates the model using the fixed-point instrumentation and data type settings specified in **Setting C**, using fixed-point settings for `Math2` and overriding the fixed-point settings for `Math1`. Overflows occur in run `Setting C`, indicating that the settings on `Math2` are causing the overflows.

Simulating Using a Different Input Stimulus

Simulate the model with a different input using the original fixed-point settings on subsystems `Math1` and `Math2`. Because you set up a shortcut for this initial set up, you can easily configure the model before rerunning the simulation. Before simulating, select to merge the simulation results so that the tool gathers the simulation range for both inputs.

1. In the **Data collection** pane, select **Merge instrumentation results from multiple simulations**.
2. In the `ex_fixedpoint_debug` model, double-click the Manual Switch block to select `Chirp Signal1` as the input to the Gain block.
3. In the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug` and simulate using the original fixed-point settings for `Math1` and `Math2`.
   a. In the **Shortcuts to set up runs** pane, click **Setting A**.
   b. Click the **Simulate** button.

      The Simulink software simulates the model using the fixed-point instrumentation and data type settings specified in **Setting A**.

Afterward, the Fixed-Point Tool displays in its **Contents** pane the simulation results for each block that logged fixed-point data. The tool stores the results in the run named **Setting A**.

Tip In the Fixed-Point Tool **Contents** pane, click **Run** to sort the results in this column.

Debugging with the New Input

1. Simulate with original fixed-point settings on `Math1` while overriding the fixed-point settings with doubles on `Math2`.
   a. In the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug`.
   b. In the **Shortcuts to set up runs** pane, click **Setting B**.
   c. Click the **Start** button.

   No overflows occur, which indicates that the settings on `Math1` are not causing the overflows.

2. Finally, simulate with original fixed-point settings on `Math2` while overriding the fixed-point settings with doubles on `Math1`.
   a. In the Fixed-Point Tool **Model Hierarchy** pane, select `ex_fixedpoint_debug`.
   b. In the **Shortcuts to set up runs** pane, click **Setting C**.
   c. Click the **Start** button.

   Overflows occur, which indicates that the fixed-point settings on `Math2` are causing the overflows. Next, use the Fixed-Point Tool to propose new data types for this subsystem.

Proposing Fraction Lengths for Math2 Based on Simulation Results

1. In the Fixed-Point Tool **Model Hierarchy** pane, select `Math2`.

2. In the **Automatic data typing for selected system** pane, click the **Propose fraction lengths** button.

3. In the **Propose Data Types** dialog box, select `Setting B` as the run to use for proposing data types and click **OK**. This run simulated `Math2` with double override to obtain the 'ideal' behavior of the subsystem based on the simulation results for both input stimuli.

   The Fixed-Point Tool proposes new fixed-point data types for the objects in subsystem `Math2` to avoid numerical issues such as overflows.

4. In the **Contents** pane **ProposedDT** column, examine the proposed data types for the objects in `Math2`. The tool proposed new fixed-point data types with reduced precision for the `Add1` block Output and Accumulator.

5. Because the Fixed-Point Tool marked all the proposed results with a green icon to indicate that the proposed data types pose no issues for these objects, accept the proposals.

   In the **Automatic data typing for selected system** pane, click the **Apply accepted fraction lengths** button.

Verifying the New Settings

Verify that the new settings do not cause any numerical problems by simulating the model using local settings for subsystems `Math1` and `Math2` and logging the results. Use shortcut `Setting A` that you set up for these settings.

1. In the **Shortcuts to set up runs** pane, click **Setting A**.

2. In the **Data collection** pane, set **Store results in run** to `Setting A2` and click **Apply** so that the Fixed-Point Tool does not overwrite the previous results for this shortcut.

3. Click the **Simulate** button.

   The Simulink software simulates the model using the new fixed-point settings. Afterward, the Fixed-Point Tool displays the simulation results in run `Setting A2`. No overflows or saturations occur indicating that the model can now handle the full input range.

7

Tutorial: Conversion to Fixed Point Workflow

Learning Objectives

In this tutorial, you learn how to:

- Convert a floating-point system to an equivalent fixed-point representation.

  This example demonstrates the recommended workflow for conversion when using proposing fraction lengths based on simulation data.

- Use the Fixed-Point Advisor to prepare your model for conversion.

  The Fixed-Point Advisor provides a set of tasks to help you convert a floating-point system to fixed point.

  You use the Fixed-Point Advisor to:

  - Set model-wide configuration options
  - Set block-specific dialog parameters
  - Check the model against fixed-point guidelines.
  - Identify unsupported blocks.
  - Remove output data type inheritance from blocks that use floating-point inheritance.
  - Promote simulation minimum and maximum values to design minimum and maximum values. This capability is useful if you want to derive ranges for objects in the model and you have not specified design ranges but you have simulated the model with inputs that cover the full intended operating range. For more information, see "Specify block minimum and maximum values" on page 12-33.

- Use the Fixed-Point Tool to propose scaling.

  The Fixed-Point Tool automates the task of specifying scaling for fixed-point data types in a system. In this example, the tool collects range data for model objects, either from design minimum and maximum values that you specify explicitly for signals and parameters, or from logged minimum and maximum values that occur during simulation. Based on these values, the tool proposes fixed-point scaling that maximizes precision and covers the range. The tool allows you to review the scaling proposals and then apply them selectively to objects in your model.

- Handle floating-point inheritance blocks during conversion.

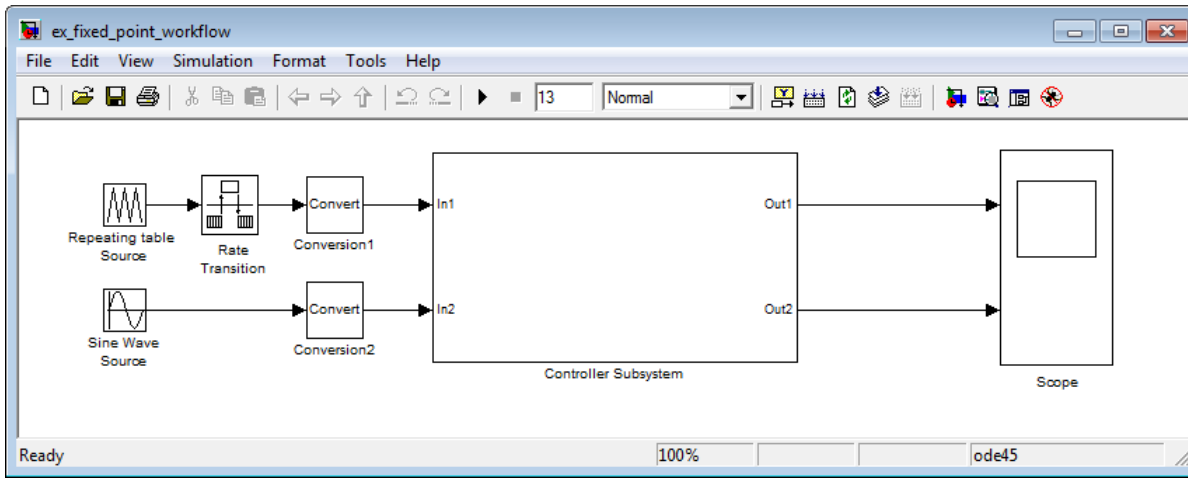
For floating-point inheritance blocks when inputs are floating point, all internal and output data types are floating point. The model in this tutorial uses a Discrete Filter block, which is a floating-point inheritance block.

Model Description

In this section...

Model Overview

This tutorial uses the `ex_fixed_point_workflow` model.



The model consists of a Source, a Controller Subsystem that you want to convert to fixed point, and a Scope to visualize the subsystem outputs. This method is how you configure a model to determine the effect of fixed-point data types on a system. Using this approach, you convert only the subsystem because this is the system of interest. There is no need to convert the Source or Scope to fixed point.

This configuration allows you to modify the inputs and collect simulation data for multiple stimuli. You can then examine the behavior of the subsystem with different input ranges and scale your fixed-point data types to provide maximum precision while accommodating the full simulation range.

Model Set Up

The model consists of the following blocks and subsystem.

Source

- **Repeating table Source**

  A Repeating Sequence (Repeating Table) block provides the first input to the Controller Subsystem and periodically repeats the sequence of data specified in the mask.

- **Rate Transition**

  A Rate Transition block outputs data from the Repeating table Source block at a different rate to the input.

- **Sine Wave Source**

  A Sine Wave block provides the second input to the Controller Subsystem.

  Initially, the amplitude of the Sine Wave block is 1. Later, you modify the amplitude to change the input range of the system.

- **Conversion1** and **Conversion2**

  These two Conversion blocks are set up so that the real-world values of their input and output are equal.

Controller Subsystem

The Controller Subsystem consists of:

- **Discrete Filter**

  The Discrete Filter block filters the Repeating table Source signal. The Discrete Filter is a floating-point inheritance block. For floating-point inheritance blocks, when inputs are floating-point, all internal and output data types are floating point.

- **Chart**

  The Chart consists of a Stateflow Chart block which converts the Sine Wave input to a positive output and multiplies it by 3.

- **Lookup Table for Chart**

The Lookup Table for Chart block is the first of two identical n-D Lookup Table blocks. This block receives the output from the Chart and, at each breakpoint, outputs the input multiplied by `10`.

- **Gain**

  The Gain block multiplies the Sine Wave input by `-3`.

- **Lookup Table for Gain**

  The Lookup Table for Gain block is a n-D Lookup Table block. It receives the output from the Gain block and, at each breakpoint, outputs its input multiplied by `10`.

- **Sum for Chart**

  This Sum block adds the outputs from the Discrete Filter and Lookup Table for Chart blocks and outputs the result to the Scope block.

- **Sum for Gain**

  This Sum block adds the outputs from the Discrete Filter and Lookup Table for Gain blocks and outputs the result to the Scope block.

Scope

- **Scope**

  The model includes a Scope block that displays the Controller Subsystem output signals.

Before You Begin

This tutorial demonstrates the recommended workflow for converting a floating-point system to fixed point using design and simulation data. It shows you how to use the Fixed-Point Advisor to prepare a floating-point subsystem for conversion to an equivalent fixed-point representation, and then how to use the Fixed-Point Tool to propose the fixed-point data types in the subsystem.

The tutorial uses the following recommended workflow:

1. "Prepare Floating-Point Model for Conversion to Fixed Point" on page 7-8.

   Step through the Fixed-Point Advisor tasks that prepare the floating-point subsystem for conversion to an equivalent fixed-point representation.

2. "Propose Data Types" on page 7-16.

   Propose scaling based on the simulation results. Examine the results to resolve any conflicts and to verify that you want to accept the proposed data type for each result.

3. "Apply Scaling" on page 7-19.

   Write the proposed data types to the model. Perform the automatic scaling procedure, which uses the double-precision simulation results to propose fixed-point scaling for appropriately configured blocks. The Fixed-Point Tool allows you to accept and apply the scaling proposals selectively.

4. "Verify Fixed-Point Settings" on page 7-19.

   Simulate the model again using the fixed-point settings. Compare the ideal results for the double-precision run with the fixed-point results.

5. Test the fixed-point settings with a different input stimulus and, if necessary, propose new data types to accommodate the simulation range for this input.

Tutorial Steps

In this section...

Open the Model

Open the `ex_fixed_point_workflow` model. At the MATLAB command line, enter:

```
run(docpath(fullfile(docroot,'toolbox','fixpoint','examples','ex_fixed_point_workflow.mdl')))
```

Prepare Floating-Point Model for Conversion to Fixed Point

The Fixed-Point Advisor provides a set of tasks that help you prepare a floating-point model or subsystem for conversion to an equivalent fixed-point representation. After preparing your model, you use the Fixed-Point Tool to perform the fixed-point conversion.

In this part of the tutorial, you use the Fixed-Point Advisor to prepare the Controller Subsystem in the `ex_fixed_point_workflow` model for conversion.

Open the Fixed-Point Advisor

1. In the `ex_fixed_point_workflow` model menu, select **Tools > Fixed-Point Tool**.

2. In the Fixed-Point Tool:

   a. In the **Model Hierarchy** pane, select the Controller Subsystem.

   b. In the **Fixed-point preparation for selected system** pane, click the **Fixed-Point Advisor** button.

   You run the Fixed-Point Advisor on the `ex_fixed_point_workflow` Controller Subsystem because this is the system of interest. There is no need to convert the system inputs or the display to fixed point.

Prepare Model for Conversion

1. In the Fixed-Point Advisor left pane, expand the **Prepare Model for Conversion** folder to view the tasks. For the purpose of this tutorial, run the tasks in the this folder one at a time. Select **Verify model simulation settings** and, in the right pane, select **Run this task**.

   This task validates that model simulation settings allow signal logging and disables data type override in the model and for `fi` objects or embedded numeric data types in your model or workspace. These settings facilitate conversion to fixed point in later tasks.

   A waitbar appears while the task runs. When the run is complete, the result shows that the task passed.

2. Select and run **Verify update diagram status**.

   Verify update diagram status runs. Your model must be able to successfully update diagram to run the checks in the Fixed-Point Advisor.

   The task passes.

3. Select and run **Address unsupported blocks**. This task identifies blocks that do not support fixed-point data types.

The task passes because the subsystem contains no blocks that do not support fixed-point data.

4 Select and run **Set up signal logging**. Prior to simulation, you must specify at least one signal for the Fixed-Point Advisor to use for analysis and comparison in downstream checks. You should log, at minimum, the unique input and output signals.

   The task generates a warning because signal logging is not specified for any signals.

5 Fix the warning using the Model Advisor Result Explorer:

   a Click the **Explore Result** button.

      The Model Advisor Result Explorer opens.

   b In the middle pane, select each signal you want to log and, next to the signal, select the corresponding **EnableLogging** check box.

      For this tutorial, log these signals:

      - `Lookup Table for Gain`
      - `Lookup Table for Chart`
      - `Chart`
      - `Discrete Filter`

   c Close the Model Advisor Result Explorer.

   d In the Fixed-Point Advisor window, click **Run This Task**.

      The task passes because signal logging is now enabled for at least one signal.

6 Select and run **Create simulation reference data**.

   The Fixed-Point Advisor simulates the model using the current solver settings, and creates and archives reference signal data in a run named `FPA_Reference` to use for analysis and comparison in later conversion tasks. This task also validates that model simulation settings allow signal logging and that the **Fixed-point instrumentation mode** is set to `Minimums, maximums and overflows`.

The Fixed-Point Advisor issues a warning and provides information in the Analysis Result box that logging simulation minimum and maximum values failed.

Logging failed because the **Fixed-point instrumentation mode** is `Use local settings`, but the recommended setting is `Minimums, maximums and overflows`.

7 To fix the failure, in the **Action** pane, click **Modify All**.

The Fixed-Point Advisor configures the model to the settings recommended in the Analysis **Result** pane. The **Action** pane displays a table of changes showing that the **Fixed-point instrumentation mode** is now `Minimums, maximums and overflows`

8 Click **Run This Task**.

Running the task after using the Modify All action verifies that you made the necessary changes. The Analysis **Result** pane updates to display a passed result and information about why the task passed.

Tip You can view the reference run data in the Fixed-Point Tool **Contents** pane in the run named **FPA_Reference**.

---

9 In the **Verify Fixed-Point Conversion Guidelines** folder, select and run **Check model configuration data validity diagnostic parameters settings**. This task verifies that the **Configuration Parameters** > **Diagnostics** > **Data Validity** > **Parameters** options are all set to `warning`. If these options are set to `error`, the model update diagram action fails in downstream checks.

The task passes because none of these options are set to `error`.

10 Select and run **Implement logic signals as Boolean data**. This task verifies that **Configuration Parameters** > **Optimization** > **Implement logic signals as Boolean data** is selected. If it is cleared, the code generated in downstream checks is not optimized.

The task passes.

11 Select and run **Check for proper bus usage**. This task identifies:

- Mux blocks that are bus creators
- Bus signals that the top-level model treats as vectors

Note This is a Simulink check. For more information, see "Check for proper bus usage" in the Simulink documentation.

The task runs and generates a warning because this check works only from top-level models and you are running from the subsystem. Because this model uses no buses, ignore this warning. For models containing buses, you must run the Fixed-Point Advisor from the top-level model to perform this check.

12 Select and run **Simulation range checking**. This tasks verifies that the **Configuration Parameters** > **Diagnostics** > **Simulation range checking** option is not set to `none`.

The task generates a warning because the Simulation range checking option is `none`.

13 To fix the warning, in the **Action** box, click **Modify All**.

The Fixed-Point Advisor sets the Simulation range checking option to `warning`.

14 Rerun the task.

The task now passes because the **Simulation range checking** option is correct.

15 Select and run **Check for implicit signal resolution**. This task checks for models that use implicit signal resolution.

The task fails because implicit signal resolution is enabled.

16 To fix the failure, in the **Action** box, click **Modify All**.

The Fixed-Point Advisor sets the **Signal resolution** option to `Explicit only`.

17 Rerun the task.

   The task now passes.

You have completed all the tasks for the **Prepare Model for Conversion** folder. At this point, you can review the results report found at the folder level, or continue to the next folder.

Prepare for Data Typing and Scaling

This folder contains tasks that set the block configuration options and set output minimum and maximum values for blocks. The block settings from this task simplify the initial scaling. Later tasks set optimal block configuration. The tasks in this folder prepare the model for automatic data typing in the Fixed-Point Tool.

1. For the purpose of this tutorial, run the tasks in the **Prepare for Data Typing and Scaling** folder one at a time.

   Open the **Prepare for Data Typing and Scaling** folder then select and run **Review locked data type settings**. This task identifies blocks that have their data type settings locked down which excludes them for autoscaling.

   This task passes because the model contains no blocks with locked scaling.

2. Select and run **Remove output data type inheritance**. This task identifies blocks that have an inherited output signal data type that might lead to data type propagation errors.

   This task fails because there are floating-point inheritance blocks in the model. For floating-point inheritance blocks, when inputs are floating-point, all internal and output data types are floating point. Therefore, you must specify an input parameter data type for these blocks.

3. In the Fixed-Point Advisor **Input Parameters** pane, set **Data type for blocks with floating-point inheritance** to `int16`, and rerun the task.

   The task fails and the Fixed-Point Advisor provides information about the failure in the Analysis Result box. The Fixed-Point Advisor recommends that you set:

- The input data type of the Discrete Filter block, which is a floating-point inheritance block, to a fixed-point data type to avoid floating-point inheritance.
- The output data type of all the other blocks that currently have their output data type set by inheritance rules to the compiled (current propagated) data type.

Tip Review the recommended data types prior to accepting them.

---

4 Fix the failure using the **Modify All** button to configure the output data types to the recommended values.

   The Action Result box displays:

   - A table showing the previous and current data types for all the floating-point inheritance blocks.
   - A table showing the previous and current data types for blocks that use other types of inheritance.

5 Rerun the task.

   The task passes.

6 Select and run **Relax input data type settings**. This task identifies blocks with input data type constraints that might cause data type propagation issues.

   The task passes because the model contains no blocks that have inherited input data types.

7 Select and run **Verify Stateflow charts have strong data typing with Simulink**. This task verifies that the configuration of all Stateflow charts ensures strong data typing with Simulink I/O.

   The task passes because the configuration of the Stateflow chart in the subsystem is correct.

8. Select and run **Remove redundant specification between signal objects and blocks**. This task identifies and removes redundant data type specification originating from blocks and Simulink signal objects.

   The task passes because the model contains no resolved Simulink signal objects.

9. Select and run **Verify hardware selection**. This task identifies the hardware device information in the **Hardware Implementation** pane of the Configuration Parameters dialog box. It also checks the default data type selected for floating-point signals in the model.

   The task fails because the default data type for all floating-point signals is set to `Remain floating-point`. Because the target hardware is an embedded processor, the Fixed-Point Advisor recommends that you set this value to the hardware integer used by the embedded hardware.

10. To fix the failure, in the **Input Parameters** pane, set **Default data type of all floating-point signals** to `Same as embedded hardware integer`.

11. Rerun the task.

    The task passes.

12. Select and run **Specify block minimum and maximum values**. Ideally, you should specify block output and parameter minimum and maximum values for, at minimum, the Inport blocks in the system. You can specify the minimum and maximum values for any block in this step. Typically, you determine these values during the design process based on the system you are creating.

    The Fixed-Point Advisor warns you that you have not specified any minimum and maximum values.

13. Fix the warning by specifying minimum and maximum values for Inport blocks:

    a. Click the **Explore Result** button.

       The Model Advisor Result Explorer opens, showing that the Inport blocks, `In1` and `In2`, do not have output minimum and maximum values specified.

b In the center pane, select `In1`. This block receives the output from Repeating table Source, which has a minimum value of `10` and a maximum value of `20`. Therefore, set **OutMin** to `10` and set **OutMax** to `20` as follows:

i In the **OutMin** column for `In1`, select `[]` and replace with `10`.

ii In the **OutMax** column for `In1`, select `[]` and replace with `20`.

c Select `In2`. This block receives the output from Sine Wave block, which has a minimum value of `-1` and a maximum value of `1`. Therefore, set **OutMin** to `-1` and set **OutMax** to `1`.

d Close the Model Advisor Result Explorer.

e In the Fixed-Point Advisor, rerun the task.

The task passes because you specified minimum and maximum values for all Inport blocks.

The tool advises you to specify minimum and maximum values for all blocks if possible. For the purpose of this tutorial, do not specify other minimum and maximum values for other blocks.

You have completed all tasks in the **Prepare for Data Typing and Scaling** folder. At this point, you can review the results report found at the folder level, or continue to the next folder.

Return to Fixed-Point Tool to Perform Data Typing and Scaling

Select and run this task to close the Fixed-Point Advisor and return to the Fixed-Point Tool.

Propose Data Types

Use the Fixed-Point Tool to propose fixed-point data types for appropriately configured blocks based on the double-precision simulation results stored in the simulation reference run that the Fixed-Point Advisor created. These results are stored in the run named **FPA_Reference**. You can view the results in the Fixed-Point Tool **Contents** pane.

The tool proposes fixed-point data types and scaling based on the ranges of the Repeating table Source and Sine Wave inputs. You can then use the tool

to accept and apply the proposed data types selectively. In this example, you propose fraction lengths for the specified word lengths.

1. In the Fixed-Point Tool, click the **Propose fraction lengths** button DT.

   The Fixed-Point Tool analyzes the scaling of all fixed-point blocks whose:

   - **Lock output data type setting against changes by the fixed-point tools** parameter is not selected.
   - **Output data type** parameter specifies a generalized fixed-point number.
   - Data types are not inherited.

   The Fixed-Point Tool updates the results in the **Contents** pane.

2. In the Fixed-Point Tool, set the **Column View** to `Autoscaling with Simulation Min/Max View` to display information relevant to the proposal. The tool displays the proposed scaling in the **ProposedDT** column in the **Contents** pane. The tool does not propose data types for objects with inherited data types.

   To accommodate the full simulation range, the Fixed-Point Tool proposes scaling for blocks that do not have inherited data types. By default, it selects the **Accept** check box for these signals because the scaling proposal differs from the object's current scaling. If you apply scaling, the tool will apply the proposed scaling to these signals. For more information, see "Applying Proposed Data Types" on page 9-27.

Contents of: Controller Subsystem (mmo)

Column View: Autoscaling with Simulation Min/Max View Show Details

Name	Run	SimDT	Accept	ProposedDT	SpecifiedDT
Chart	FPA_Reference		☐	n/a	
Chart/ Demux	FPA_Reference		☐	n/a	
Chart/fixout/fixin	FPA_Reference	double	☑	fixdt(1,32,29)	double
Chart/fixout/out	FPA_Reference	double	☑	fixdt(1,32,28)	double
Chart/input	FPA_Reference	double	☐	n/a	Inherit: Same as Simulink
Chart/output	FPA_Reference	double	☑	fixdt(1,32,29)	double
Conversion	FPA_Reference	double	☑	fixdt(1,32,29)	double
Conversion1	FPA_Reference	double	☑	fixdt(1,32,24)	double
Discrete Filter : Denominator accumulator	FPA_Reference	double	☐	n/a	Inherit: Inherit via internal rule
Discrete Filter : Denominator product output	FPA_Reference	double	☐	n/a	Inherit: Inherit via internal rule
Discrete Filter : Numerator accumulator	FPA_Reference	double	☐	n/a	Inherit: Inherit via internal rule
Discrete Filter : Numerator product output	FPA_Reference	double	☐	n/a	Inherit: Inherit via internal rule
Discrete Filter : Output	FPA_Reference	double	☑	fixdt(1,16,18)	fixdt(1,16,0)
Discrete Filter : States	FPA_Reference	double	☐	n/a	Inherit: Same as input
Gain	FPA_Reference	double	☑	fixdt(1,32,29)	double
Gain : Gain	FPA_Reference		☐	n/a	Inherit: Same as input
In1	FPA_Reference		☑	fixdt(1,32,26)	double
In2	FPA_Reference		☑	fixdt(1,32,30)	double
Lookup Table for Chart	FPA_Reference	double	☑	fixdt(1,32,26)	double
Lookup Table for Chart : BreakpointsForDimension1	FPA_Reference		☐	n/a	Inherit: Inherit from 'Breakpoint data'
Lookup Table for Chart : Intermediate Results	FPA_Reference	double	☐	n/a	Inherit: Inherit via internal rule
Lookup Table for Chart : Table	FPA_Reference		☐	n/a	Inherit: Inherit from 'Table data'
Lookup Table for Gain	FPA_Reference	double	☑	fixdt(1,32,26)	double
Lookup Table for Gain : BreakpointsForDimension1	FPA_Reference		☐	n/a	Inherit: Inherit from 'Breakpoint data'
Lookup Table for Gain : Intermediate Results	FPA_Reference	double	☐	n/a	Inherit: Inherit via internal rule
Lookup Table for Gain : Table	FPA_Reference		☐	n/a	Inherit: Inherit from 'Table data'
Sum : Accumulator	FPA_Reference	double	☐	n/a	Inherit: Inherit via internal rule
Sum : Output	FPA_Reference	double	☑	fixdt(1,32,26)	double
Sum1 : Accumulator	FPA_Reference	double	☐	n/a	Inherit: Inherit via internal rule
Sum1 : Output	FPA_Reference	double	☑	fixdt(1,32,26)	double

3 Examine the results to resolve any conflicts and to ensure that you want to accept the proposed data type for each result.

In the Fixed-Point Tool toolbar, select **Show > Conflicts with proposed data types**.

The Fixed-Point Tool detected no conflicts.

Tip If the tool does detect conflicts, you must resolve these before applying scaling. For more information, see "Examining Results to Resolve Conflicts" on page 9-22.

Now that you have reviewed the results and ensured that there are no issues, you are ready to apply the proposed scaling to the model, as described in "Apply Scaling" on page 7-19.

Apply Scaling

1. Click the **Apply accepted fraction lengths** button to write the proposed data types to the model. 

   The Fixed-Point Tool applies the scaling proposals to the subsystem blocks.

2. In the Fixed-Point Tool toolbar, select **Show > All results**.

   The tool has set all the specified data types to the proposed types.

You are now ready to check that the new scaling is acceptable, as described in "Verify Fixed-Point Settings" on page 7-19.

Verify Fixed-Point Settings

Next, you simulate again using the new fixed-point settings. You then use the Fixed-Point Tool plotting capabilities to compare the results from the floating-point **FPA_Reference** run with the fixed-point results.

1. In the Fixed-Point Tool **Model Hierarchy** pane, select the Controller Subsystem.

2. In the **Data collection** pane, set **Store results in run** to `Initial_fixed_point`. You specify a new run name to prevent the tool from overwriting the results that you want to retain in the **FPA_Reference** run.

3. Click the Fixed-Point Tool **Simulation** button  to run the simulation.

The Simulink software simulates using the new scaling that you applied in the previous step. Afterward, the Fixed-Point Tool displays in its **Contents** pane information about blocks that logged fixed-point data. The **SimDT** (simulation data type) column for the active run shows that the Controller Subsystem blocks use fixed-point data types with the new scaling.

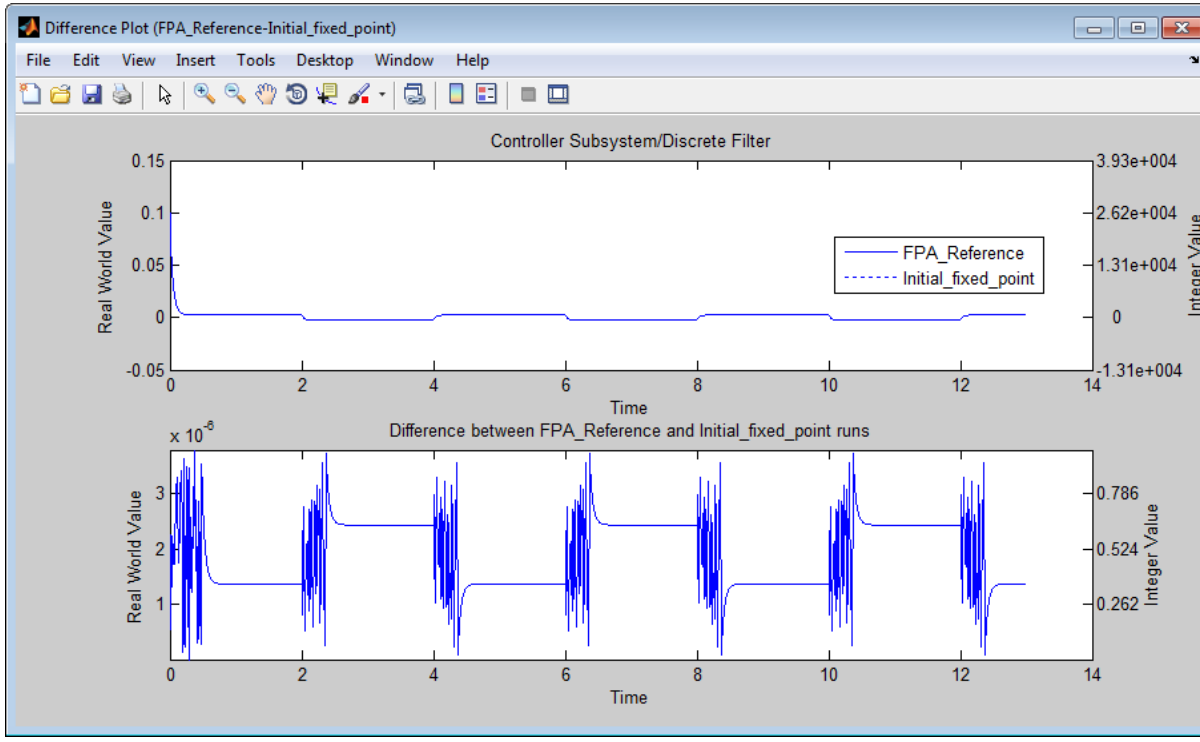
Tip In the **Contents** pane, click the **Run** column heading to sort the runs.

---

4 Examine the results to verify that there are no overflows or saturations.

5 In the Fixed-Point Tool **Model Hierarchy** pane, select the Controller Subsystem. In the **Contents** pane, select the `Discrete Filter: Output` that corresponds to the **FPA_Reference** run, and then click the **Difference Plot of Signal** button.

The Fixed-Point Tool plots the signal for the `FPA_Reference` and `Initial_fixed_point` runs, as well as their difference. The difference plot shows that the floating-point signal and the fixed-point signal are almost identical, the difference is on the order of `10^-5`.



6 Optionally, you can zoom in to view the steady-state region with greater detail. From the **Tools** menu of the figure window, select **Zoom In** and then drag the pointer to draw a box around the area you want to view more closely.

Now you are ready to test the fixed-point settings with new the input data, as described in "Test Fixed-Point Settings With New Input Data" on page 7-21.

Test Fixed-Point Settings With New Input Data

You have successfully used the Fixed-Point tool to propose fixed-point data types for your model. In the previous step, you saw that the numerical results for the double-precision system and the fixed-point system are very close. These results indicate that the fixed-point data types are suitable for the range of input data that you used. In practice, you might need to run multiple simulations to cover the entire design range of your system and use

the results of these simulations to refine the scaling of the fixed-point data types in your model.

In this part of the tutorial, you continue working on the model. First, you modify the range of the Sine Wave input and obtain simulation data based on this new range. Then, you use the Fixed-Point Tool to refine the model fixed-point settings based on the new simulation data. The Fixed-Point Tool proposes new data types that can accommodate the new input range.

To change the range of the input data and test the fixed-point settings:

1. In the `ex_fixed_point_workflow` model, double-click the Sine Wave Source block.

   The **Source Block Parameters** dialog box opens.

2. In this dialog box, change the **Amplitude** to `2` and click **OK**.

3. In the Fixed-Point Tool **Model Hierarchy** pane , select the Controller Subsystem.

4. In the **Data collection** pane, set **Store results in run** to `Input2`.

5. Click the Fixed-Point Tool **Simulation** button  to run the simulation.

   The Simulink software simulates the `ex_fixed_point_workflow` model. The Stateflow debugger reports a data overflow error in the Stateflow chart.

6. In the **Stateflow Debugging** window, under **Error checking options**, clear the **Data Range** option and close the debugger and the Chart.

   This action disables data range error detection and allows the simulation to run to completion.

7. In the Fixed-Point Tool **Model Hierarchy** pane, select the Controller Subsystem.

   The Fixed-Point Tool **Contents** pane displays the simulation results for each block in the subsystem that logged fixed-point data. The tool stores the results in the `Input2` run.

In the `Input2` run, the tool highlights in red the result for the Gain block, indicating that there are issues.

8 Examine the result for the Gain block.

The result shows that the Gain block output saturated, which indicates that the fixed-point data settings for this block are not suitable for the new input range.

Next, override the fixed-point data types with doubles and simulate the model again to obtain the ideal behavior of the subsystem, as described in "Gather a Floating-Point Benchmark" on page 7-23.

Gather a Floating-Point Benchmark

Run the model with a global override of the fixed-point data types using double-precision numbers to avoid quantization effects. This provides a floating-point benchmark that represents the ideal output. The Simulink software logs the signal logging results to the MATLAB workspace. The Fixed-Point Tool displays the simulation results including minimum and maximum values that occur during the run.

1 In the Fixed-Point Tool **Model Hierarchy** pane, select the Controller Subsystem.

2 In the **Settings for selected system** pane, set **Data type override** to `Double`.

Using this setting, the Fixed-Point Tool performs a global override of the fixed-point data types and scaling using double-precision numbers, thus avoiding quantization effects.

3 In the **Data collection** pane, set **Store results in run** to `DTO_Input2`.

4 Click the Fixed-Point Tool **Simulate** button  to run the simulation.

The Fixed-Point Tool highlights any simulation results that have issues, such as overflows or saturations.

5 In the Contents pane, click the Run column to sort the runs. Verify that there were no overflows or saturations in the **DTO_Input2** run.

Propose Data Types for the New Input

Now, use the Fixed-Point Tool to propose fixed-point data types based on the double-precision simulation results for the new input stored in the **DTO_Input2** run.

1. In the Fixed-Point Tool, click the **Propose fraction lengths** button DT.
2. In the **Propose Data Types** dialog box, select `DTO_Input2` as the run to use for proposing data types, and then click **OK**.

   The Fixed-Point Tool proposes new data types for all objects in the model and updates the results in the **Contents** pane.
3. In the Fixed-Point Tool, set the **Column View** to `Autoscaling with Simulation Min/Max View` to display information relevant to the proposal. The tool displays the proposed scaling in the **ProposedDT** column in the **Contents** pane. The tool does not propose data types for objects with inherited data types.

   To accommodate the full simulation range, the Fixed-Point Tool proposes new scaling with reduced precision for the Chart/output and Gain block output.
4. Examine the results to resolve any conflicts and to ensure that you want to accept the proposed data type for each result.

   In the Fixed-Point Tool toolbar, select **Show > Conflicts with proposed data types**.

   The Fixed-Point Tool detected no conflicts, so you are ready to apply the new data types as described in "Apply the New Fixed-Point Data Types" on page 7-24.

Apply the New Fixed-Point Data Types

1. Click **Apply accepted fraction lengths** to write the proposed data types to the model.
2. In the **Apply Data Types** dialog box, select `DTO_Input2` as the run to use for applying proposed data types and then click **OK**.

3 In the Fixed-Point Tool toolbar, select **Show > All results**.

   The tool has set all the specified data types to the proposed types.

Verify New Fixed-Point Settings

Finally, you simulate again using the new fixed-point settings. You then use the Fixed-Point Tool plotting capabilities to compare the results for the initial and final fixed-point settings.

1 In the Fixed-Point Tool **Model Hierarchy** pane, select the Controller Subsystem.

2 In the **Settings for selected system** pane, set **Data type override** to `Use local settings`.

3 In the **Data collection** pane, set **Store results in run** to `Final_fixed_point`.

4 Click **Start** to run the simulation.

   The Simulink software simulates using the new scaling that you applied in the previous step and stores the results in the `Final_fixed_point` run.

5 Examine the results to verify that there are no overflows or saturations.

6 In the Fixed-Point Tool **Model Hierarchy** pane , select the Controller Subsystem. In the **Contents** pane, select the `Discrete Filter: Output` that corresponds to the `Initial_fixed_point` run, and then click the **Difference Plot of Signal** button.

7 In the **Difference Plot Selector** dialog box, select `Final_fixed_point`, and then click **OK**.

   The Fixed-Point Tool plots the signal for both runs, as well as their difference. The difference plot shows that the floating-point signal and the fixed-point signal are identical.

8 Optionally, you can zoom in to view the steady-state region with greater detail. From the **Tools** menu of the figure window, select **Zoom In** and

then drag the pointer to draw a box around the area you want to view more closely.

Prepare for Code Generation

Optionally, use the Simulink Model Advisor to identify model settings that might lead to nonoptimal results in code generation.

1. From the Simulink **Tools** menu, select **Model Advisor**.
2. In the **System Selector** dialog box, select `Controller Subsystem`, and then click **OK**.
3. In the Model Advisor left pane, expand the **By Task** node.
4. Expand the **Code Generation Efficiency** node.
5. Select and run **Identify blocks that generate expensive saturation and rounding code**. This task optimizes the code to eliminate unnecessary saturation and rounding.

   The result is a warning because there are settings that can result in nonoptimized code. The Fixed-Point Advisor identified that:

   - The Gain block has the **Saturate on integer overflow** parameter selected. This setting can result in unnecessary condition-checking code.
   - The integer rounding mode selected for the model is `Undefined`. This setting results in inefficient generated code.
6. Fix the warning conditions.
   a. Click **Explore Result** to open the Model Advisor Result Explorer.
   b. Clear the **SaturateOnOverflow** setting for the Gain block and close the Model Advisor Result Explorer.
   c. In the Analysis Result box, click the **Embedded Hardware properties** link to open the Configuration Parameters dialog box Hardware Implementation pane.
   d. Set the **Signed integer division rounds to** parameter to `Zero` and click **OK** to close the dialog box.
7. Rerun the task.

The task passes.

8 Select and run **Identify questionable fixed-point operations**. This task identifies fixed-point operations that can lead to nonoptimal results.

The task passes.

Key Points to Remember

- Convert subsystems within your model, rather than the entire model. This practice saves time and avoids unnecessary conversions.
- Use the Fixed-Point Advisor to prepare your model for conversion to fixed point.
- Use the Fixed-Point Tool to propose fixed-point data types for your model or subsystem.
- When using the Fixed-Point Advisor, consider saving a restore point before applying recommendations.

  A restore point provides a fallback in case the recommended scaling causes subsequent update diagram failure. If you do not save a restore point and you encounter an update diagram failure, you must start the conversion from the beginning.
- Provide as much design minimum and maximum information as possible before starting the conversion to fixed point.

  Providing this information enables the fixed-point tools to choose fixed-point data types that maximize precision and cover the range.

  Specify minimum and maximum values for signals and parameters in the model for:

  - Inport and Outport blocks
  - Block outputs
  - The interface between MATLAB Function and Stateflow Chart blocks and the Simulink model to ensure strong data typing
  - Simulink.Signal objects

- Ensure that you simulate the system using the full range of inputs.

  If you use simulation minimum and maximum values to scale fixed-point data types, the tools provide meaningful results when exercising the full range of values over which your design is meant to run.

Where to Learn More

To learn more about...	See...
Fixed-Point Advisor capabilities	Chapter 12, "Fixed-Point Advisor Reference"
Best practices for using the Fixed-Point Advisor	"Best Practices for Using the Fixed-Point Advisor" on page 5-2
Using restore points in the Fixed-Point Advisor	"Restore Points" on page 5-10
Fixed-Point Tool capabilities	"Overview of the Fixed-Point Tool" on page 6-2 `fxptdlg`
Best practices for using the Fixed-Point Tool	"Best Practices for Using the Fixed-Point Tool to Propose Data Types for Your Simulink Model" on page 9-5
Using the Fixed-Point Tool to merge multiple simulation results	"Proposing Data Types For a Model Using Results from Multiple Simulations" on page 9-73

8

Tutorial: Producing Lookup Table Data

Overview

A function lookup table is a method by which you can approximate a function by a table with a finite number of points (X,Y). Function lookup tables are essential to many fixed-point applications. The function you want to approximate is called the *ideal function*. The X values of the lookup table are called the *breakpoints*. You approximate the value of the ideal function at a point by linearly interpolating between the two breakpoints closest to the point.

In creating the points for a function lookup table, you generally want to achieve one or both of the following goals:

- Minimize the worst-case error for a specified maximum number of breakpoints
- Minimize the number of breakpoints for a specified maximum allowed error

This tutorial shows you how to create function lookup tables using the function `fixpt_look1_func_approx`. You can optimize the lookup table to minimize the number of data points, the error, or both. You can also restrict the spacing of the breakpoints to be even or even powers of two to speed up computations using the table.

This tutorial also explains how to use the function `fixpt_look1_func_plot` to find the worst-case error of a lookup table and plot the errors at all points.

Worst-Case Error for a Lookup Table

In this section...

What Is Worst-Case Error for a Lookup Table?

The error at any point of a function lookup table is the absolute value of the difference between the ideal function at the point and the corresponding Y value found by linearly interpolating between the adjacent breakpoints. The *worst-case error*, or *maximum absolute error*, of a lookup table is the maximum absolute value of all errors in the interval containing the breakpoints.

For example, if the ideal function is the square root, and the breakpoints of the lookup table are 0, 0.25, and 1, then in a perfect implementation of the lookup table, the worst-case error is 1/8 = 0.125, which occurs at the point 1/16 = 0.0625. In practice, the error could be greater, depending on the fixed-point quantization and other factors.

The section that follows demonstrates how to use the function `fixpt_look1_func_plot` to find the worst-case error of a lookup table for the square root function.

Example: Square Root Function

This example shows how to use the function `fixpt_look1_func_plot` to find the maximum absolute error for the simple lookup table whose breakpoints are 0, 0.25, and 1. The corresponding Y data points of the lookup table, which you find by taking the square roots of the breakpoints, are 0, 0.5, and 1.

To use the function `fixpt_look1_func_plot`, you need to define its parameters first. To do so, type the following at the MATLAB prompt:

```
funcstr = 'sqrt(x)'; %Define the square root function
xdata = [0;.25;1]; %Set the breakpoints
ydata = sqrt(xdata); %Find the square root of the breakpoints
xmin = 0; %Set the minimum breakpoint
```

```
xmax = 1; %Set the maximum breakpoint
xdt = ufix(16); %Set the x data type
xscale = 2^-16; %Set the x data scaling
ydt = sfix(16); %Set the y data type
yscale = 2^-14; %Set the y data scaling
rndmeth = 'Floor'; %Set the rounding method
```

Next, type

```
errworst = fixpt_look1_func_plot(xdata,ydata,funcstr, ...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth)
```

This returns the worst-case error of the lookup table as the variable `errworst`:

```
errworst =
    0.1250
```

It also generates the plots shown in the following figure. The upper box (Outputs) displays a plot of the square root function with a plot of the fixed-point lookup approximation underneath. The approximation is found by linear interpolation between the breakpoints. The lower box (Absolute Error) displays the errors at all points in the interval from 0 to 1. Notice that the maximum absolute error occurs at 0.0625. The error at the breakpoints is 0.

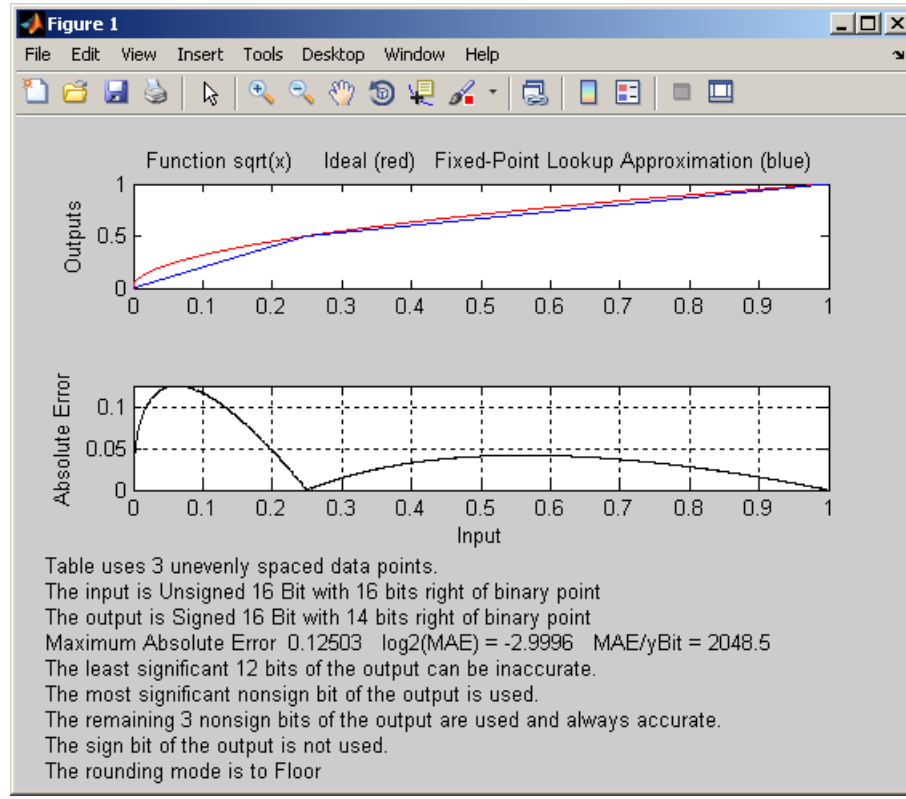


Figure 1
File Edit View Insert Tools Desktop Window Help
Function sqrt(x) Ideal (red) Fixed-Point Lookup Approximation (blue)
Outputs
1
0.5
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
Absolute Error
0.1
0.05
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
Input
Table uses 3 unevenly spaced data points.
The input is Unsigned 16 Bit with 16 bits right of binary point
The output is Signed 16 Bit with 14 bits right of binary point
Maximum Absolute Error 0.12503 log2(MAE) = -2.9996 MAE/yBit = 2048.5
The least significant 12 bits of the output can be inaccurate.
The most significant nonsign bit of the output is used.
The remaining 3 nonsign bits of the output are used and always accurate.
The sign bit of the output is not used.
The rounding mode is to Floor

Creating Lookup Tables for a Sine Function

In this section...

Introduction

The sections that follow explain how to use the function `fixpt_look1_func_approx` to create lookup tables. It gives examples that show how to create lookup tables for the function $\sin(2\pi x)$ on the interval from 0 to 0.25.

Parameters for fixpt_look1_func_approx

To use the function `fixpt_look1_func_approx`, you must first define its parameters. The required parameters for the function are

- `funcstr` — Ideal function
- `xmin` — Minimum input of interest
- `xmax` — Maximum input of interest
- `xdt` — x data type
- `xscale` — x data scaling

- `ydt` — y data type
- `yscale` — y data scaling
- `rndmeth` — Rounding method

In addition there are three optional parameters:

- `errmax` — Maximum allowed error of the lookup table
- `nptsmax` — Maximum number of points of the lookup table
- `spacing` — Spacing allowed between breakpoints

You must use at least one of the parameters `errmax` and `nptsmax`. The next section, "Setting Function Parameters for the Lookup Table" on page 8-8, gives typical settings for these parameters.

Using Only errmax

If you use only the `errmax` parameter, without `nptsmax`, the function creates a lookup table with the fewest points, for which the worst-case error is at most `errmax`. See "Example: Using errmax with Unrestricted Spacing" on page 8-8.

Using Only nptsmax

If you use only the `nptsmax` parameter without `errmax`, the function creates a lookup table with at most `nptsmax` points, which has the smallest worse case error. See "Example: Using nptsmax with Unrestricted Spacing" on page 8-11.

The section "Specifying Both errmax and nptsmax" on page 8-18 describes how the function behaves when you specify both `errmax` and `nptsmax`.

Spacing

You can use the optional `spacing` parameter to restrict the spacing between breakpoints of the lookup table. The options are

- `'unrestricted'` — Default.
- `'even'` — Distance between any two adjacent breakpoints is the same.
- `'pow2'` — Distance between any two adjacent breakpoints is the same and the distance is a power of two.

The section "Restricting the Spacing" on page 8-12 and the examples that follow it explain how to use the spacing parameter.

Setting Function Parameters for the Lookup Table

To do the examples in this section, you must first set parameter values for the `fixpt_look1_func_approx` function. To do so, type the following at the MATLAB prompt:

```
funcstr = 'sin(2*pi*x)'; %Define the sine function
xmin = 0; %Set the minimum input of interest
xmax = 0.25; %Set the maximum input of interest
xdt = ufix(16); %Set the x data type
xscale = 2^-16; %Set the x data scaling
ydt = sfix(16); %Set the y data type
yscale = 2^-14; %Set the y data scaling
rndmeth = 'Floor'; %Set the rounding method
errmax = 2^-10; %Set the maximum allowed error
nptsmax = 21; %Specify the maximum number of points
```

If you exit the MATLAB software after typing these commands, you must retype them before trying any of the other examples in this section.

Example: Using errmax with Unrestricted Spacing

The first example shows how to create a lookup table that has the fewest data points for a specified worst-case error, with unrestricted spacing. Before trying the example, enter the same parameter values given in the section "Setting Function Parameters for the Lookup Table" on page 8-8, if you have not already done so in this MATLAB session.

You specify the maximum allowed error by typing

```
errmax = 2^-10;
```

Creating the Lookup Table

To create the lookup table, type

```
[xdata ydata errworst] = fixpt_look1_func_approx(funcstr, ...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[]);
```

Note that the `nptsmax` and `spacing` parameters are not specified.

The function returns three variables:

- `xdata`, the vector of breakpoints of the lookup table
- `ydata`, the vector found by applying ideal function $\sin(2\pi x)$ to `xdata`
- `errworst`, which specifies the maximum possible error in the lookup table

The value of `errworst` is less than or equal to the value of `errmax`.

You can find the number of X data points by typing

```
length(xdata)

ans =
    16
```

This means that 16 points are required to approximate $\sin(2\pi x)$ to within the tolerance specified by `errmax`.

You can display the maximum error by typing `errworst`. This returns

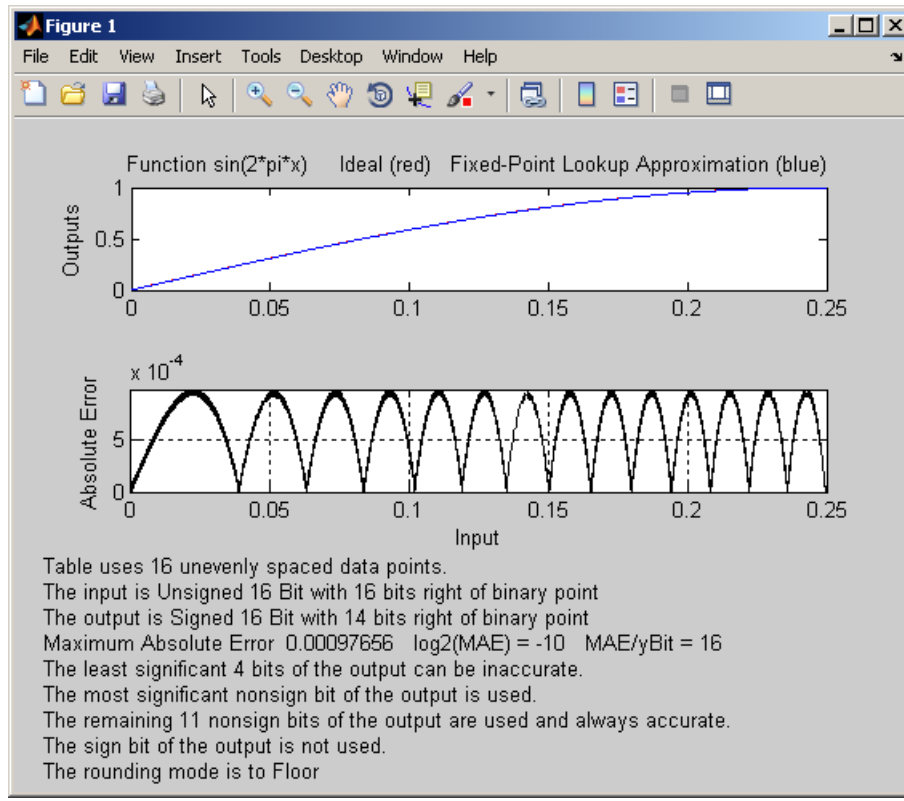
```
errworst =
  9.7656e-004
```

Plotting the Results

You can plot the output of the function `fixpt_look1_func_plot` by typing

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt, ...
xscale,ydt,yscale,rndmeth);
```

The resulting plots are shown.



The upper plot shows the ideal function sin(2π*x*) and the fixed-point lookup approximation between the breakpoints. In this example, the ideal function and the approximation are so close together that the two graphs appear to coincide. The lower plot displays the errors.

In this example, the Y data points, returned by the function `fixpt_look1_func_approx` as `ydata`, are equal to the ideal function applied to the points in `xdata`. However, you can define a different set of values for `ydata` after running `fixpt_look1_func_plot`. This can sometimes reduce the maximum error.

You can also change the values of `xmin` and `xmax` in order to evaluate the lookup table on a subset of the original interval.

To find the new maximum error after changing `ydata`, `xmin` or `xmax`, type

```
errworst = fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax, ...
xdt,xscale,ydt,yscale,rndmeth)
```

Example: Using nptsmax with Unrestricted Spacing

The next example shows how to create a lookup table that minimizes the worst-case error for a specified maximum number of data points, with unrestricted spacing. Before starting the example, enter the same parameter values given in the section "Setting Function Parameters for the Lookup Table" on page 8-8, if you have not already done so in this MATLAB session.

Setting the Number of Breakpoints

You specify the number of breakpoints in the lookup table by typing

```
nptsmax = 21;
```

Creating the Lookup Table

Next, type

```
[xdata ydata errworst] = fixpt_look1_func_approx(funcstr, ...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax);
```

The empty brackets, `[]`, tell the function to ignore the parameter `errmax`, which is not used in this example. Omitting `errmax` causes the function `fixpt_look1_func_approx` to return a lookup table of size specified by `nptsmax`, with the smallest worst-case error.

The function returns a vector `xdata` with 21 points. You can find the maximum error for this set of points by typing `errworst` at the MATLAB prompt. This returns

```
errworst =
  5.1139e-004
```

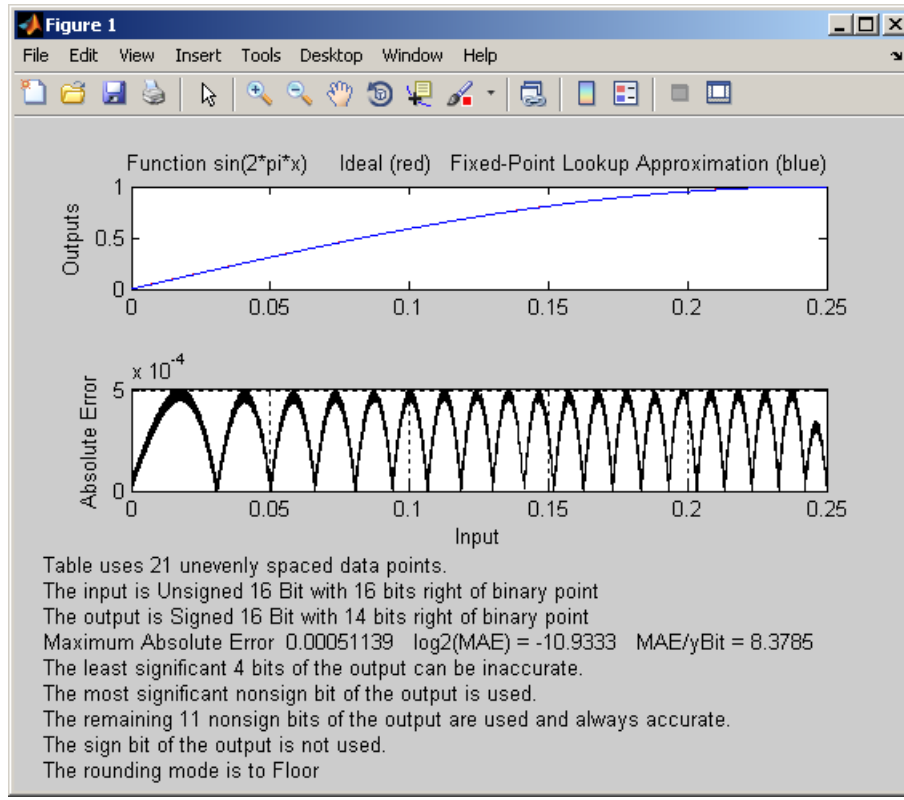
Plotting the Results

To plot the lookup table along with the errors, type

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt, ...
```

```
xscale,ydt,yscale,rndmeth);
```

The resulting plots are shown.



Restricting the Spacing

In the previous two examples, the function `fixpt_look1_func_approx` creates lookup tables with unrestricted spacing between the breakpoints. You can restrict the spacing to improve the computational efficiency of the lookup table, using the spacing parameter.

The options for spacing are

- `'unrestricted'` — Default.

- `'even'` — Distance between any two adjacent breakpoints is the same.
- `'pow2'` — Distance between any two adjacent breakpoints is the same and is a power of two.

Both power of two and even spacing increase the computational speed of the lookup table and use less command read-only memory (ROM). However, specifying either of the spacing restrictions along with `errmax` usually requires more data points in the lookup table than does unrestricted spacing to achieve the same degree of accuracy. The section "Effects of Spacing on Speed, Error, and Memory Usage" on page 8-22 discusses the tradeoffs between different spacing options.

Example: Using errmax with Even Spacing

The next example shows how to create a lookup table that has evenly spaced breakpoints and a specified worst-case error. To try the example, you must first enter the parameter values given in the section "Setting Function Parameters for the Lookup Table" on page 8-8, if you have not already done so in this MATLAB session.

Next, at the MATLAB prompt type

```
spacing = 'even';
[xdata ydata errworst] = fixpt_look1_func_approx(funcstr, ...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[],spacing);
```

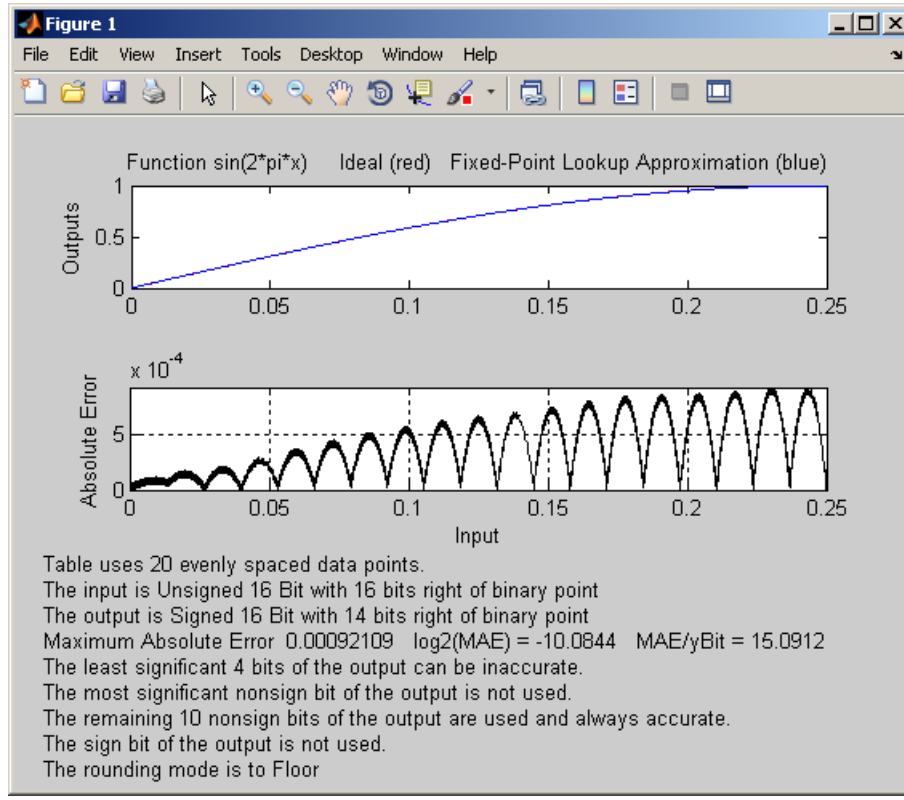
You can find the number of points in the lookup table by typing `length(xdata)`:

```
ans =
    20
```

To plot the lookup table along with the errors, type

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt, ...
xscale,ydt,yscale,rndmeth);
```

This produces the following plots:



Example: Using nptsmax with Even Spacing

The next example shows how to create a lookup table that has evenly spaced breakpoints and minimizes the worst-case error for a specified maximum number of points. To try the example, you must first enter the parameter values given in the section "Setting Function Parameters for the Lookup Table" on page 8-8, if you have not already done so in this MATLAB session.

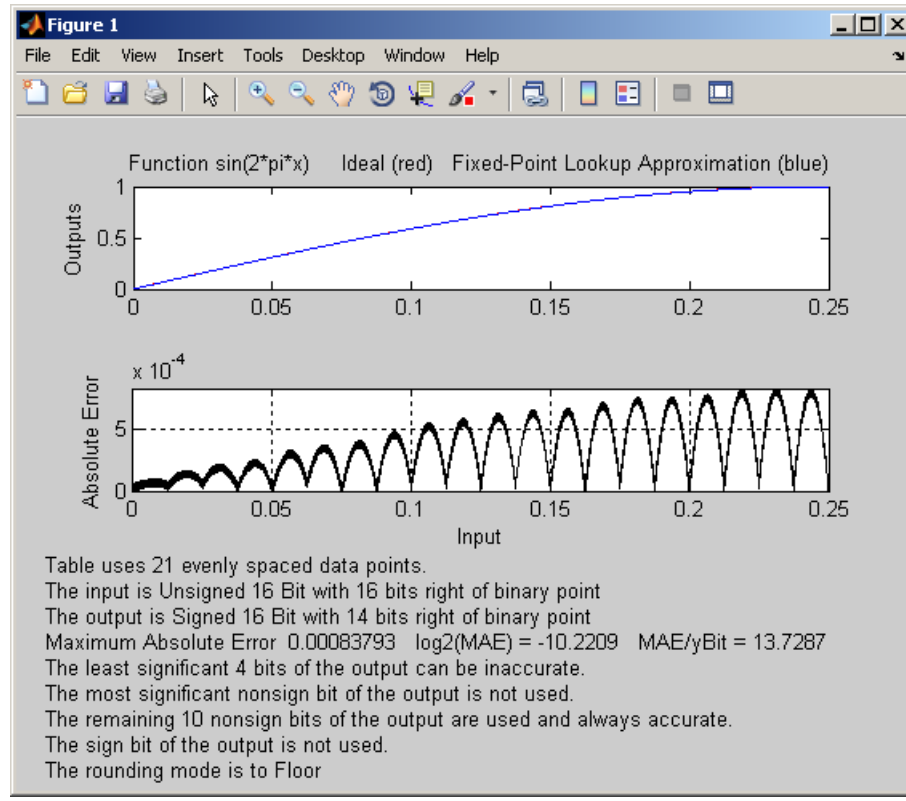
Next, at the MATLAB prompt type

```
spacing = 'even';
[xdata ydata errworst] = fixpt_look1_func_approx(funcstr, ...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax,spacing);
```

The result requires 21 evenly spaced points to achieve a maximum absolute error of `2^-10.2209`.

To plot the lookup table along with the errors, type

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt, ...
xscale,ydt,yscale,rndmeth);
```



Example: Using errmax with Power of Two Spacing

The next example shows how to construct a lookup table that has power of two spacing and a specified worst-case error. To try the example, you must first enter the parameter values given in the section "Setting Function

Parameters for the Lookup Table" on page 8-8, if you have not already done so in this MATLAB session.

Next, at the MATLAB prompt type

```
spacing = 'pow2';
[xdata ydata errworst] = ...
fixpt_look1_func_approx(funcstr,xmin, ...
xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[],spacing);
```

To find out how many points are in the lookup table, type

```
length(xdata)
```

```
ans =
    33
```

This means that 33 points are required to achieve the worst-case error specified by `errmax`. To verify that these points are evenly spaced, type

```
widths = diff(xdata)
```

This generates a vector whose entries are the differences between consecutive points in `xdata`. Every entry of `widths` is 2^{-7}.

To find the maximum error for the lookup table, type

```
errworst
```

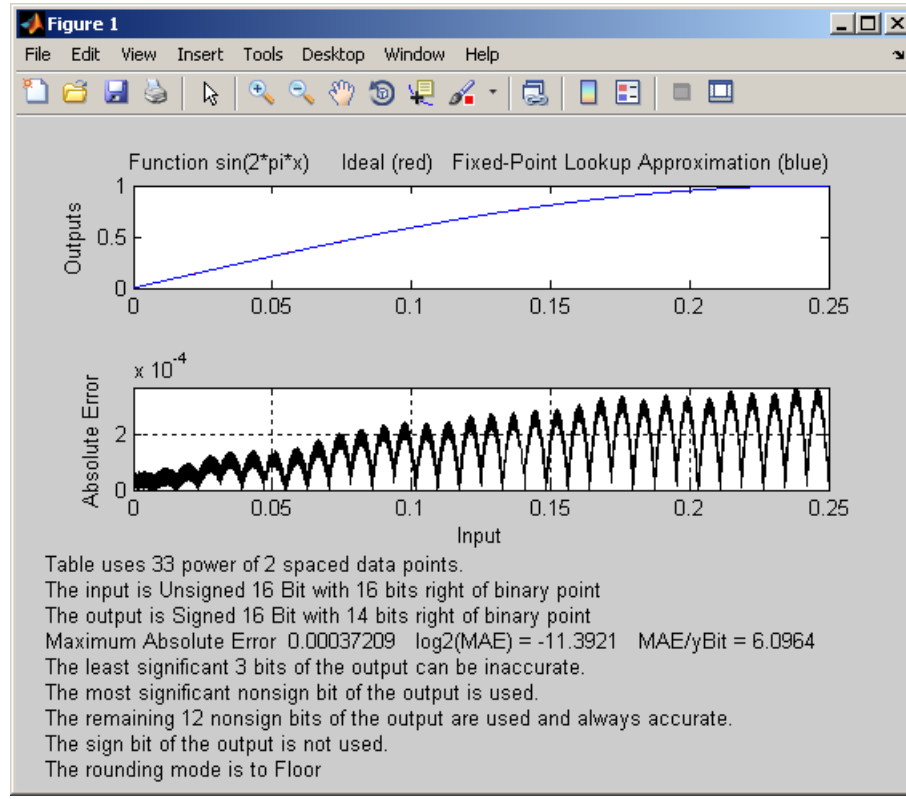
```
errworst =
  3.7209e-004
```

This is less than the value of `errmax`.

To plot the lookup table data along with the errors, type

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt, ...
xscale,ydt,yscale,rndmeth);
```

This displays the plots shown.



Example: Using nptsmax with Power of Two Spacing

The next example shows how to create a lookup table that has power of two spacing and minimizes the worst-case error for a specified maximum number of points. To try the example, you must first enter the parameter values given in the section "Setting Function Parameters for the Lookup Table" on page 8-8, if you have not already done so in this MATLAB session:

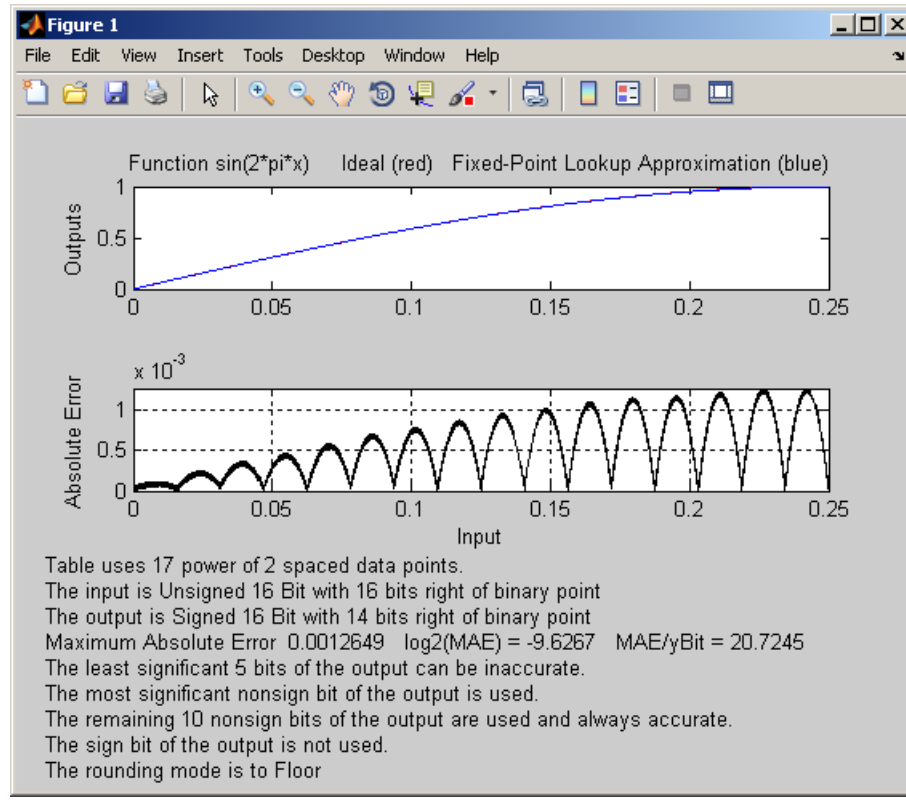
```
spacing = 'pow2';
[xdata ydata errworst] = fixpt_look1_func_approx(funcstr, ...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax,spacing);
```

The result requires 17 points to achieve a maximum absolute error of 2^-9.6267.

To plot the lookup table along with the errors, type

```
fixpt_look1_func_plot(xdata,ydata,funcstr,xmin,xmax,xdt, ...
xscale,ydt,yscale,rndmeth);
```

This produces the plots shown below:



Specifying Both errmax and nptsmax

If you include both the `errmax` and the `nptsmax` parameters, the function `fixpt_look1_func_approx` tries to find a lookup table with at most `nptsmax` data points, whose worst-case error is at most `errmax`. If it can find a lookup table meeting both conditions, it uses the following order of priority for spacing:

1. Power of two
2. Even
3. Unrestricted

If the function cannot find any lookup table satisfying both conditions, it ignores `nptsmax` and returns a lookup table with unrestricted spacing, whose worst-case error is at most `errmax`. In this case, the function behaves the same as if the `nptsmax` parameter were omitted.

Using the parameters described in the section "Setting Function Parameters for the Lookup Table" on page 8-8, the following examples illustrate the results of using different values for `nptsmax` when you enter

```
[xdata ydata errworst] = fixpt_look1_func_approx(funcstr, ...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,nptsmax);
```

The results for three different settings for `nptsmax` are as follows:

- `nptsmax = 33;` — The function creates the lookup table with 33 points having power of two spacing, as in Example 3.
- `nptsmax = 21;` — Because the `errmax` and `nptsmax` conditions cannot be met with power of two spacing, the function creates the lookup table with 20 points having even spacing, as in Example 5.
- `nptsmax = 16;` — Because the `errmax` and `nptsmax` conditions cannot be met with either power of two or even spacing, the function creates the lookup table with 16 points having unrestricted spacing, as in Example 1.

Comparison of Example Results

The following table summarizes the results for the examples. Note that when you specify `errmax`, even spacing requires more data points than unrestricted, and power of two spacing requires more points than even spacing.

Example	Options	Spacing	Worst-Case Error	Number of Points in Table
1	`errmax=2^-10`	`'unrestricted'`	`2^-10`	16
2	`nptsmax=21`	`'unrestricted'`	`2^-10.933`	21

Example	Options	Spacing	Worst-Case Error	Number of Points in Table
3	`errmax=2^-10`	`'even'`	`2^-10.0844`	20
4	`nptsmax=21`	`'even'`	`2^-10.2209`	21
5	`errmax=2^-10`	`'pow2'`	`2^-11.3921`	33
6	`nptsmax=21`	`'pow2'`	`2^-9.627`	17

Summary for Using Lookup Table Approximation Functions

The following steps summarize how to use the lookup table approximation functions:

1. Define:
   a. The ideal function to approximate
   b. The range, `xmin` to `xmax`, over which to find X and Y data
   c. The fixed-point implementation: data type, scaling, and rounding method
   d. The maximum acceptable error, the maximum number of points, and the spacing
2. Run the `fixpt_look1_func_approx` function to generate X and Y data.
3. Use the `fixpt_look1_func_plot` function to plot the function and error between the ideal and approximated functions using the selected X and Y data, and to calculate the error and the number of points used.
4. Vary input criteria, such as `errmax`, `nptsmax`, and `spacing`, to produce sets of X and Y data that generate functions with varying worst-case error, number of points required, and spacing.
5. Compare results of the number of points required and maximum absolute error from various runs to choose the best set of X and Y data.

Effects of Spacing on Speed, Error, and Memory Usage

In this section...

Criteria for Comparing Types of Breakpoint Spacing

The sections that follow compare implementations of lookup tables that use breakpoints whose spacing is uneven, even, and power of two. The comparison focuses on:

- Execution speed of commands
- Rounding error during interpolation
- The amount of read-only memory (ROM) for data
- The amount of ROM for commands

This comparison is valid only when the breakpoints are not tunable. If the breakpoints are tunable in the generated code, all three cases generate the same code. For a summary of the effects of breakpoint spacing on execution speed, error, and memory usage, see "Summary of the Effects of Breakpoint Spacing" on page 8-29.

Model That Illustrates Effects of Breakpoint Spacing

This comparison uses the model `fxpdemo_approx_sin`. Three fixed-point lookup tables appear in this model. All three tables approximate the function `sin(2*pi*u)` over the first quadrant and achieve a worst-case error of less

than `2^-8`. However, they have different restrictions on their breakpoint spacing.

You can use the model `fxpdemo_approx`, which `fxpdemo_approx_sin` opens, to generate Simulink Coder code (Simulink Coder software license required). The sections that follow present several segments of generated code to emphasize key differences.

To open the model, type at the MATLAB prompt:

```
fxpdemo_approx_sin
```

Data ROM Required for Each Lookup Table

This section looks at the data ROM required by each of the three spacing options.

Uneven Case

Uneven spacing requires both Y data points and breakpoints:

```
int16_T  yuneven[8];
uint16_T xuneven[8];
```

The total bytes used is 32.

Even Case

Even spacing requires only Y data points:

```
int16_T yeven[10];
```

The total bytes used is 20. The breakpoints are not explicitly required. The code uses the spacing between the breakpoints, and might use the smallest and largest breakpoints. At most, three values related to the breakpoints are necessary.

Power of Two Case

Power of two spacing requires only Y data points:

```
int16_T ypow2[17];
```

The total bytes used is 34. The breakpoints are not explicitly required. The code uses the spacing between the breakpoints, and might use the smallest and largest breakpoints. At most, three values related to the breakpoints are necessary.

Determination of Out-of-Range Inputs

In all three cases, you must guard against the chance that the input is less than the smallest breakpoint or greater than the biggest breakpoint. There can be differences in how occurrences of these possibilities are handled. However, the differences are generally minor and are normally not a key factor in deciding to use one spacing method over another. The subsequent sections assume that out-of-range inputs are impossible or have already been handled.

How the Lookup Tables Determine Input Location

This section describes how the three fixed-point lookup tables determine where the current input is relative to the breakpoints.

Uneven Case

Unevenly-spaced breakpoints require a general-purpose algorithm such as a binary search to determine where the input lies in relation to the breakpoints. The following code provides an example:

```
iLeft = 0;
iRght = 7; /* number of breakpoints minus 1 */

while ( ( iRght - iLeft ) > 1 )
{
  i = ( iLeft + iRght ) >> 1;

if ( uAngle < xuneven[i] )
  {
    iRght = i;
  }
  else
  {
    iLeft = i;
  }
```

```
}
```

The while loop executes up to log2(N) times, where N is the number of breakpoints.

Even Case

Evenly-spaced breakpoints require only one step to determine where the input lies in relation to the breakpoints:

```
iLeft = uAngle / 455U;
```

The divisor `455U` represents the spacing between breakpoints. In general, the dividend would be `(uAngle - SmallestBreakPoint)`. In this example, the smallest breakpoint is zero, so the code optimizes out the subtraction.

Power of Two Case

Power of two spaced breakpoints require only one step to determine where the input lies in relation to the breakpoints:

```
iLeft = uAngle >> 8;
```

The number of shifts is 8 because the breakpoints have spacing `2^8`. The smallest breakpoint is zero, so `uAngle` replaces the general case of `(uAngle - SmallestBreakPoint)`.

Comparison

To determine where the input lies with respect to the breakpoints, the unevenly-spaced case requires much more code than the other two cases. This code requires additional command ROM. If many lookup tables share the binary search algorithm as a function, you can reduce this ROM penalty. Even if the code is shared, the number of clock cycles required to determine the location of the input is much higher for the unevenly-spaced cases than the other two cases. If the code is shared, function-call overhead decreases the speed of execution a little more.

In the evenly-spaced case and the power of two spaced case, you can determine the location of the input with a single line of code. The evenly-spaced case uses a general integer division. The power of two case uses a shift instead of general division because the divisor is an exact power of two. Without

knowing the specific processor, you cannot be certain that a shift is better than division.

Many processors can implement division with a single assembly language instruction, so the code will be small. However, this instruction often takes many clock cycles to complete. Many processors do not provide a division instruction. Division on these processors occurs through repeated subtractions. This process is slow and requires a lot of machine code, but this code can be shared.

Most processors provide a way to do logical and arithmetic shifts left and right. A key difference is whether the processor can do N shifts in one instruction (barrel shift) or requires N instructions that shift one bit at a time. The barrel shift requires less code. Whether the barrel shift also increases speed depends on the hardware that supports the operation.

The compiler can also complicate the comparison. In the previous example, the command `uAngle >> 8` essentially takes the upper 8 bits in a 16-bit word. The compiler can detect this situation and replace the bit shifts with an instruction that takes the bits directly. If the number of shifts is some other value, such as 7, this optimization would not occur.

Interpolation for Each Lookup Table

In theory, you can calculate the interpolation with the following code:

```
y = ( yData[iRght] - yData[iLeft] ) * ( u - xData[iLeft] ) ...
    / ( xData[iRght] - xData[iLeft] ) + yData[iLeft]
```

The term `(xData[iRght] - xData[iLeft])` is the spacing between neighboring breakpoints. If this value is constant, due to even spacing, some simplification is possible. If spacing is not just even but also a power of two, significant simplifications are possible for fixed-point implementations.

Uneven Case

For the uneven case, one possible implementation of the ideal interpolation in fixed point is as follows:

```
xNum  = uAngle         - xuneven[iLeft];
xDen  = xuneven[iRght] - xuneven[iLeft];
```

```
yDiff = yuneven[iRght] - yuneven[iLeft];

MUL_S32_S16_U16( bigProd, yDiff, xNum );

  DIV_NZP_S16_S32_U16_FLOOR( yDiff, bigProd, xDen );

  yUneven = yuneven[iLeft] + yDiff;
```

The multiplication and division routines are not shown here. These routines can be complex and depend on the target processor. For example, these routines look different for a 16-bit processor than for a 32-bit processor.

Even Case

Evenly-spaced breakpoints implement interpolation using slightly different calculations than the uneven case. The key difference is that the calculations do not directly use the breakpoints. When the breakpoints are not required in ROM, you can save a lot of memory:

```
xNum  = uAngle - ( iLeft * 455U );

  yDiff = yeven[iLeft+1] - yeven[iLeft];

  MUL_S32_S16_U16( bigProd, yDiff, xNum );

  DIV_NZP_S16_S32_U16_FLOOR( yDiff, bigProd, 455U );

  yEven = yeven[iLeft] + yDiff;
```

Power of Two Case

Power of two spaced breakpoints implement interpolation using very different calculations than the other two cases. As in the even case, breakpoints are not used in the generated code and therefore not required in ROM:

```
lambda = uAngle & 0x00FFU;

  yPow2 = ypow2[iLeft)+1] - ypow2[iLeft];

  MUL_S16_U16_S16_SR8(yPow2,lambda,yPow2);
```

```
yPow2 += ypow2[iLeft];
```

This implementation has significant advantages over the uneven and even implementations:

- A bitwise AND combined with a shift right at the end of the multiplication replaces a subtraction and a division.
- The term `(u - xData[iLeft] ) / ( xData[iRght] - xData[iLeft])` results in no loss of precision, because the spacing is a power of two.

  In contrast, the uneven and even cases usually introduce rounding error in this calculation.

Summary of the Effects of Breakpoint Spacing

The following table summarizes the effects of breakpoint spacing on execution speed, error, and memory usage.

Parameter	Even Power of 2 Spaced Data	Evenly Spaced Data	Unevenly Spaced Data
Execution speed	The execution speed is the fastest. The position search and interpolation are the same as for evenly-spaced data. However, to increase the speed more, a bit shift replaces the position search, and a bit mask replaces the interpolation.	The execution speed is faster than that for unevenly-spaced data, because the position search is faster and the interpolation requires a simple division.	The execution speed is the slowest of the different spacings because the position search is slower, and the interpolation requires more operations.
Error	The error can be larger than that for unevenly-spaced data because approximating a function with nonuniform curvature requires more points to achieve the same accuracy.	The error can be larger than that for unevenly-spaced data because approximating a function with nonuniform curvature requires more points to achieve the same accuracy.	The error can be smaller because approximating a function with nonuniform curvature requires fewer points to achieve the same accuracy.
ROM usage	Uses less command ROM, but more data ROM.	Uses less command ROM, but more data ROM.	Uses more command ROM, but less data ROM.
RAM usage	Not significant.	Not significant.	Not significant.

The number of Y data points follows the expected pattern. For the same worst-case error, unrestricted spacing (uneven) requires the fewest data points, and power-of-two-spaced breakpoints require the most. However, the implementation for the evenly-spaced and the power of two cases does not

need the breakpoints in the generated code. This reduces their data ROM requirements by half. As a result, the evenly-spaced case actually uses less data ROM than the unevenly-spaced case. Also, the power of two case requires only slightly more ROM than the uneven case. Changing the worst-case error can change these rankings. Nonetheless, when you compare data ROM usage, you should always take into account the fact that the evenly-spaced and power of two spaced cases do not require their breakpoints in ROM.

The effort of determining where the current input is relative to the breakpoints strongly favors the evenly-spaced and power of two spaced cases. With uneven spacing, you use a binary search method that loops up to log2(N) times. With even and power of two spacing, you can determine the location with the execution of one line of C code. But you cannot decide the relative advantages of power of two versus evenly spaced without detailed knowledge of the hardware and the C compiler.

The effort of calculating the interpolation favors the power of two case, which uses a bitwise AND operation and a shift to replace a subtraction and a division. The advantage of this behavior depends on the specific hardware, but you can expect an advantage in code size, speed, and also in accuracy. The evenly-spaced case calculates the interpolation with a minor improvement in efficiency over the unevenly-spaced case.

9

Automatic Data Typing

Automatic Data Typing Methods

You can use the Fixed-Point Tool to select data types automatically for your model using the following methods.

Automatic Data Typing Method	Advantages	Disadvantages
Using simulation minimum and maximum values	• Useful if you know the inputs to use for the model. • You do not need to specify any design range information.	• Not always feasible to collect full simulation range. • Simulation might take a very long time.
Using design minimum and maximum values	You can use this method if the model contains blocks that range analysis does not support. However, if possible, use simulation data to propose data types.	• Design range often available only on some input and output signals. • Can propose data types only for signals with specified design minimum and maximum values.
Using derived minimum and maximum values	You do not have to simulate multiple times to ensure that simulation data covers the full intended operating range.	• Derivation might take a very long time.

Before Using the Fixed-Point Tool to Propose Data Types for Your Simulink Model

Before you use the Fixed-Point Tool to autoscale your Simulink model, consider how automatic data typing affects your model:

- The Fixed-Point Tool proposes new data types for the fixed-point data types in your model. If you choose to apply the proposed data types, the Fixed-Point Tool changes the data types in your model. Before using the Fixed-Point Tool, back up your model and workspace variables to ensure that you can recover your original data type settings and capture the fixed-point instrumentation and data type override settings using the Shortcut Editor.

  For more information, see "Best Practices for Using the Fixed-Point Tool to Propose Data Types for Your Simulink Model" on page 9-5.

- Ensure that you can update diagram successfully before proposing data types. Sometimes, changing the data types in your model results in subsequent update diagram errors. It is good practice to test update diagram again immediately before and after applying data type proposals. This practice enables you to fix any errors before making further modifications to your model.

  For more information, see "Updating a Block Diagram" in the Simulink documentation.

- The Fixed-Point Tool alerts you to potential issues with proposed data types for each object in your model:

  - If the Fixed-Point Tool detects that the proposed data type introduces data type errors when applied to an object, the tool marks the object with an error, . You must inspect this proposal and fix the problem in the Simulink model. After fixing the problem, rerun the simulation and generate a proposal again to confirm that you have resolved the issue.

    For more information, see "Examining Results to Resolve Conflicts" on page 9-22.

  - If the Fixed-Point Tool detects that the proposed data type poses potential issues for an object, the tool marks the object with a yellow caution, . Review the proposal before accepting it.

- If the Fixed-Point Tool detects that the proposed data type poses no issues for an object, the tool marks the object with a green check, .

Caution The Fixed-Point Tool does not detect all potential data type issues. If the Fixed-Point Tool does not detect any issues for your model, it is still possible to experience subsequent data type propagation issues. For more information, see "Models That Might Cause Data Type Propagation Errors" on page 9-8.

Best Practices for Using the Fixed-Point Tool to Propose Data Types for Your Simulink Model

Start with a Known Working Simulink Model

Before you begin automatic data typing, verify that update diagram succeeds for your model. To update the diagram, press Ctrl+D. If update diagram fails, before automatic data typing to propose data types, fix the failure in your model.

Back Up Your Simulink Model

Before using the Fixed-Point Tool, always back up your Simulink model and associated workspace variables.

Backing up your model provides a fallback in case of error and a baseline for testing and validation.

Capture the Current Fixed-Point Instrumentation and Data Type Override Settings

Before changing these settings, use the Fixed-Point Tool Shortcut Editor to create a shortcut for these settings. Creating a shortcut allows you to revert to the original model settings. For more information, see "How to Capture Current Model Settings Using the Shortcut Editor" on page 6-11.

Convert Individual Subsystems

Convert individual subsystems in your model one at a time. This practice facilitates debugging by isolating the source of fixed-point issues. For example, see "Debugging a Fixed-Point Model" on page 6-12.

Isolate the System Under Conversion

If you encounter data type propagation issues with a particular subsystem during the conversion, isolate this subsystem by placing Data Type Conversion blocks on the inputs and outputs of the system. The Data Type Conversion block converts an input signal of any Simulink data type to the data type

and scaling you specify for its **Output data type** parameter. This practice enables you to continue automatic data typing for the rest of your model.

Use Lock Output Data Type Setting When Necessary

You can prevent the Fixed-Point Tool from replacing the current data type. Use the **Lock output data type setting against changes by the fixed-point tools** parameter that is available on many blocks. The default setting allows for replacement. Use this setting when:

- You already know the fixed-point data types that you want to use for a particular block.

  For example, the block is modeling a real-world component. Set up the block to allow for known hardware limitations, such as restricting outputs to integer values.

  Explicitly specify the output data type of the block and select **Lock output data type setting against changes by the fixed-point tools**.

- You are debugging a model and know that a particular block accepts only certain input signal data types.

  Explicitly specify the output data type of upstream blocks and select **Lock output data type setting against changes by the fixed-point tools**.

Save Simulink Signal Objects Before Closing Your Model

If your model contains Simulink signal objects and you accept proposed data types, the Fixed-Point Tool automatically applies the changes to the signal objects. However, the Fixed-Point Tool does not automatically save changes that it makes to Simulink signal objects. To preserve changes, before closing your model, save the Simulink signal objects in your workspace and model.

Test Update Diagram Failure

Test update diagram immediately after applying data type proposals. If update diagram fails, take one of the following actions:

- Use the failure information to fix the errors in your model. After fixing the errors, test update diagram again.

- If you are unable to fix the errors, restore your backed up model. After restoring the model, try to fix the errors by, for example, locking output data type settings, isolating the system, as described in the preceding sections. After addressing the errors, test update diagram again.

Models That Might Cause Data Type Propagation Errors

When the Fixed-Point Tool proposes changes to the data types in your model, it alerts you to potential issues with the proposed data types for objects in the model. If the Fixed-Point Tool alerts you to data type errors, you must diagnose the errors and fix the problems. For more information, see "Examining Results to Resolve Conflicts" on page 9-22.

The Fixed-Point Tool does not detect all potential data type issues. If the Fixed-Point Tool does not report any issues for your model, it is still possible to experience subsequent data type propagation errors. Before you use the Fixed-Point Tool, back up your model to ensure that you can recover your original data type settings. For more information, see "Best Practices for Using the Fixed-Point Tool to Propose Data Types for Your Simulink Model" on page 9-5.

The following models are likely to cause data type propagation issues.

Model Uses...	Fixed-Point Tool Behavior	Data Type Propagation Issue
Buses	Does not detect the minimum, maximum, data type, and initial value information for bus objects and does not use them for automatic data typing.	Fixed-Point Tool might propose data types that are inconsistent with the data types for the bus object or generate proposals that cause overflows.
Simulink parameter objects	Does not consider any data type information for Simulink parameter objects and does not use them for automatic data typing.	Fixed-Point Tool might propose data types that are inconsistent with the data types for the parameter object or generate proposals that cause overflows.

Model Uses...	Fixed-Point Tool Behavior	Data Type Propagation Issue
User-defined S-functions	Cannot detect the operation of user-defined S-functions.	• The user-defined S-function accepts only certain input data types. The Fixed-Point Tool cannot detect this requirement and proposes a different data type upstream of the S-function. Update diagram fails on the model due to data type mismatch errors. • The user-defined S-function specifies certain output data types. The Fixed-Point Tool is not aware of this requirement and does not use it for automatic data typing. Therefore, the tool might propose data types that are inconsistent with the data types for the S-function or generate proposals that cause overflows.
User-defined masked subsystems	Has no knowledge of the masked subsystem workspace and therefore cannot take this subsystem into account when proposing data types.	Fixed-Point Tool might propose data types that are inconsistent with the requirements of the masked subsystem, particularly if the subsystem uses mask initialization. The proposed data types might cause data type mismatch errors or overflows.

Model Uses...	Fixed-Point Tool Behavior	Data Type Propagation Issue
Linked subsystems	Does not include linked subsystems when proposing data types.	Data type mismatch errors might occur at the linked subsystem boundaries.
MATLAB Function blocks	Does not propose data types for MATLAB Function blocks.	Fixed-Point Tool might propose data types that are inconsistent with the requirements of the MATLAB Function blocks. The proposed data types might cause data type mismatch errors or overflows.
Model reference	Does not propose data types for referenced models.	Data type propagation errors might occur at the referenced model's boundaries.

Automatic Data Typing Using Simulation Data

In this section...

Workflow for Automatic Data Typing Using Simulation Data

Workflow for Automatic Data Typing Using Simulation Data	
1. Prerequisites	
1.1	Open your model in Simulink and set it up for use with the Fixed-Point Tool.See Prerequisites.
1.2	From the Simulink **Tools** menu, select **Fixed-Point Tool** to open the Fixed-Point Tool.
2. Prepare the model for conversion **Note** If you do not have a floating-point model, skip this step.	
2.1	In the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem of interest.

Workflow for Automatic Data Typing Using Simulation Data	
2.2	In the **Fixed-point preparation for selected system** pane, click **Fixed-Point Advisor**.
2.3	Use the Fixed-Point Advisor to prepare the model for conversion. See Chapter 12, "Fixed-Point Advisor Reference".
3. Run the model to gather floating-point benchmark	
3.1	Enable signal logging for system or subsystem of interest: • In the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem. • Right-click to open the context menu. • Use the **Enable Signal Logging** option to enable signal logging as necessary. For more information, see "Signal Logging Options" in the `fxptdlg` reference.
3.2	In the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem for which you want a data type proposal.
3.3	In the **Shortcuts to set up runs** pane, click the **Model-wide double override and full instrumentation** button to set: • **Data type override** to `Double` • **Data type override applies to** to `All numeric types` • **Fixed-point instrumentation mode** to `Minimums, maximums and overflows` • The run name (in the **Data collection** pane **Store results in run** field) to `DoubleOverride` Using these settings, the Fixed-Point Tool performs a global override of the fixed-point data types and scaling using double-precision numbers, thus avoiding quantization effects. During simulation, the tool logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem in the run `DoubleOverride`.

Workflow for Automatic Data Typing Using Simulation Data	
	Note Data type override does not apply to `boolean` or enumerated data types.
3.4	Click the Fixed-Point Tool **Start simulation** button to run the simulation. The Fixed-Point Tool highlights any simulation results that have issues, such as overflows or saturations.
4 Propose data types	
4.1	In the **Automatic data typing for selected system Settings** pane, select either **Propose fraction lengths for specified word lengths** or **Propose word lengths for specified fraction lengths**. **Tip** If these options are not visible, use the **Configure** link to display them.
4.2	If you have safety margins to apply, set **Percent safety margin for design and derived min/max** and **Percent safety margin for simulation min/max** as necessary. For more information, see "Safety margin for simulation min/max (%)" and "Safety margin for design and derived min/max (%)" in the `fxptdlg` reference.

Workflow for Automatic Data Typing Using Simulation Data	
4.3	Click the **Propose fraction lengths** or **Propose word lengths** button DT. **Note** This action does not write the proposed data types to the model. If there are conflicts in your model, the Fixed-Point Tool displays the **Result Details** dialog box. If you do not see this warning, there are no conflicts in your model, skip the next step.
4.4	Click **OK** on the **Result Details** dialog box to close it.
5. Examine results to resolve conflicts	
5.1	Use the **Show** option on the Fixed-Point Tool toolbar to filter the results to show **Conflicts with proposed data types**. The Fixed-Point Tool alerts you to potential issues with the proposed data types by displaying a block icon annotated with red, yellow, and green symbols.
5.2	The proposed data type will introduce data type errors if applied to this object. You **must** inspect this result and fix the problem in the Simulink model. To review the error and correct the problem: **1** Select the result, right-click and select **Highlight Block In Model** from the context menu to identify which block has a conflict. **2** Click the **Show details for selected result** button to open the **Result Details** dialog box. **3** Use the advice provided in the **Needs Attention** section of the **Result Details** dialog box to resolve the conflict by fixing the problem in the Simulink model.

Workflow for Automatic Data Typing Using Simulation Data	
5.3	The proposed data type poses potential issues for this object. You should review this proposal. Review the **Result Details** for the warnings and correct the problem if necessary.
5.4	You have changed the Simulink model, so the benchmark data is not up to date. Click the Fixed-Point Tool **Start** button to rerun the simulation. The Fixed-Point Tool warns you that you have not applied proposals. Click the **Ignore and Simulate** button to continue.
5.5	Click the **Propose fraction lengths** or **Propose word lengths** button to generate a data type proposal, DT.
5.6	Use the **Show** option on the Fixed-Point Tool toolbar to filter the results to show **All results**.
6. Apply proposed data types	
6.1	Examine each result. For more information about a particular result, select the result then click the **Show details for selected result** button to display the **Result Details** dialog box.
6.2	If you do **not** want to accept the proposal for a result, in the Fixed-Point Tool **Contents** pane, clear the **Accept** checkbox for that result.
6.3	Click the **Apply accepted fraction lengths** or **Apply accepted word lengths** button to write the proposed data types to the model. **Note** If you have not fixed all the warnings in the model, the Fixed-Point Tool displays a warning dialog box.
7. Verify new settings	

Workflow for Automatic Data Typing Using Simulation Data	
7.1	In the **Shortcuts to set up runs** pane, click the **Model-wide no override and full instrumentation** button to use the locally specified data type settings. This sets: • **Data type override** to `Use local settings` • **Fixed-point instrumentation mode** to `Minimums, maximums and overflows` • The run name (in the **Data collection** pane **Store results in run** field) to `NoOverride` Using these settings, the Fixed-Point Tool simulates the model using the new fixed-point settings and logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem in the run `NoOverride`.
7.2	Click the Fixed-Point Tool **Start** button to run the simulation.
7.3	Compare the ideal results stored in the **DoubleOverride** run with the fixed-point results in the **NoOverride** run.: **1** In the **Contents** pane, select a result that has logged signal data. These results are annotated with the icon. **2** Click the **Difference Plot of Signal** to view the difference between the two runs for the selected result. **Note** If you have more than two runs, in the **Difference Plot Selector** dialog box, select the two runs that you want to compare.

Prerequisites For Automatic Data Typing Using Simulation Data

To use the Fixed-Point Tool to generate data type proposals for your model based on simulation minimum and maximum values only, you must first set up your model in Simulink.

1. Open your model in Simulink.

2. Select **Simulation > Normal** in the Simulink menu to ensure the model runs in **Normal** mode.

   Note The Fixed-Point Tool supports only **Normal** mode.

3. If you are using design minimum and maximum range information, add this information to blocks.

   You specify a design range for model objects using parameters typically titled **Output minimum** and **Output maximum**. See "Blocks That Allow Signal Range Specification" in *Simulink User's Guide* for a list of blocks that permit you to specify these values.

4. Specify fixed-point data types for blocks and signals in your model. For blocks with the **Data Type Assistant**, use the **Calculate Best-Precision Scaling** button to calculate best-precision scaling automatically. For more information, see "Specifying Fixed-Point Data Types with the Data Type Assistant" on page 1-25.

   Note If you have a floating-point model, use the Fixed-Point Advisor to prepare your model for conversion to an equivalent fixed-point representation. To learn more about the Fixed-Point Advisor, see Chapter 5, "Fixed-Point Advisor".

5. You can choose to lock some blocks against automatic data typing by selecting the **Lock output data type setting against changes by the fixed-point tools** parameter. If an object's **Lock output data type**

setting against changes by the fixed-point tools parameter is selected, the tool does not propose data types for that object.

6 From the Simulink **Edit** menu, select **Update Diagram** to perform parameter range checking for all blocks in the model.

   If update diagram fails, use the failure information to fix the errors in your model. After fixing the errors, test update diagram again. If you are unable to fix the errors, restore your backed up model.

7 Back up the model as a fallback in case of error and a baseline for testing and validation.

8 Create a shortcut to capture the initial fixed-point instrumentation and data type override settings. For more information, see "How to Capture Current Model Settings Using the Shortcut Editor" on page 6-11.

Preparing the Model for Conversion

If you have a floating-point model or subsystem, first use the Fixed-Point Advisor to prepare the model or subsystem for conversion to fixed point. The Fixed-Point Advisor checks the model against fixed-point guidelines and provides advice about unsupported blocks. You need only do this preparation once.

1 From the Simulink **Tools** menu, select **Fixed-Point Tool**.

2 In the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem of interest.

3 In the **Fixed-point preparation for selected system** pane, click **Fixed-Point Advisor**.

4 Use the Fixed-Point Advisor to prepare the model for conversion. See Chapter 12, "Fixed-Point Advisor Reference".

Running the Model to Gather a Floating-Point Benchmark

You first run the model with a global override of the fixed-point data types using double-precision numbers to avoid quantization effects. This provides a floating-point benchmark that represents the ideal output. The Simulink

software logs the signal logging results to the MATLAB workspace. The Fixed-Point Tool displays the simulation results including minimum and maximum values that occur during the run.

To Gather a Floating-Point Benchmark

1. Launch Fixed-Point Tool.

   From the Simulink **Tools** menu, select **Fixed-Point Tool**.

2. Enable signal logging for the system or subsystem of interest. Using the Fixed-Point Tool you can enable signal logging for multiple signals simultaneously. For more information, see "Signal Logging Options" in the `fxptdlg` Reference.

   To enable signal logging:

   a. Select the system or subsystem in the Fixed-Point Tool **Model Hierarchy** pane.

   b. Right-click to open the context menu.

   c. Use the **Enable Signal Logging** option to enable signal logging, as necessary.

   The **Contents** pane of the Fixed-Point Tool displays an antenna icon next to items that have signal logging enabled.

   Note You can only plot results for signals that have signal logging enabled.

3. In the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem for which you want a proposal.

4. In the **Shortcuts to set up runs** pane, click the **Model-wide double override and full instrumentation** button to set:

   - **Data type override** to `Double`
   - **Data type override applies to** to `All numeric types`
   - **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`

- The run name (in the **Data collection** pane **Store results in run** field) to `DoubleOverride`

Using these settings, the Fixed-Point Tool performs a global override of the fixed-point data types with double-precision data types, thus avoiding quantization effects. During simulation, the tool logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem in the run `DoubleOverride`.

Note Data type override does not apply to `boolean` or enumerated data types.

---

5 Click the Fixed-Point Tool **Simulate** button  to run the simulation.

The Fixed-Point Tool highlights any simulation results that have issues, such as overflows or saturations.

Proposing Data Types

The Fixed-Point Tool proposes data types for model objects that specify fixed-point data types unless an object's **Lock output data type setting against changes by the fixed-point tools** parameter is selected or the data types are using inheritance rules.

When proposing data types, the Fixed-Point Tool collects the following types of range data for model objects:

- Design minimum or maximum values — You specify a design range for model objects using parameters typically titled **Output minimum** and **Output maximum**. See "Blocks That Allow Signal Range Specification" in the *Simulink User's Guide* for a list of blocks that permit you to specify these values.
- Simulation minimum or maximum values — When simulating a system whose **Fixed-point instrumentation mode** parameter specifies `Minimums, maximums and overflows`, the Fixed-Point Tool logs the minimum and maximum values generated by model objects. For more information about the **Fixed-point instrumentation mode** parameter, see the documentation for the `fxptdlg` function in the *Simulink Reference*.

- Derived minimum or maximum values — When deriving minimum and maximum values for a selected system, the Fixed-Point Tool uses the design minimum and maximum values that you specify for the model to derive range information for signals in your model. For more information, see Chapter 10, "Range Analysis".

The Fixed-Point Tool uses available range data to calculate data type proposals according to the following rules:

- Design minimum and maximum values take precedence over the simulation and derived range.

  The **Safety margin for design and derived min/max (%)** parameter specifies a range that differs from that defined by the design range. For example, a value of 20 specifies that a range of at least 20 percent larger is desired. A value of -10 specifies that a range of up to 10 percent smaller is acceptable. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the *Configure* link.

- The tool observes the derived range only when the **Derived min/max** option is selected. Otherwise, the tool ignores the derived range.

  The **Safety margin for design and derived min/max (%)** parameter specifies a range that differs from that defined by the derived range. For example, a value of 20 specifies that a range of at least 20 percent larger is desired. A value of -10 specifies that a range of up to 10 percent smaller is acceptable. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the *Configure* link.

- The tool observes the simulation range only when the **Simulation min/max** option is selected. Otherwise, the tool ignores the simulation range.

  The **Safety margin for simulation min/max (%)** parameter specifies a range that differs from that defined by the simulation range. For example, a value of 20 specifies that a range of at least 20 percent larger is desired. A value of -10 specifies that a range of up to 10 percent smaller is acceptable. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the *Configure* link.

To Propose Data Types

1. On the **Settings** pane, select either **Propose fraction lengths for specified word lengths** or **Propose word lengths for specified fraction lengths**.

2. To use simulation min/max information only, clear **Derived min/max**.

3. If you have safety margins to apply:

   a. Enter **Safety margin for design and derived min/max (%)**, if applicable. For example, enter 10 for a 10% safety margin. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the *Configure* link.

   b. Enter **Safety margin for simulation min/max (%)**, if applicable. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the *Configure* link.

4. Click the **Propose fraction lengths** or **Propose word lengths** button to generate a proposal, DT.

Note The Fixed-Point Tool does not alter your model when it proposes data types.

If there are conflicts in your model, the Fixed-Point Tool displays the **Result Details** dialog box.

If you do not see this warning, there are no conflicts in your model, go to "Applying Proposed Data Types" on page 9-27.

Examining Results to Resolve Conflicts

You can examine each proposal using the **Result Details** dialog box, which displays the rationale underlying the proposed data types. Also, this dialog box describes potential issues or errors, and it suggests methods for resolving them. To open the dialog box:

1. In the **Contents** pane, select an object that has proposed data types.

2 Click the **Show details for selected result** button .

The **Result Details** dialog box provides the following information about the proposed data types, as appropriate.

Summary

Details which run the result is in and the current data type specified for the selected object.

Proposed Data Type Summary

This section describes a proposal in terms of how it differs from the object's current data type. For cases when the Fixed-Point Tool does not propose data types, this section provides an explanation. For example, the data type might be locked against changes by the fixed-point tool.

Needs Attention

This section lists potential issues and errors associated with data type proposals. It describes the issues and suggests methods for resolving them. The dialog box uses the following icons to differentiate warnings from errors.

Indicates a warning message.

Indicates an error message.

Shared Data Type Summary

This section of the dialog box tells you that the selected object must share the same data type as other objects in the model because of data type propagation rules. For example, the inputs to a Merge block must have the same data type. Therefore, the outputs of blocks that connect to these inputs must share the same data type.

The dialog box provides a hyperlink that you can click to highlight the objects that share data types in the model. To clear this highlighting, from the model **View** menu, select **Remove Highlighting**.

The Fixed-Point Tool allocates an identification tag to objects that must share the same data type. The tool displays this identification tag in the **DTGroup** column for the object. To display only the objects that must share data types, from the Fixed-Point Tool main toolbar, select the **Show** option. For more information, see "Main Toolbar" in the `fxptdlg` reference documentation.

Constrained Data Type Summary

Some Simulink blocks are designed to accept only certain data types on some ports. This section of the dialog box tells you when a block that connects to the selected object has data type constraints that impact the proposed data type of the selected object. The dialog box lists the blocks that have data type constraints, provides details of the constrained data types, and links to the blocks in the model.

Data Type Details

This section provides a table that lists model object attributes that influence its data type proposal.

Item	Description
Currently Specified Data Type	Data type that an object currently specifies.
Proposed Data Type	Data type that the Fixed-Point Tool proposes for this object.
Proposed Representable Maximum	Maximum value that the proposed data type can represent.
Design Maximum	Design maximum value that an object specifies using, e.g., its **Output maximum** parameter.
Simulation Maximum	Maximum value that occurs during simulation.
Simulation Minimum	Minimum value that occurs during simulation.

Item	Description
Design Minimum	Design minimum value that an object specifies using, e.g., its **Output minimum** parameter.
Proposed Representable Minimum	Minimum value that the proposed data type can represent.

The table also includes a column titled **Percent Proposed Representable**. This column indicates the percentage of the proposed representable range that each value covers. Overflows occur when values lie outside this range.

Shared Values. When proposing data types, the Fixed-Point Tool attempts to satisfy data type requirements that model objects impose on one another. For example, the Sum block provides an option that requires all of its inputs to have the same data type. Consequently, the table might also list attributes of other model objects that impact the proposal for the selected object. In such cases, the table displays the following types of shared values:

- **Initial Values**

  Some model objects provide parameters that allow you to specify the initial values of their signals. For example, the Constant block includes a **Constant value** parameter that initializes the block output signal. The Fixed-Point Tool uses initial values to propose data types for model objects whose design and simulation ranges are unavailable. When data type dependencies exist, the tool considers how initial values impact the proposals for neighboring objects.

- **Model-Required Parameters**

  Some model objects require the specification of numeric parameters to compute the value of their outputs. For example, the **Table data** parameter of an n-D Lookup Table block specifies values that the block requires to perform a lookup operation and generate output. When proposing data types, the Fixed-Point Tool considers how this “model-required” parameter value impacts the proposals for neighboring objects.

To Examine the Results and Resolve Conflicts

1. Click **OK** on the **Result Details** dialog box to close the dialog box.

2 On the Fixed-Point Tool toolbar, use the **Show** option to filter the results to show **Conflicts with proposed data types**.

The Fixed-Point Tool lists its data type proposals in the **Contents** pane under the **ProposedDT** column. The tool alerts you to potential issues for each object in the list by displaying a green, yellow, or red icon.

The proposed data type poses no issues for this object.

The proposed data type poses potential issues for this object.

The proposed data type will introduce data type errors if applied to this object.

3 Review and fix each error.

a Select the error, right-click and select **Highlight Block In Model** from the context menu to identify which block has a conflict.

b Click the **Show details for selected result** button to open the **Result Details** dialog box.

c Use the advice provided in the **Needs Attention** section of the **Result Details** dialog box to resolve the conflict by fixing the problem in the Simulink model.

4 Review the **Result Details** for the warnings and correct the problem if necessary.

5 You have changed the Simulink model, so the benchmark data is not up to date. Click the Fixed-Point Tool **Start** button to rerun the simulation.

The Fixed-Point Tool warns you that you have not applied proposals. Click the **Ignore and Simulate** button to continue.

6 Click the **Propose fraction lengths** or **Propose word lengths** button to generate a proposal, DT.

7 Use the **Show** option on the Fixed-Point Tool toolbar to filter the results to show **All results**.

Applying Proposed Data Types

After reviewing the data type proposals, you are ready to apply the proposed data types to your model. The Fixed-Point Tool allows you to apply its data type proposals selectively to objects in your model. Use the **Accept** check box in the **Contents** pane to specify the proposals that you want to assign to model objects. The check box indicates the status of a proposal:

- ☑ The Fixed-Point Tool will apply the proposed data type to this object. By default, the tool selects the **Accept** check box when a proposal differs from the object's current data type.
- ☐ The Fixed-Point Tool will ignore the proposed data type and leave the current data type intact for this object.
- ▢ No proposal exists for this object. This occurs, for example, when the object specifies a data type inheritance rule or is locked against automatic data typing.

To Apply Proposed Data Types

1. Examine each result. For more information about a particular result, select the result and then click the **Show details for selected result** button to display the **Result Details** dialog box.

2. If you do **not** want to accept the proposal for a result, clear the **Accept** checkbox in the Fixed-Point Tool **Contents** pane for that result.

Tip The Fixed-Point Tool enables you to customize its proposals before applying them to your model. To do so, in the **Contents** pane, click a **ProposedDT** cell and edit the data type expression. See documentation for the `fixdt` function for information about specifying fixed-point data types.

---

3. Click the **Apply accepted fraction lengths** or **Apply accepted word lengths** button to write the proposed data types to the model.

Note If you have not fixed all the warnings in the model, the Fixed-Point Tool displays a warning dialog box.

Verifying New Settings

After applying proposed data types to your model, you simulate the model using the fixed-point data types that you applied.

To Verify the New Settings

1. In the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem for which you want a proposal.

2. In the **Shortcuts to set up runs** pane, click the **Model-wide no override and full instrumentation** button to use the locally specified data type settings. This shortcut also sets the run name to `NoOverride`

3. Click the Fixed-Point Tool **Start** button to run the simulation.

4. Compare the ideal results stored in the **DoubleOverride** run with the fixed-point results in the **NoOverride** run:

   a. In the **Contents** pane, select a result that has logged signal data. These results are annotated with the icon.

   b. Click the **Difference Plot of Signal** to view the difference between the fixed-point and double override runs for the selected result.

   Note If you have more than two runs, in the **Difference Plot Selector** dialog box, select the two runs that you want to compare.

Automatic Data Typing of Simulink Signal Objects

Fixed-Point Tool can propose new data types for Simulink signal objects in the base or model workspace. If you accept the proposed data types, the Fixed-Point Tool will apply the new data types to the Simulink signal objects automatically.

Caution The Fixed-Point Tool does not save the changes to the signal object. You must save the changes before closing the model.

If you delete or manipulate a signal object in the base workspace after automatic data typing, you must rerun the automatic data typing.

Automatic Data Typing Using Derived Minimum and Maximum Values

In this section...

Prerequisites for Automatic Data Typing Using Derived Minimum and Maximum Values

The Fixed-Point Tool uses range analysis to derive minimum and maximum values for objects in your model.

Range analysis:

- Requires a Simulink Fixed Point license.
- Does not run on Mac® platforms.

For more information, see Chapter 10, "Range Analysis".

Workflow for Automatic Data Typing Using Derived Data

Workflow for Automatic Data Typing Using Derived Data	
1. Prerequisites	
1.1	Open your model in Simulink and set it up for use with the Fixed-Point Tool.
2. Prepare model prior to automatic data typing using derived data	
2.1	If you have a floating-point model, use the Fixed-Point Advisor to prepare the model for conversion to fixed point. From the Simulink **Tools** menu, select **Fixed-Point Tool**.
2.2	In the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem of interest.
2.3	In the **Fixed-point preparation for selected system** pane, click **Fixed-Point Advisor**.
2.4	Use the Fixed-Point Advisor to prepare the model for conversion. See Chapter 12, “Fixed-Point Advisor Reference”.
3.Derive minimum and maximum values	
3.1	In the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem of interest.
3.2	In the **Settings for selected system** pane, set **Data type override** to `Double`.
3.3	Optionally, in the **Data collection** pane **Store results in run** field, specify a run name. Specifying a unique run name avoids overwriting results from previous runs.

Workflow for Automatic Data Typing Using Derived Data	
3.4	In the Fixed-Point Tool, click **Derive min/max values for model**. The analysis runs and tries to derive range information for objects in the selected system. If the analysis successfully derives range data for the model, the Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the selected system. (See "Viewing Derived Range Information in the Fixed-Point Tool" on page 10-10.) Review the results and then skip the following two steps; you are ready to propose data types. If the analysis fails, examine the error messages and resolve the issues.
4. Resolve any issues	
4.1	If the model contains blocks that range analysis does not support, the Fixed-Point Tool generates an error. To fix the error, examine the error message and replace the unsupported blocks.
4.2	If the analysis cannot derive range data because the model contains conflicting design range information, the Fixed-Point Tool generates an error and highlights the objects that have no derived ranges. To fix this error, examine the design ranges specified in the model to identify inconsistent design specifications and modify them to make them consistent.
4.3	If the analysis cannot derive range data for an object because there is insufficient design range information specified on the model, the Fixed-Point Tool highlights the results for the object. Examine the model to determine which design range information is missing. Specify additional design range information as necessary.
5. Derive minimum and maximum values	

Workflow for Automatic Data Typing Using Derived Data	
5.1	After resolving range analysis issues, derive range information again. **Note** You might need to repeat the previous two steps until the analysis is successful.
6 Propose data types	
6.1	In the **Automatic data typing for selected system Settings** pane, select either **Propose fraction lengths for specified word lengths** or **Propose word lengths for specified fraction lengths**, as applicable. **Tip** If these options are not visible, use the **Configure** link to display them.
6.2	If you have safety margins to apply, set **Percent safety margin for design and derived min/max**. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the *Configure* link. For more information, see "Safety margin for simulation min/max (%)" in the `fxptdlg` reference.
6.3	Click the **Propose fraction lengths** or **Propose word lengths** button DT. **Note** This action does not write the proposed data types to the model. If there are conflicts in your model, the Fixed-Point Tool displays the **Result Details** dialog box. If you do not see this warning, there are no conflicts in your model, skip the next step.
7. Examine results to resolve conflicts	

Workflow for Automatic Data Typing Using Derived Data	
7.1	Click **OK** on the **Result Details** dialog box to close it.
7.2	Use the **Show** option on the Fixed-Point Tool toolbar to filter the results to show **Conflicts with proposed data types**. The Fixed-Point Tool alerts you to potential issues for each result by displaying a block icon annotated with red, yellow, and green symbols.
7.3	The proposed data types will introduce data type errors if applied to this object. You **must** inspect this result and fix the problem in the Simulink model. To review the error and correct the problem: • Select the result, right-click and select **Highlight Block In Model** from the context menu to identify which block has a conflict. • Click the **Show details for selected result** button to open the **Result Details** dialog box. • Use the advice provided in the **Needs Attention** section of the **Result Details** dialog box to resolve the conflict by fixing the problem in the Simulink model.
7.4	The proposed data type poses potential issues for this object. You should review this proposal. Review the **Result Details** for the warnings and correct the problem if necessary.
7.5	You have changed the Simulink model, so the benchmark data is not up to date. Click the Fixed-Point Tool **Start** button to rerun the simulation. The Fixed-Point Tool warns you that you have not applied proposals. Click the **Ignore and Simulate** button to continue.
7.6	Click the **Propose fraction lengths** or **Propose word lengths** button to generate a data type proposal, DT.
7.7	Use the **Show** option on the Fixed-Point Tool toolbar to filter the results to show **All results**.

Workflow for Automatic Data Typing Using Derived Data	
8. Apply proposed data types	
8.1	Examine each result. For more information about a particular result, select the result then click the **Show details for selected result** button to display the **Result Details** dialog box.
8.2	If you do **not** want to accept the proposal for a result, clear the **Accept** checkbox in the Fixed-Point Tool **Contents** pane for that result.
8.3	Click the **Apply accepted fraction lengths** or **Apply accepted word lengths** button to write the proposed data types to the model. **Note** If you have not fixed all the warnings in the model, the Fixed-Point Tool displays a warning dialog box.
9. Update Diagram	
9.1	From the model's **Edit** menu, select **Update Diagram**. After applying the data types to the model, update diagram to check for data type propagation issues. If update diagram fails, use the failure information to fix the errors in your model. After fixing the errors, test update diagram again. If you are unable to fix the errors, restore your backed up model.

Prerequisites for Automatic Data Typing Using Derived Data

To use the Fixed-Point Tool to generate data type proposals for your model based on derived minimum and maximum values only, you must first set up your model in Simulink.

1. Open your model in Simulink.
2. Select **Simulation > Normal** in the Simulink menu to ensure the model runs in **Normal** mode.

Note The Fixed-Point Tool supports only **Normal** mode.

3 To autoscale using derived data, you **must** specify design minimum and maximum values on at least the model inputs. The range analysis tries to narrow the derived range by using all the specified design ranges in the model. The more design range information you specify, the more likely the range analysis is to succeed. As the analysis is performed, it derives new range information for the model and then attempts to use this new information together with the specified ranges to derive ranges for the remaining objects in the model. For this reason, the analysis results might depend on block priorities because these priorities determine the order in which the software analyzes the blocks.

   You specify a design range for model objects using parameters typically titled **Output minimum** and **Output maximum**. See "Blocks That Allow Signal Range Specification" in *Simulink User's Guide* for a list of blocks that permit you to specify these values.

4 Specify fixed-point data types for blocks and signals in your model. For blocks with the **Data Type Assistant**, use the **Calculate Best-Precision Scaling** button to calculate best-precision scaling automatically. For more information, see "Specifying Fixed-Point Data Types with the Data Type Assistant" on page 1-25.

5 You can choose to lock some blocks against automatic data typing by selecting the **Lock output data type setting against changes by the fixed-point tools** parameter. If an object's **Lock output data type setting against changes by the fixed-point tools** parameter is selected, the tool does not propose data types for that object.

6 From the Simulink **Edit** menu, select **Update Diagram** to perform parameter range checking for all blocks in the model.

   If update diagram fails, use the failure information to fix the errors in your model. After fixing the errors, test update diagram again. If you are unable to fix the errors, restore your backed up model.

7 Back up the model as a fallback in case of error and a baseline for testing and validation.

8. Create a shortcut to capture the initial fixed-point instrumentation and data type override settings. For more information, see "How to Capture Current Model Settings Using the Shortcut Editor" on page 6-11.

Preparing Model Prior to Automatic Data Typing Using Derived Data

If you have a floating-point model, use the Fixed-Point Advisor to prepare the model for conversion to fixed point. The Fixed-Point Advisor:

- Checks the model against fixed-point guidelines.
- Identifies unsupported blocks.
- Removes output data type inheritance from blocks..
- Allows you to promote simulation minimum and maximum values to design minimum and maximum values. This capability is useful if you have not specified design ranges and you have simulated the model with inputs that cover the full intended operating range. For more information, see "Specify block minimum and maximum values" on page 12-33.
- Runs simulation range detection diagnostics. When preparing the model for automatic data typing using derived data, you can complete the preparation without setting up signal logging and creating a simulation reference run. However, creating at least one simulation run is useful for early error detection. Simulating the model helps to verify that the design minimum and maximum values specified on the model are correct and that the model conforms to modeling guidelines.

To learn more about the Fixed-Point Advisor, see Chapter 5, "Fixed-Point Advisor".

Deriving Minimum and Maximum Values

1. In the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem of interest.
2. In the **Settings for selected system** pane, set **Data type override** to `Double`.
3. In the Fixed-Point Tool, click **Derive min/max values for model**.

The analysis runs and tries to derive range information for objects in the selected system.

If the analysis successfully derives range data for the model, the Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the selected system. (See "Viewing Derived Range Information in the Fixed-Point Tool" on page 10-10.) Review the results before proposing data types.

If the analysis fails, examine the error messages and resolve the issues. See "Resolving Range Analysis Issues" on page 9-39.

Resolving Range Analysis Issues

The following table shows the different types of range analysis issues and the next steps that you should take to resolve them.

Analysis Results	Next Steps	For More Information
The analysis fails because the model contains blocks that it does not support. The Fixed-Point Tool generates an error.	To fix the error, review the error message information and replace the unsupported blocks.	"Model Compatibility with Range Analysis" on page 10-6
The analysis cannot derive range data because the model contains conflicting design range information. The Fixed-Point Tool generates an error.	To fix this error, examine the design ranges specified in the model to identify inconsistent design specifications and modify them to make them consistent.	"Fixing Design Range Conflicts" on page 10-22
The analysis cannot derive range data for an object because there is insufficient design range information specified on the model. The Fixed-Point Tool highlights the results for the object.	Examine the model to determine which design range information is missing.	"Providing More Design Range Information" on page 10-20

Proposing Data Types

The Fixed-Point Tool proposes data types for model objects that specify fixed-point data types unless an object's **Lock output data type setting against changes by the fixed-point tools** parameter is selected or the data types are using inheritance rules. You set up the tool to either propose fraction lengths for specified word lengths or to propose word lengths for specified fraction lengths. For more information, see "Proposing Fraction Lengths" on page 9-48 and "Proposing Word Lengths" on page 9-64.

When generating data type proposals, the Fixed-Point Tool collects the following types of range data for model objects:

- Design minimum or maximum values — You specify a design range for model objects using parameters typically titled **Output minimum** and **Output maximum**. See "Blocks That Allow Signal Range Specification" in the *Simulink User's Guide* for a list of blocks that permit you to specify these values.
- Simulation minimum or maximum values — When simulating a system whose **Fixed-point instrumentation mode** parameter specifies `Minimums, maximums and overflows`, the Fixed-Point Tool logs the minimum and maximum values generated by model objects. For more information about the **Fixed-point instrumentation mode** parameter, see the documentation for the `fxptdlg` function in the *Simulink Reference*.
- Derived minimum or maximum values — When deriving minimum and maximum values for a selected system, the Fixed-Point Tool uses the design minimum and maximum values that you specify for the model to derive range information for signals in your model. For more information, see Chapter 10, "Range Analysis".

  For models that contain floating-point operations, range analysis might report a range that is slightly larger than expected due to rounding errors in the analysis. Automatic data typing bases its proposal on this slightly larger derived range. To avoid this issue, use the safety margin for design and derived min/max.

The Fixed-Point Tool uses available range data to calculate data type proposals according to the following rules:

- Design minimum and maximum values take precedence over the simulation and derived range.

  The **Safety margin for design and derived min/max (%)** parameter specifies a range that differs from that defined by the design range. For example, a value of 20 specifies that a range of at least 20 percent larger is desired. A value of -10 specifies that a range of up to 10 percent smaller is acceptable. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the *Configure* link.

- The tool observes the derived range only when the **Derived min/max** option is selected. Otherwise, the tool ignores the derived range.

  The **Safety margin for design and derived min/max (%)** parameter specifies a range that differs from that defined by the derived range. For example, a value of 20 specifies that a range of at least 20 percent larger is desired. A value of -10 specifies that a range of up to 10 percent smaller is acceptable. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the *Configure* link.

- The tool observes the simulation range only when the **Simulation min/max** option is selected. Otherwise, the tool ignores the simulation range.

  The **Safety margin for simulation min/max (%)** parameter specifies a range that differs from that defined by the simulation range. For example, a value of 20 specifies that a range of at least 20 percent larger is desired. A value of -10 specifies that a range of up to 10 percent smaller is acceptable. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the *Configure* link.

To Propose Data Types

1. On the **Settings** pane, select either **Propose fraction lengths for specified word lengths** or **Propose word lengths for specified fraction lengths**.

2. If you have a safety margin to apply, set **Safety margin for design and derived min/max (%)**. For example, enter 10 for a 10% safety margin. If this parameter is not visible in the **Automatic data typing for selected system** pane, click the *Configure* link.

3. Click the **Propose fraction lengths** or **Propose word lengths** button to generate a proposal, DT.

Note The Fixed-Point Tool does not alter your model when it proposes data types.

If there are conflicts in your model, the Fixed-Point Tool displays the **Result Details** dialog box.

If you do not see this warning, there are no conflicts in your model, go to "Applying Proposed Data Types" on page 9-27.

Examining Results to Resolve Conflicts

You can examine each data type proposal using the **Result Details** dialog box, which displays the rationale underlying the proposal. Also, this dialog box describes potential issues or errors, and it suggests methods for resolving them. To open the dialog box:

1. In the **Contents** pane, select an object that has proposed data types.
2. Click the **Show details for selected result** button .

The **Result Details** dialog box provides the following information about the proposed data type, as appropriate.

Summary

Details which run the result is in and the current data type specified for the selected object.

Proposed Data Type Summary

This section describes a data type proposal in terms of how it differs from the object's current data type. For cases when the Fixed-Point Tool does not propose data types, this section provides an explanation. For example, the data type might be locked against changes by the fixed-point tool.

Needs Attention

This section lists potential issues and errors associated with data type proposals. It describes the issues and suggests methods for resolving them. The dialog box uses the following icons to differentiate warnings from errors.

Indicates a warning message.

Indicates an error message.

Shared Data Type Summary

This section of the dialog box tells you that the selected object must share the same data type as other objects in the model because of data type propagation rules. For example, the inputs to a Merge block must have the same data type. Therefore, the outputs of blocks that connect to these inputs must share the same data type.

The dialog box provides a hyperlink that you can click to highlight the objects that share data types in the model. To clear this highlighting, from the model **View** menu, select **Remove Highlighting**.

The Fixed-Point Tool allocates an identification tag to objects that must share the same data type. The tool displays this identification tag in the **DTGroup** column for the object. To display only the objects that must share data types, from the Fixed-Point Tool main toolbar, select the **Show** option. For more information, see "Main Toolbar" in the `fxptdlg` reference documentation.

Constrained Data Type Summary

Some Simulink blocks are designed to accept only certain data types on some ports. This section of the dialog box tells you when a block that connects to the selected object has data type constraints that impact the proposed data type of the selected object. The dialog box lists the blocks that have data type constraints, provides details of the constrained data types, and links to the blocks in the model.

Data Type Details

This section provides a table that lists a model object attributes that influence its data type proposal.

Item	Description
Currently Specified Data Type	Data type that an object currently specifies.
Proposed Data Type	Data type that the Fixed-Point Tool proposes for this object.

Item	Description
Proposed Representable Maximum	Maximum value that the proposed data type can represent.
Design Maximum	Design maximum value that an object specifies using, e.g., its **Output maximum** parameter.
Simulation Maximum	Maximum value that occurs during simulation.
Simulation Minimum	Minimum value that occurs during simulation.
Design Minimum	Design minimum value that an object specifies using, e.g., its **Output minimum** parameter.
Proposed Representable Minimum	Minimum value that the proposed data type can represent.

The table also includes a column titled **Percent Proposed Representable**. This column indicates the percentage of the proposed representable range that each value covers. Overflows occur when values lie outside this range.

Shared Values. When proposing data types, the Fixed-Point Tool attempts to satisfy data type requirements that model objects impose on one another. For example, the Sum block provides an option that requires all of its inputs to have the same data type. Consequently, the table might also list attributes of other model objects that impact the data type proposal for the selected object. In such cases, the table displays the following types of shared values:

- **Initial Values**

  Some model objects provide parameters that allow you to specify the initial values of their signals. For example, the Constant block includes a **Constant value** parameter that initializes the block output signal. The Fixed-Point Tool uses initial values to propose data types for model objects whose design and simulation ranges are unavailable. When data type dependencies exist, the tool considers how initial values impact the proposals for neighboring objects.

- **Model-Required Parameters**

  Some model objects require the specification of numeric parameters to compute the value of their outputs. For example, the **Table data** parameter of an n-D Lookup Table block specifies values that the block requires to perform a lookup operation and generate output. When proposing data types, the Fixed-Point Tool considers how this "model-required" parameter value impacts the proposals for neighboring objects.

To Examine the Results and Resolve Conflicts

1. Click **OK** on the **Result Details** dialog box to close the dialog box.

2. On the Fixed-Point Tool toolbar, use the **Show** option to filter the results to show **Conflicts with proposed data types**.

   The Fixed-Point Tool lists its data type proposals in the **Contents** pane under the **ProposedDT** column. The tool alerts you to potential issues for each object in the list by displaying a green, yellow, or red icon.

   The proposed data type poses no issues for this object.

   The proposed data type poses potential issues for this object.

   The proposed data type will introduce data type errors if applied to this object.

3. Review and fix each error.

   a. Select the error, right-click and select **Highlight Block In Model** from the context menu to identify which block has a conflict.

   b. Click the **Show details for selected result** button to open the **Result Details** dialog box.

   c. Use the advice provided in the **Needs Attention** section of the **Result Details** dialog box to resolve the conflict by fixing the problem in the Simulink model.

4. Review the **Result Details** for the warnings and correct the problem if necessary.

5 You have changed the Simulink model, so the benchmark data is not up to date. Click the Fixed-Point Tool **Start** button to rerun the simulation.

   The Fixed-Point Tool warns you that you have not applied proposals. Click the **Ignore and Simulate** button to continue.

6 Click the **Propose fraction lengths** or **Propose word lengths** button to generate a data type proposal, DT.

7 Use the **Show** option on the Fixed-Point Tool toolbar to filter the results to show **All results**.

Applying Proposed Data Types

After reviewing the data type proposals, you are ready to apply the proposed data types to your model. The Fixed-Point Tool allows you to apply its data type proposals selectively to objects in your model. Use the **Accept** check box in the **Contents** pane to specify the proposals that you want to assign to model objects. The check box indicates the status of a proposal:

- ☑ The Fixed-Point Tool will apply the proposed data type to this object. By default, the tool selects the **Accept** check box when a proposal differs from the object's current data type.
- ☐ The Fixed-Point Tool will ignore the proposed data type and leave the current data type intact for this object.
- ☐ No proposal exists for this object. This occurs, for example, when the object specifies a data type inheritance rule or is locked against automatic data typing.

To Apply Proposed Data Types

1 Examine each result. For more information about a particular result, select the result and then click the **Show details for selected result** button to display the **Result Details** dialog box.

2 If you do **not** want to accept the proposal for a result, clear the **Accept** checkbox in the Fixed-Point Tool **Contents** pane for that result.

Tip The Fixed-Point Tool enables you to customize its proposals before applying them to your model. To do so, in the **Contents** pane, click a **ProposedDT** cell and edit the data type expression. See documentation for the `fixdt` function for information about specifying fixed-point data types.

---

3 Click the **Apply accepted fraction lengths** or **Apply accepted word lengths** button to write the proposed data types to the model.

Note If you have not fixed all the warnings in the model, the Fixed-Point Tool displays a warning dialog box.

Proposing Fraction Lengths

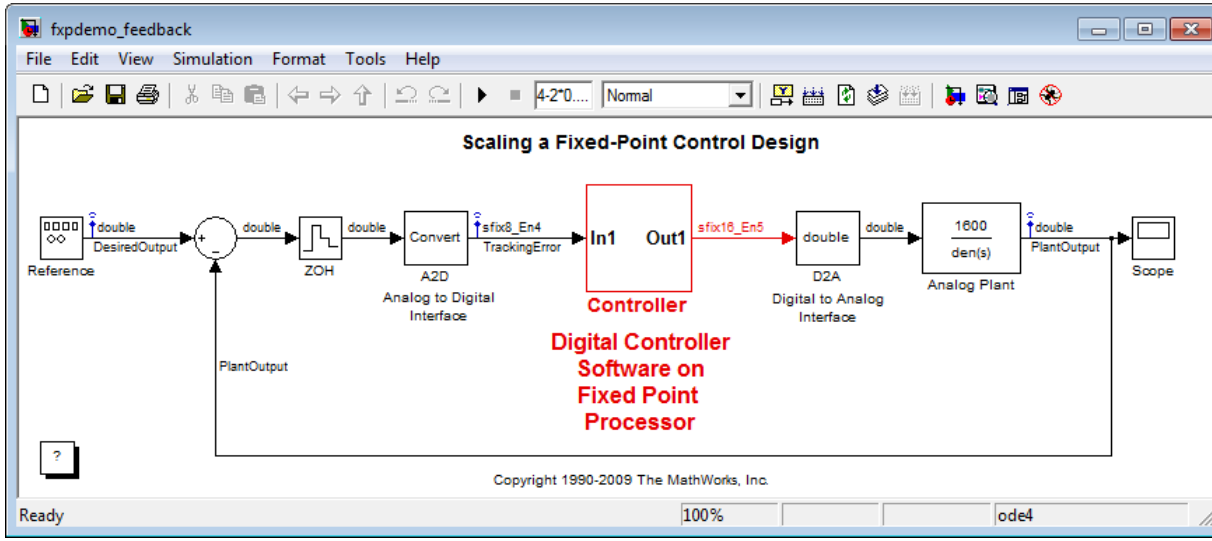
In this section...

About the Feedback Controller Example Model

Opening the Feedback Controller Model

To open the Simulink feedback design model used for this tutorial, type `fxpdemo_feedback` at the MATLAB command line.

The feedback design model opens.



About the Feedback Controller Model

The Simulink model of the feedback design consists of the following blocks and subsystems:

- **Reference**

  This Signal Generator block generates a continuous-time reference signal. It is configured to output a square wave.

- **Sum**

  This Sum block subtracts the plant output from the reference signal.

- **ZOH**

  The Zero-Order Hold block samples and holds the continuous signal. This block is configured so that it quantizes the signal in time by 0.01 seconds.

- **Analog to Digital Interface**

  The analog to digital (A/D) interface consists of a Data Type Conversion block that converts a `double` to a fixed-point data type. It represents any hardware that digitizes the amplitude of the analog input signal. In the real world, its characteristics are fixed.

- **Controller**

  The digital controller is a subsystem that represents the software running on the hardware target. Refer to "Digital Controller Realization" on page 9-52.

- **Digital to Analog Interface**

  The digital to analog (D/A) interface consists of a Data Type Conversion block that converts a fixed-point data type into a `double`. It represents any hardware that converts a digitized signal into an analog signal. In the real world, its characteristics are fixed.

- **Analog Plant**

  The analog plant is described by a transfer function, and is controlled by the digital controller. In the real world, its characteristics are fixed.

- **Scope**

  The model includes a Scope block that displays the plant output signal.

Simulation Setup

To set up this kind of fixed-point feedback controller simulation, typically you perform the following steps:

1. Identify all design components.

   In the real world, there are design components with fixed characteristics (the hardware) and design components with characteristics that you can change (the software). In this feedback design, the main hardware components are the A/D hardware, the D/A hardware, and the analog plant. The main software component is the digital controller.

2. Develop a theoretical model of the plant and controller.

   For the feedback design used in this tutorial, the plant is characterized by a transfer function. The characteristics of the plant are unimportant for this tutorial, and are not discussed.

   The digital controller model used in this tutorial is described by a z-domain transfer function and is implemented using a direct-form realization.

3. Evaluate the behavior of the plant and controller.

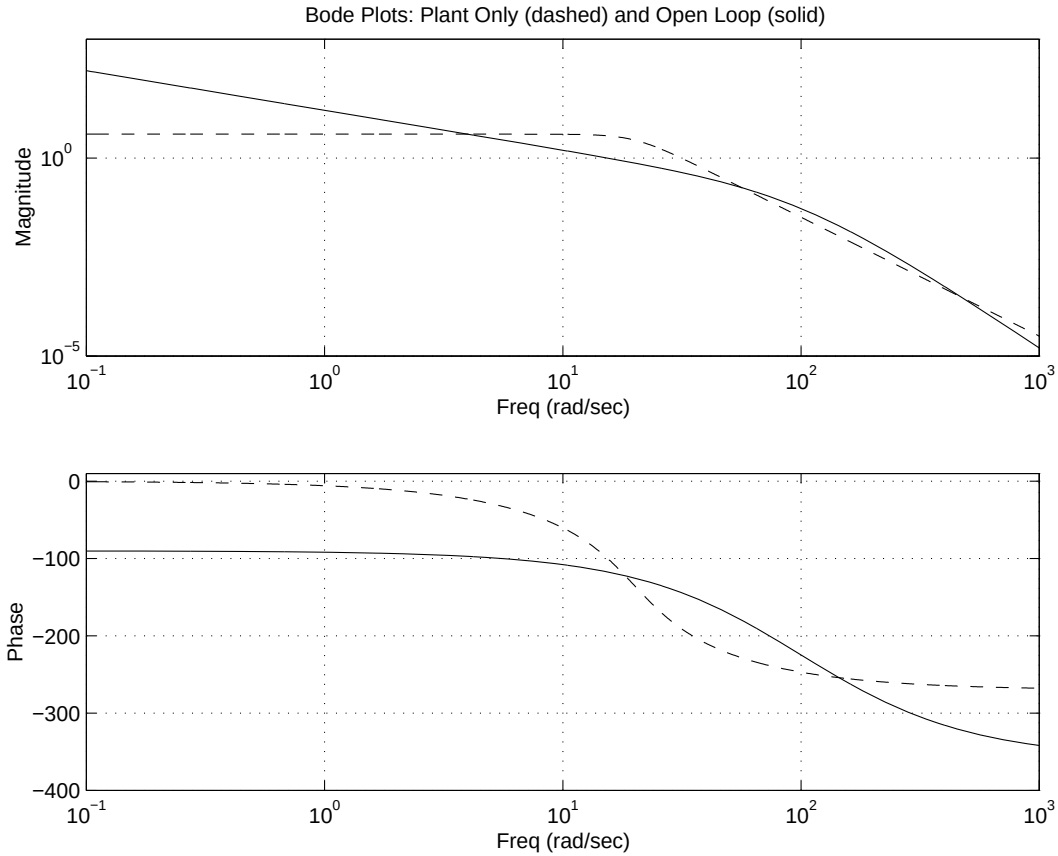
   You evaluate the behavior of the plant and the controller with a Bode plot. This evaluation is idealized, because all numbers, operations, and states are double-precision.

4. Simulate the system.

   You simulate the feedback controller design using Simulink and Simulink Fixed Point software. Of course, in a simulation environment, you can treat all components (software *and* hardware) as though their characteristics are not fixed.

Idealized Feedback Design

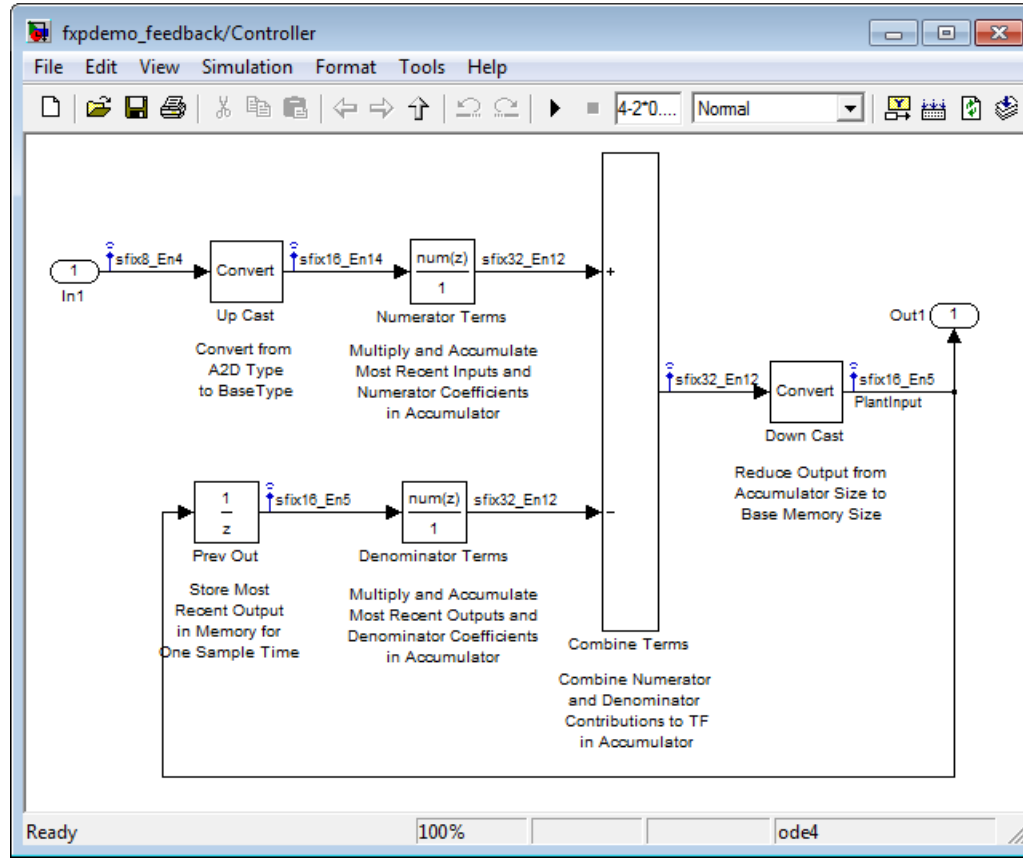
Open loop (controller and plant) and plant-only Bode plots for the "Scaling a Fixed-Point Control Design" demo are shown in the following figure. The open loop Bode plot results from a digital controller described in the idealized world of continuous time, double-precision coefficients, storage of states, and math operations.



The plant and controller design criteria are not important for the purposes of this tutorial. The Bode plots were created using workspace variables produced by a script named `preload_feedback.m`.

Digital Controller Realization

In this simulation, the digital controller is implemented using the fixed-point direct form realization shown in the following diagram. The hardware target is a 16-bit processor. Variables and coefficients are generally represented using 16 bits, especially if these quantities are stored in ROM or global RAM. Use of 32-bit numbers is limited to temporary variables that exist briefly in CPU registers or in a stack.



The realization consists of these blocks:

- **Up Cast**

  Up Cast is a Data Type Conversion block that connects the A/D hardware with the digital controller. It pads the output word of the A/D hardware with trailing zeros to a 16-bit number (the base data type).

- **Numerator Terms** and **Denominator Terms**

  Each of these Discrete FIR Filter blocks represents a weighted sum carried out in the CPU target. The word size and precision used in the calculations reflect those of the accumulator. Numerator Terms multiplies and

accumulates the most recent inputs with the FIR numerator coefficients. Denominator Terms multiples and accumulates the most recent delayed outputs with the FIR denominator coefficients. The coefficients are stored in ROM using the base data type. The most recent inputs are stored in global RAM using the base data type.

- **Combine Terms**

  Combine Terms is a Sum block that represents the accumulator in the CPU. Its word size and precision are twice that of the RAM (double bits).

- **Down Cast**

  Down Cast is a Data Type Conversion block that represents taking the number from the CPU and storing it in RAM. The word size and precision are reduced to half that of the accumulator when converted back to the base data type.

- **Prev Out**

  Prev Out is a Unit Delay block that delays the feedback signal in memory by one sample period. The signals are stored in global RAM using the base data type.

Direct Form Realization. The controller directly implements this equation,

$$y(k) = \sum_{i=0}^{N} b_i u(k-1) - \sum_{i=1}^{N} a_i y(k-1),$$

where

- $u(k-1)$ represents the *input* from the previous time step.
- $y(k)$ represents the current output, and $y(k-1)$ represents the output from the previous time step.
- b_i represents the FIR numerator coefficients.
- a_i represents the FIR denominator coefficients.

The first summation in $y(k)$ represents multiplication and accumulation of the most recent inputs and numerator coefficients in the accumulator. The second summation in $y(k)$ represents multiplication and accumulation of the most

recent outputs and denominator coefficients in the accumulator. Because the FIR coefficients, inputs, and outputs are all represented by 16-bit numbers (the base data type), any multiplication involving these numbers produces a 32-bit output (the accumulator data type).

Proposing Fraction Lengths for a Feedback Controller Using Simulation Range Data

Before You Begin

This example shows you how to use the Fixed-Point Tool to refine the scaling of fixed-point data types associated with a feedback controller model (see "About the Feedback Controller Example Model" on page 9-48). Although the tool enables multiple workflows for converting a digital controller described in ideal double-precision numbers to one realized in fixed-point numbers, this example demonstrates the following approach:

1. "Initial Guess at Scaling" on page 9-56. Run an initial "proof of concept" simulation using a reasonable guess at the fixed-point word size and scaling. This task illustrates how difficult it is to guess the best scaling.

2. "Data Type Override" on page 9-58. Perform a global override of the fixed-point data types using double-precision numbers. The Simulink software logs the simulation results to the MATLAB workspace, and the Fixed-Point Tool displays them.

3. "Automatic Data Typing" on page 9-59. Perform the automatic data typing procedure, which uses the double-precision simulation results to propose fixed-point scaling for appropriately configured blocks. The Fixed-Point Tool allows you to accept and apply the scaling proposals selectively. Afterward, you determine the quality of the results by examining the input and output of the model's analog plant.

To begin, open both the `fxpdemo_feedback` model and the Fixed-Point Tool.

Initial Guess at Scaling

Initial guesses for the scaling of each block are already specified in each block mask in the model. This task illustrates the difficulty of guessing the best scaling.

1. In the Fixed-Point Tool **Shortcuts to set up runs** pane, click the **Model-wide no override and full instrumentation** button to set:
   - **Data type override** to `Use local settings`. This option enables each of the model's subsystems to use its locally specified data type settings.
   - **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`.
   - The run name to `NoOverride`.
2. In the Fixed-Point Tool, click the **Simulate** button  .

   The Simulink software simulates the `fxpdemo_feedback` model. Afterward, the Fixed-Point Tool displays in its **Contents** pane the simulation results for each block that logged fixed-point data. By default, it displays the Simulation View of these results, you can customize this view by clicking **Show Details**. For more information about the standard views provided by the Fixed-Point Tool, see "Customizing the Contents Pane View" in the `fxptdlg` function reference. For more information about customizing views, see "The Model Explorer: Controlling Contents Using Views".

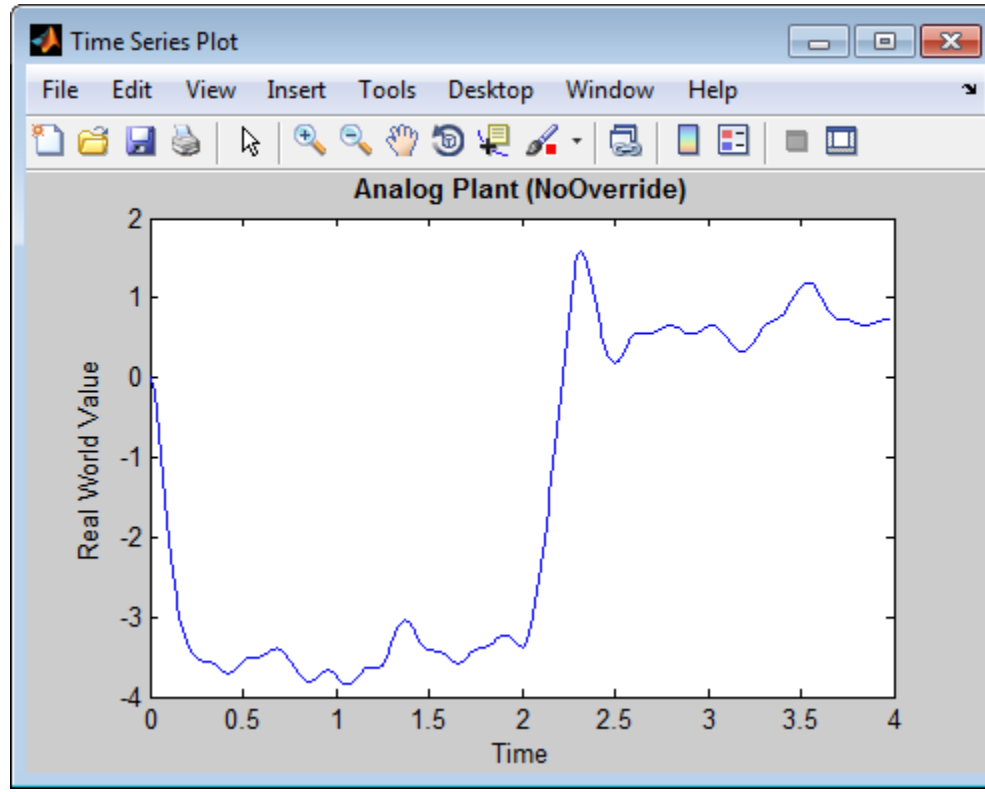
   The tool stores the results in the `NoOverride` run, denoted by the **NoOverride** label in the **Run** column. The Fixed-Point tool highlights the `Up Cast` block to indicate that there is an issue with this result. The **Saturations** column for this result shows that the block saturated 23 times, which indicates a poor guess for its scaling.

Tip You can use the **Show** option in the main toolbar to view only blocks that have Overflows.

---

3 In the **Contents** pane of the Fixed-Point Tool, select the Transfer Fcn block named `Analog Plant` and then click the **Plot of Signal** button .

The Fixed-Point Tool plots the signal associated with the plant output.



The preceding plot of the plant output signal reflects the initial guess at scaling. The Bode plot design sought to produce a well-behaved linear response for the closed-loop system. Clearly, the response is nonlinear. Significant quantization effects cause the nonlinear features. An important part of fixed-point design is finding a scaling that reduces quantization effects to acceptable levels.

Tip You can use the Fixed-Point Tool plotting tools to plot simulation results associated with logged signal data. To view a list of all logged signals, use the `Show` option in the main toolbar and select `Signal logging results`.

Data Type Override

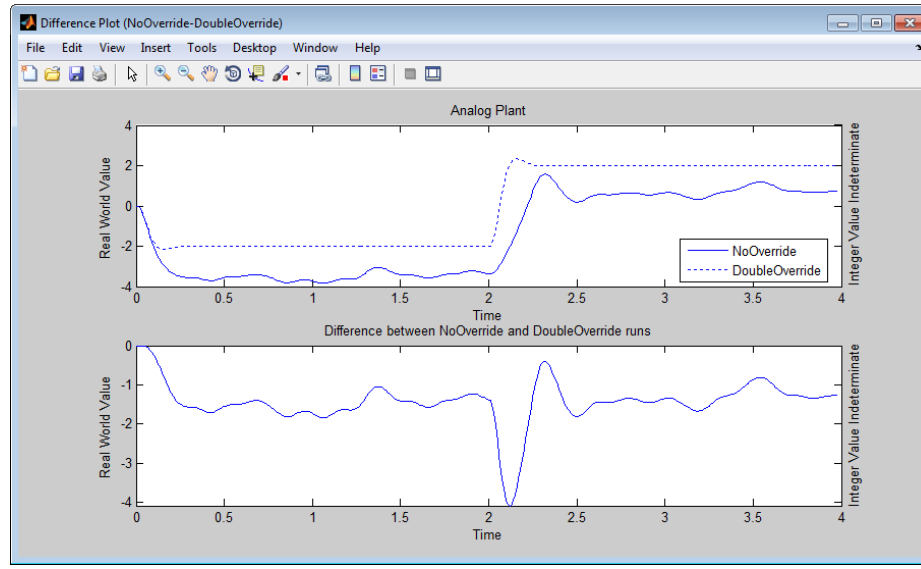
Data type override mode enables you to perform a global override of the fixed-point data types with double-precision data types, thereby avoiding quantization effects. The Fixed-Point Tool will use these simulation results when performing automatic scaling to propose higher fidelity fixed-point scaling.

1. In the Fixed-Point Tool **Shortcuts to set up runs** pane, click the **Model-wide double override and full instrumentation** button to set:
   - **Data type override** to `Double`
   - **Data type override applies to** to `All numeric types`
   - **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`
   - The run name (in the **Data collection** pane **Store results in run** field) to `DoubleOverride`
2. In the Fixed-Point Tool, click the **Simulate** button  .

   The Simulink software simulates the `fxpdemo_feedback` model in data type override mode and stores the results as the `DoubleOverride` run. Afterward, the Fixed-Point Tool displays in its **Contents** pane the **DoubleOverride** run results along with those of the **NoOverride** run that you generated previously (see "Initial Guess at Scaling" on page 9-56). The **SimDT** (simulation data type) column for the `DoubleOverride` run shows that the model's blocks used a `double` data type during simulation.
3. In the **Contents** pane of the Fixed-Point Tool, select the Transfer Fcn block named `Analog Plant` in the `NoOverride` run, and then click the **Difference Plot of Signal** button  .

The Fixed-Point Tool plots both the `DoubleOverride` and `NoOverride` versions of the signal associated with the plant output (upper axes), and plots the difference between the active and reference versions of that signal (lower axes). Compare the ideal (`double` data type) plant output signal with its fixed-point version.

Tip You can use the Zoom tool  to zoom in on an area of interest. By default, synchronized zooming is enabled for the difference plot. Zooming on either plot zooms both plots. For more information, see "Plot Interface" in the `fxptdlg` Reference.



Automatic Data Typing

Using the automatic data typing procedure, you can easily maximize the precision of the output data type while spanning the full simulation range. For a complex model, the absence of such a procedure can make achieving this goal tedious and time consuming.

Note The Fixed-Point Tool yields meaningful automatic data typing results when the maximum and minimum simulation values cover the full intended operating range of your design.

Perform automatic data typing for the `Controller` block. This block is a subsystem that represents software running on the target, and it requires optimization.

1. In the **Model Hierarchy** pane of the Fixed-Point Tool, select the `Controller` subsystem. In the **Automatic data typing for selected system** pane, click the *Configure* link. Ensure that **Simulation min/max** is selected for **Propose using information from design min/max and** then specify the **Safety margin for simulation min/max** parameter as `20`, and then click **Apply**.

   Because no design min/max information is supplied, the simulation min/max data that was collected during the simulation run is used for proposing data types. The **Safety margin for simulation min/max (%)** parameter value multiplies the "raw" simulation values by a factor of 1.2. Setting this parameter to a value greater than 1 decreases the likelihood that an overflow will occur when fixed-point data types are being used. For more information about how the Fixed-Point Tool calculates data type proposals, see "Proposing Data Types" on page 9-20.

   Because of the nonlinear effects of quantization, a fixed-point simulation will produce results that are different from an idealized, doubles-based simulation. Signals in a fixed-point simulation can cover a larger or smaller range than in a doubles-based simulation. If the range increases enough, overflows or saturations could occur. A safety margin decreases the likelihood of this happening, but it might also decrease the precision of the simulation.

2. In the Fixed-Point Tool,:

   a. Click the **Propose fraction lengths** button DT.

   b. In the **Propose Data Types** dialog box, select **DoubleOverride**, and then click **OK**.

The Fixed-Point Tool analyzes the scaling of all fixed-point blocks whose:

- **Lock output data type setting against changes by the fixed-point tools** parameter is not selected.
- **Output data type** parameter specifies a generalized fixed-point number.
- Data types are not inherited types.

The Fixed-Point Tool uses the minimum and maximum values stored in the `DoubleOverride` run to propose each block's scaling such that the precision is maximized while the full range of simulation values is spanned. The tool displays the proposed scaling in its **Contents** pane. Now, it displays the Autoscaling View to provide information, such as **ProposedDT**, **ProposedMin**, **ProposedMax**, which are relevant at this stage of the fixed-point conversion.

Tip You can use the **Show** option in the main toolbar to view the groups that must share data types. For more information, see `fxptdlg` in the Simulink Reference.

---

3. Review the scaling that the Fixed-Point Tool proposes. You can choose to accept the scaling proposal for each block by selecting the corresponding **Accept** check box in the **Contents** pane. By default, the Fixed-Point Tool accepts all scaling proposals that differ from the current scaling. For this example, ensure that the **Accept** check box associated with the `DoubleOverride` run is selected for each of the Controller subsystem's blocks.

4. In the Fixed-Point Tool, click the **Apply accepted fraction lengths** button  .

   The Fixed-Point Tool applies to the `Controller` subsystem's blocks the scaling proposals that you accepted in the previous step.

5. In the **Model Hierarchy** pane of the Fixed-Point Tool, select the `fxpdemo_feedback` system.

a In the **Shortcuts to set up runs** pane, click the **Model-wide no override and full instrumentation** button to use the locally specified data type settings.

b In the **Data collection** pane, set **Store results in run** to `FixedPoint` so that Fixed-Point Tool stores the results with a new run name and does not overwrite the results for the initial fixed-point set up. Storing the results in different runs allows you to compare the initial system behavior with the behavior of the autoscaled model.

6 In the Fixed-Point Tool, click **Simulate**.

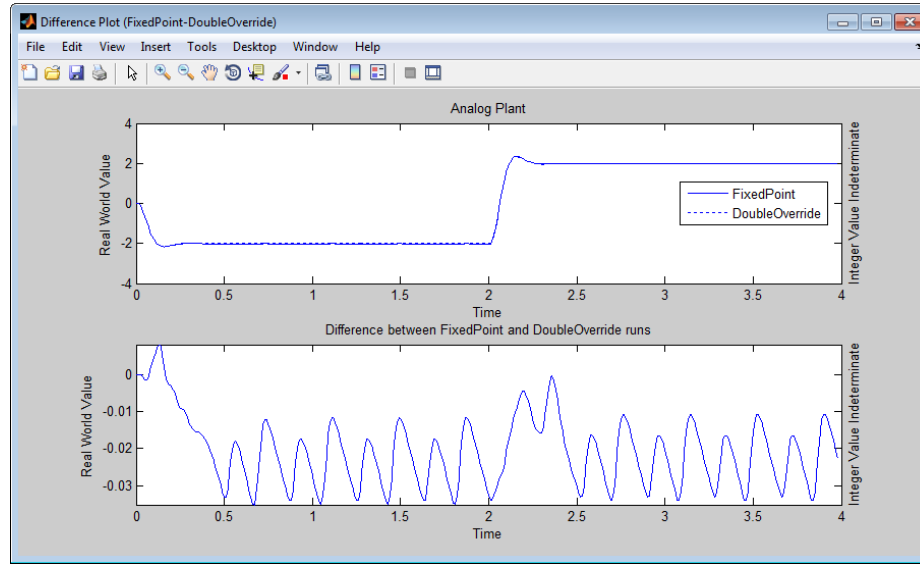
The Simulink software simulates the `fxpdemo_feedback` model using the new scaling that you applied earlier. Afterward, the Fixed-Point Tool displays in its **Contents** pane information about blocks that logged fixed-point data. The **SimDT** (simulation data type) column for the `FixedPoint` run shows that the Controller subsystem's blocks used fixed-point data types with the new scaling.

7 In the **Model Hierarchy** pane of the Fixed-Point Tool, select the `fxpdemo_feedback` system.

a In the **Contents** pane, select the Transfer Fcn block named `Analog Plant` for the `FixedPoint` run, and then click the **Difference Plot of Signal** button  .

b In the **Difference Plot Selector** dialog box, select `DoubleOverride`, and then click **OK**.

The Fixed-Point Tool plots the fixed-point and double override versions of the plant output signal, as well as their difference.



Tip Optionally, you can zoom in to view the steady-state region with greater detail. From the **Tools** menu of the figure window, select **Zoom In** and then drag the pointer to draw a box around the area you want to view more closely. For instance, the following figure shows a close-up view of a portion of the previous plot.

The plant output signal represented by the fixed-point run achieves a steady state, but a small limit cycle is present because of poor A/D design.

Proposing Word Lengths

In this section...

How the Fixed-Point Tool Proposes Word Lengths

To use the Fixed-Point Tool to propose word lengths, you must specify the target hardware and the fraction length requirements for data types in the model. Select the fraction lengths based on the precision required for the system that you are modeling. If you do not specify fraction lengths, the Fixed-Point Tool sets the fraction length to zero. The Fixed-Point Tool uses these specified fraction lengths to recommend the minimum word length for fixed-point data types in the selected model or subsystem to avoid overflow for the collected range information.

The proposal uses the following information:

- Design range information and range information that the Fixed-Point Tool or Fixed-Point Advisor collects. This collected range information can be either simulation range data or derived range data.
- The signedness and fraction lengths of data types specified for blocks, signal objects.
- The signedness and fraction lengths of the default data types specified in the Fixed-Point Tool or Fixed-Point Advisor.
- The embedded hardware implementation settings specified in the Configuration Parameters dialog box.

How the Fixed-Point Tool Uses Range Information

The Fixed-Point Tool determines whether to use different types of range information based on its availability and on the Fixed-Point Tool **Automatic data typing for selected system** > **Propose using information from design min/max and** pane **Derived min/max** and **Simulation min/max** settings.

Design range information always takes precedence over both simulation and derived range data. When there is no design range information, the Fixed-Point Tool uses the union of available simulation and derived range data. If you specify safety margins, the Fixed-Point Tool takes these into account.

For example, if a signal has a design range of `[-10,10]`, the Fixed-Point Tool uses this range for the proposal and ignores all simulation and derived range information. If you specify a safety margin of 10% for design range, the Fixed-Point Tool uses a range of `[-11,11]` for the proposal.

If the signal has no specified design information, but does have a simulation range of `[-8,8]` and a derived range of `[-2,2]`, the proposal uses the union of the ranges, `[-8,8]`. If you specify a safety margin of 50%, the proposal uses a range of `[-12, 12]`.

How the Fixed-Point Tool Uses Target Hardware Information

The Fixed-Point Tool calculates the ideal word length and then checks this length against the embedded hardware implementation settings for the target hardware. It uses the following rules:

Target Hardware	Ideal Word Length	Proposed Word Length	Notes
FPGA/ASIC	Ideal word length=<128	Ideal word length	None
	Ideal word length>128	128	Maximum word length is 128

Target Hardware	Ideal Word Length	Proposed Word Length	Notes
Embedded Processor	Ideal word length=< character bit length for the embedded processor (`char`)	`char`	Rounds up word length
	char <Ideal word length=< short bit length for the embedded processor (`short`)	`short`	Rounds up word length
	short<Ideal word length=< integer bit length for the embedded processor (`int`)	`int`	Rounds up word length
	int<Ideal word length=<long bit length for the embedded processor (`long`)	`long`	Rounds up word length
	Ideal word length>long bit length for the embedded processor	`long`	Maximum word length is the target hardware `long`

How to Propose Word Lengths

1. Specify the target hardware.
   a. In the model, select **Simulation > Configuration Parameters**.
   b. In the Configuration Parameters dialog box, select **Hardware Implementation**.

c On the **Hardware Implementation** pane, specify the **Device vendor** and **Device type**, then click **Apply**.

2 On the Fixed-Point Tool **Automatic data typing for selected system** pane, select **Propose word lengths for specified fraction lengths**. If you cannot see this option, click *Configure* to display more options.

3 In the same pane:

- To use simulation min/max information only, clear **Derived min/max**.
- To use derived min/max information only, clear **Simulation min/max**.

.

4 If you have safety margins to apply, in the same pane, set **Safety margin for design and derived min/max (%)** and **Safety margin for design and derived min/max (%)**, as applicable.

5 Click the **Propose word lengths** button to generate a proposal, DT.

Note The Fixed-Point Tool does not alter your model when it proposes data types.

If there are conflicts in your model, the Fixed-Point Tool displays the **Result Details** dialog box.

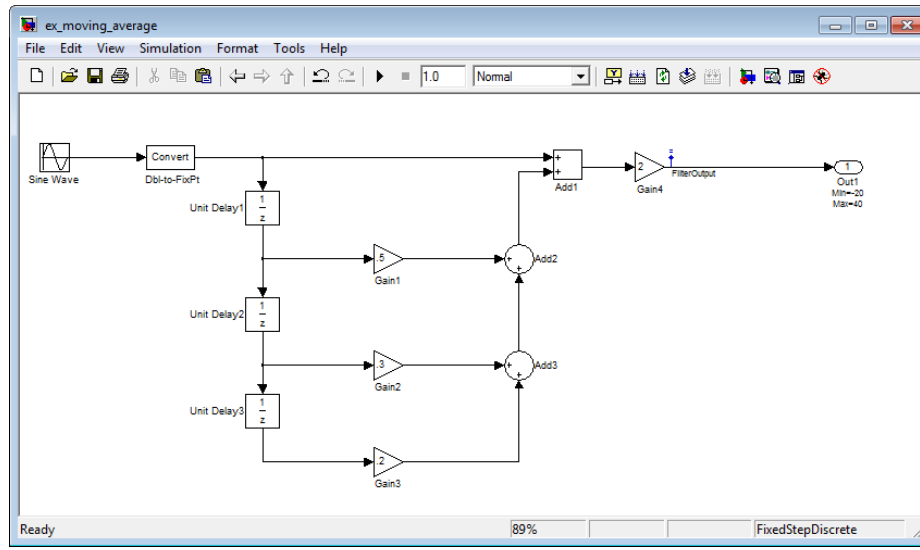
If you do not see this warning, there are no conflicts in your model. Review the proposed word lengths,

Proposing Word Lengths Based on Simulation Data

This example shows how to use the Fixed-Point Tool to propose word lengths for a model that implements a simple moving average algorithm. The model already uses fixed-point data types, but they are not optimal. Simulate the model and propose data types based on simulation data. To see how the target hardware affects the word length proposals, first set the target hardware to an embedded processor and propose word lengths. Then, set the target hardware to an FPGA and propose word lengths.

1 Open the `ex_moving_average` model. At the MATLAB command line, enter:

```
run(docpath(fullfile(docroot,'toolbox','fixpoint','examples','ex_moving_average.mdl')))
```



Some blocks in the model already have specified fixed-point data types.

Block	Data Type Specified on Block
`Dbl2Fixpt`	`fixdt(1,16,10)`
`Gain1`	`fixdt(1,32,17)`
`Gain2`	`fixdt(1,32,17)`
`Gain3`	`fixdt(1,32,17)`
`Gain4`	`fixdt(1,16,1)`
`Add1`	`fixdt(1,32,17)`

Block	Data Type Specified on Block
`Add2`	`fixdt(1,32,17)`
`Add3`	`fixdt(1,32,17)`

2 Verify that the target hardware is an embedded processor.

   a In the model, select **Simulation > Configuration Parameters**.

   b In the Configuration Parameters dialog box, select **Hardware Implementation**.

      On the **Hardware Implementation** pane, the **Device vendor** is `Generic` and the **Device type** is `16 bit embedded processor`.

   c Close the Configuration Parameters dialog box.

3 From the model **Tools** menu, select **Fixed-Point Tool**.

4 In the **Shortcuts to set up runs** pane, click the **Model-wide double override and full instrumentation** button to set:

   - **Data type override** to `Double`
   - **Data type override applies to** to `All numeric types`
   - **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`
   - The run name (in the **Data collection** pane **Store results in run** field) to `DoubleOverride`

   Using these settings, the Fixed-Point Tool performs a global override of the fixed-point data types with double-precision data types, thus avoiding quantization effects. During simulation, the tool logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem in the run `DoubleOverride`.

5 Click the Fixed-Point Tool **Simulate** button  to run the simulation.

   The Fixed-Point Tool simulates the model and displays the results in the **Contents** pane in the run named `DoubleOverride`.

Contents of: ex_moving_average* (mmo-dbl)

Column View: Simulation View Show Details

Name	Run	SimMin	SimMax	SimDT	DesignMin	DesignMax	OverflowWraps	Saturations
Add1 : Accumulator	DoubleOverride	0	10.086854311623828	double				
Add1 : Output	DoubleOverride	0	10.086854311623828	double				
Add2 : Accumulator	DoubleOverride	0	5.038028402776449	double				
Add2 : Output	DoubleOverride	0	5.038028402776449	double	-8	15		
Add3 : Accumulator	DoubleOverride	0	2.5167860950974905	double				
Add3 : Output	DoubleOverride	0	2.5167860950974905	double	-4	8		
Data Type Conversion1	DoubleOverride	0	5.048825908847379	double				
Gain1	DoubleOverride	0	2.521242307678959	double				
Gain2	DoubleOverride	0	1.5108372258966896	double				
Gain3	DoubleOverride	0	1.005948869200801	double				
Gain4	DoubleOverride	0	20.173708623247656	double				
Out1	DoubleOverride				-20	40		

6 In the **Automatic data typing for selected system** pane:

a Click *Configure* to display more options.

b Select **Propose word lengths for specified fraction lengths**, then click **Apply**.

7 Click the **Propose word lengths** button, DT.

The Fixed-Point Tool uses available range data to calculate data type proposals according to the following rules:

- Design minimum and maximum values take precedence over the simulation range.

  The **Safety margin for design and derived min/max (%)** parameter specifies a range that differs from that defined by the design range. In this example, no safety margins are set.

- The tool observes the simulation range because the **Simulation min/max** option is selected.

  The **Safety margin for simulation min/max (%)** parameter specifies a range that differs from that defined by the simulation range. In this example, no safety margins are set.

The Fixed-Point Tool analyzes the data types of all fixed-point blocks whose:

- **Lock output data type setting against changes by the fixed-point tools** parameter is not selected.

- **Output data type** parameter specifies a generalized fixed-point number.
- Data types are not inherited types.

For each object in the model, the Fixed-Point Tool proposes the minimum word length that avoids overflow for the collected range information. Because the target hardware is a 16–bit embedded processor, the Fixed-Point tool proposes word lengths based on the number of bits used by the processor for each data type. For more information, see "How the Fixed-Point Tool Uses Target Hardware Information" on page 9-65.

The tool proposes smaller word lengths for `Gain4` and `Gain4:Gain`. The tool calculated that their ideal word length is less than or equal to the character bit length for the embedded processor (`8`), so the tool rounds up the word length to `8`.

Contents of: ex_moving_average* (mmo-dbl)

Column View: Autoscaling View Show Details

Name	Run	SimDT	CompiledDT	Accept	ProposedDT	SpecifiedDT
Add1 : Accumulator	DoubleOverride	double		☐	n/a	Inherit: Inherit via internal rule
Add1 : Output	DoubleOverride	double		☐	fixdt(1,32,17)	fixdt(1,32,17)
Add2 : Accumulator	DoubleOverride	double		☐	n/a	Inherit: Inherit via internal rule
Add2 : Output	DoubleOverride	double		☐	fixdt(1,32,17)	fixdt(1,32,17)
Add3 : Accumulator	DoubleOverride	double		☐	n/a	Inherit: Inherit via internal rule
Add3 : Output	DoubleOverride	double		☐	fixdt(1,32,17)	fixdt(1,32,17)
Data Type Conversion1	DoubleOverride	double		☐	fixdt(1,16,10)	fixdt(1,16,10)
Gain1 : Gain	DoubleOverride			☐	n/a	Inherit: Inherit via internal rule
Gain1	DoubleOverride	double		☐	fixdt(1,32,17)	fixdt(1,32,17)
Gain2 : Gain	DoubleOverride			☐	n/a	Inherit: Inherit via internal rule
Gain2	DoubleOverride	double		☐	fixdt(1,32,17)	fixdt(1,32,17)
Gain3 : Gain	DoubleOverride			☐	n/a	Inherit: Inherit via internal rule
Gain3	DoubleOverride	double		☐	fixdt(1,32,17)	fixdt(1,32,17)
Gain4 : Gain	DoubleOverride			☑	fixdt(1,8,0)	fixdt(1,16,0)
Gain4	DoubleOverride	double		☑	fixdt(1,8,1)	fixdt(1,16,1)
Out1	DoubleOverride			☐	n/a	Inherit: auto
Unit Delay1	DoubleOverride			☐	n/a	
Unit Delay2	DoubleOverride			☐	n/a	

8 To see how the target hardware affects the word length proposal, change the target hardware to FPGA/ASIC.

a In the model, select **Simulation > Configuration Parameters**.

b In the Configuration Parameters dialog box, select **Hardware Implementation**.

c On the **Hardware Implementation** pane, set **Device vendor** to `ASIC/FPGA`. Simulink automatically sets the **Device type** to `ASIC/FPGA`.

d Click **Apply** and close the Configuration Parameters dialog box.

9 In the Fixed-Point Tool **Automatic data typing for selected system** pane, click the **Propose word lengths** button.

Because the target hardware is an FPGA, there are no constraints on the word lengths that the Fixed-Point Tool proposes. The word length for `Gain4:Gain` is `3` and for `Gain4` is `7`.

Contents of: ex_moving_average* (mmo-dbl)

Column View: Autoscaling View Show Details

Name	Run	SimDT	CompiledDT	Accept	ProposedDT	SpecifiedDT
Add1 : Accumulator	DoubleOverride	double		☐	n/a	Inherit: Inherit via internal rule
Add1 : Output	DoubleOverride	double		☑	fixdt(1,22,17)	fixdt(1,32,17)
Add2 : Accumulator	DoubleOverride	double		☐	n/a	Inherit: Inherit via internal rule
Add2 : Output	DoubleOverride	double		☑	fixdt(1,22,17)	fixdt(1,32,17)
Add3 : Accumulator	DoubleOverride	double		☐	n/a	Inherit: Inherit via internal rule
Add3 : Output	DoubleOverride	double		☑	fixdt(1,22,17)	fixdt(1,32,17)
Data Type Conversion1	DoubleOverride	double		☑	fixdt(1,14,10)	fixdt(1,16,10)
Gain1 : Gain	DoubleOverride			☐	n/a	Inherit: Inherit via internal rule
Gain1	DoubleOverride	double		☑	fixdt(1,20,17)	fixdt(1,32,17)
Gain2 : Gain	DoubleOverride			☐	n/a	Inherit: Inherit via internal rule
Gain2	DoubleOverride	double		☑	fixdt(1,19,17)	fixdt(1,32,17)
Gain3 : Gain	DoubleOverride			☐	n/a	Inherit: Inherit via internal rule
Gain3	DoubleOverride	double		☑	fixdt(1,19,17)	fixdt(1,32,17)
Gain4 : Gain	DoubleOverride			☑	fixdt(1,3,0)	fixdt(1,16,0)
Gain4	DoubleOverride	double		☑	fixdt(1,7,1)	fixdt(1,16,1)
Out1	DoubleOverride			☐	n/a	Inherit: auto
Unit Delay1	DoubleOverride			☐	n/a	
Unit Delay2	DoubleOverride			☐	n/a	

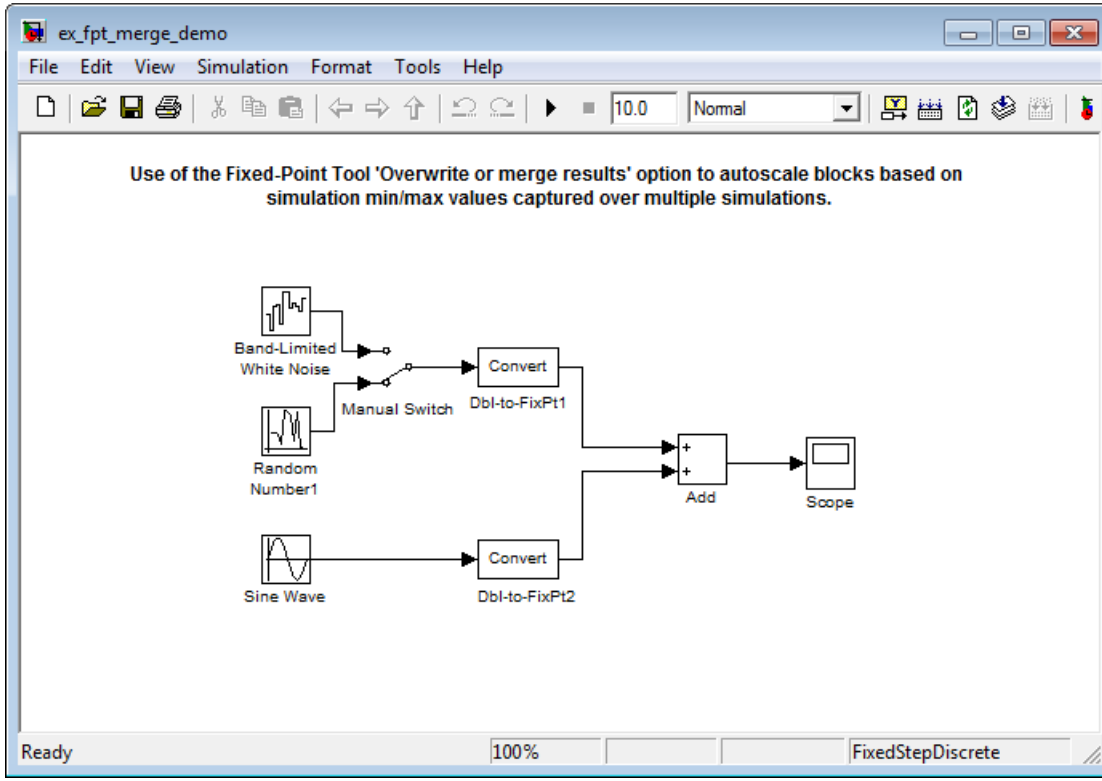
Proposing Data Types For a Model Using Results from Multiple Simulations

In this section...

About this Example

This example demonstrates how to use the Fixed-Point Tool to propose fraction lengths for a model based on the simulation minimum and maximum values captured over multiple simulations.

This example uses the `ex_fpt_merge_demo.mdl` model.



About the Model

The model contains a sine wave input and two alternate noise sources, band-limited white noise or random uniform noise. The software converts the sine wave input and selected noise signal to fixed point and then adds them.

- The Data Type Conversion block Dbl-to-FixPt1 converts the double-precision noise input to the fixed-point data type `fixdt(1,16,15)`.
- The Data Type Conversion block FixPt-to-Dbl2 converts the double-precision sine wave input to the fixed-point data type `fixdt(1,16,10)`.
- The Add block **Accumulator data type** is `fixdt(1,32,30)` and **Output data type** is `fixdt(1,16,14)`.

About the Procedure

In this example, you use the Fixed-Point Tool to merge the results from two simulation runs. Merging results allows you to autoscale your model over the complete simulation range.

The procedure consists of these tasks.

1. "Simulate Using Random Uniform Noise" on page 9-76. Using the Fixed-Point Tool, you simulate the model with the random uniform noise signal and observe the simulation minimum and maximum values for the Add block. The Fixed-Point Tool uses these simulation settings:

   - **Fixed-point instrumentation mode**: `Minimums, maximums and overflows`
   - **Data type override**: `Double`
   - **Data type override applies to**: `All numeric types`
   - **Merge instrumentation results from multiple simulations** is not selected.

   This run provides the simulation results for the random uniform noise input only.

2. "Simulate Using Band-Limited White Noise" on page 9-77. You select the band-limited white noise signal and run another simulation using the same Fixed-Point Tool simulation settings. The Fixed-Point Tool overwrites the results of the previous run.

   This run provides the simulation range for the band-limited white noise input only.

3. "Merge Results" on page 9-77. You configure the Fixed-Point Tool to merge results. Select the random uniform noise input again, rerun the simulation, and observe the simulation results for the Add block.

   This run provides the simulation range based on the entire set of input data for both noise sources.

4. "Propose Fraction Lengths Based on Merged Results" on page 9-77. The Fixed-Point Tool uses the merged simulation minimum and maximum

values to propose scaling for each block to ensure maximum precision while spanning the full range of simulation values.

Running the Simulation

Simulate Using Random Uniform Noise

1. Open the `ex_fpt_merge_demo` model. At the MATLAB command line, enter:

```
run(docpath(fullfile(docroot,'toolbox','fixpoint','examples','ex_fpt_merge_demo.mdl')))
```

2. From the model main menu, select **Tools > Fixed-Point Tool**.

3. In the Fixed-Point Tool **Shortcuts to set up runs** pane, click the **Model-wide double override and full instrumentation** button to set:

   - **Data type override** to `Double`. This option enables each of the model's subsystems to use its locally specified data type settings.
   - **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`.
   - The run name to `DoubleOverride`.

4. In the Fixed-Point Tool, click the **Simulate** button  .

   The Simulink software simulates the `ex_fpt_merge_demo` model, using the random uniform noise signal. Afterward, the Fixed-Point Tool **Contents** pane displays the simulation results for each block that logged fixed-point data. The tool stores the results in a run named `DoubleOverride`, denoted by the **DoubleOverride** label in the **Run** column.

5. Note the **SimMin** and **SimMax** values for the Add block.

   SimMin is `-3.5822`

   SimMax is `2.7598`

Simulate Using Band-Limited White Noise

1. In the model, double-click the switch to select the band-limited white noise signal.

2. In the Fixed-Point Tool, click the **Simulate** button.

   The Simulink software simulates the `ex_fpt_merge_demo` model, now using the band-limited white noise signal.

3. Note the changed values for **SimMin** and **SimMax** for the Add block.

   SimMin is now `-2.5317`

   SimMax is now `3.1542`

Merge Results

1. In the model, double-click the switch to select the random uniform noise signal.

2. In the Fixed-Point Tool **Data collection** pane, select **Merge instrumentation results from multiple simulations**, click **Apply** and rerun the simulation.

3. Note that the **SimMin** and **SimMax** values for the Add block now cover the entire simulation range for both the random uniform and band-limited white noise signals.

   SimMin is `-3.5822`

   SimMax is `3.1542`

Propose Fraction Lengths Based on Merged Results

1. In the **Automatic data typing for selected system** pane, click the **Propose fraction lengths** button.

   The Fixed-Point Tool analyzes the data types of all fixed-point blocks whose:

- **Lock output data type setting against changes by the fixed-point tools** parameter is not selected.
- **Output data type** parameter specifies a generalized fixed-point number.
- Data types are not inherited.

The Fixed-Point Tool uses the merged minimum and maximum values to propose fraction lengths for each block. These values ensure maximum precision while spanning the full range of simulation values. The tool displays the proposed data types in the **Contents** pane.

Contents of: ex_fpt_merge_demo* (mmo-dbl)

Name	Run	SimDT	SimMin	SimMax	SpecifiedDT	ProposedDT	Accept	DesignMin	ProposedMin	ProposedMax
Add : Accumulator	Active	double	-3.5822	3.1542	fixdt(1,32,30)	fixdt(1,32,29)	☑		-4	4
Add : Output	Active	double	-3.5822	3.1542	fixdt(1,16,14)	fixdt(1,16,13)	☑		-4	3.9999
Band-Limited White Noise/Output	Active	double	-1.6059	2.1863	Inherit: Same as input	n/a	☐			
Band-Limited White Noise/Output : ...	Active				Inherit: Same as input	n/a	☐			
Data Type Conversion	Active	double	-2.8944	2.1863	fixdt(1,16,15)	fixdt(1,16,13)	☑		-4	3.9999
Data Type Conversion1	Active	double	-0.99992	0.99957	fixdt(1,16,10)	fixdt(1,16,15)	☑		-1	0.99997
Manual Switch/Constant	Active				Inherit: Inherit from 'Constan...	n/a	☐			
Manual Switch/S-Function	Active					n/a	☐			
Manual Switch/SwitchControl	Active	double	-2.8944	2.1863	Inherit: Inherit via internal rule	n/a	☐			

10

Range Analysis

How Range Analysis Works

In this section...

Prerequisites

Range analysis:

- Requires a Simulink Fixed Point license.
- Does not run on Mac platforms.

Analyzing a Model with Range Analysis

The model that you want to analyze must be compatible with range analysis. If your model is not compatible, either replace unsupported blocks or divide the model so that you can analyze the parts of the model that are compatible. For more information, see "Model Compatibility with Range Analysis" on page 10-6.

The Simulink Fixed Point software performs a static range analysis of your model to derive minimum and maximum range values for signals in the model. The software analyzes the model behavior and computes the values that can occur during simulation for each block Outport. The range of these values is called a *derived range*.

The software statically analyzes the ranges of the individual computations in the model based on:

- Specified design ranges, known as *design minimum and maximum* values, for example, minimum and maximum values specified for:
  - Inport and Outport blocks
  - Block outputs

  - Input, output, and local data used in MATLAB Function and Stateflow Chart blocks
  - Simulink data objects (`Simulink.Signal` and `Simulink.Parameter` objects)

- Inputs
- The semantics of each calculation in the blocks

If the model contains objects that the analysis cannot support, where possible, the software uses automatic stubbing. For more information, see "Automatic Stubbing" on page 10-5.

The range analysis tries to narrow the derived range by using all the specified design ranges in the model. The more design range information you specify, the more likely the range analysis is to succeed. As the software performs the analysis, it derives new range information for the model. The software then attempts to use this new information, together with the specified ranges, to derive ranges for the remaining objects in the model.

For models that contain floating-point operations, range analysis might report a range that is slightly larger than expected. This difference is due to rounding errors because the software approximates floating-point numbers with infinite-precision rational numbers for analysis and then converts to floating point for reporting.

The following table summarizes how the analysis derives range information and provides links to examples.

When...	How the Analysis Works	Examples
You specify design minimum and maximum data for a block output.	The derived range at the block output is based on these specified values and on the following values for blocks connected to its inputs and outputs: • Specified minimum and maximum values • Derived minimum and maximum values .	"Deriving Ranges for an Add Block" on page 10-11
A parameter on a block has initial conditions and a design range.	The analysis takes both factors into account by taking the union of the design range and the initial conditions.	"Using Block Initial Conditions" on page 10-13
The model contains a global tunable parameter with a specified range. (See "Global Tunable Parameters")	The analysis takes into account the range specified for the parameter and ignores the value.	"Using Design Range Information for Simulink.Parameter Objects" on page 10-15
The model contains a global nontunable parameter with a specified range.	The analysis does not take into account the range specified for the parameter. Instead, it uses the parameter value.	"Using Design Range Information for Simulink.Parameter Objects" on page 10-15

When...	How the Analysis Works	Examples
The model contains insufficient design range information.	The analysis cannot determine derived ranges. You must specify more design range information and rerun the analysis.	"Providing More Design Range Information" on page 10-20 The analysis results might depend on block sorting order which determines the order in which the software analyzes the blocks. For more information, see "Controlling and Displaying the Sorted Order" in the Simulink documentation.
The model contains conflicting design range information.	The analysis cannot determine the derived minimum or derived maximum value for an object. The Fixed-Point Tool generates an error. To fix this error, examine the design ranges specified in the model to identify inconsistent design specifications. Modify them to make them consistent.	"Fixing Design Range Conflicts" on page 10-22

Automatic Stubbing

What is Automatic Stubbing?

Automatic stubbing is when the software considers only the interface of the unsupported objects in a model, not their actual behavior. Automatic stubbing

lets you analyze a model that contains objects that the Simulink Fixed Point software does not support. However, if any unsupported model element affects the derivation results, the analysis might achieve only partial results.

How Automatic Stubbing Works

If you enable automatic stubbing, when the range analysis comes to an unsupported block, the software ignores ("stubs") that block. The analysis ignores the behavior of the block. As a result, the block output can take any value.

The software cannot “stub” all Simulink blocks, such as the Integrator block. See the blocks marked “not stubbable” in “Supported and Unsupported Simulink Blocks” on page 10-27.

Model Compatibility with Range Analysis

To verify that your model is compatible with range analysis, see:

- “Unsupported Simulink Software Features” on page 10-25
- “Supported and Unsupported Simulink Blocks” on page 10-27
- “Limitations of Support for Model Blocks” on page 10-36

Deriving Ranges

1. In Simulink, open your model and set it up for use with the Fixed-Point Tool. For more information, see "Prerequisites for Automatic Data Typing Using Derived Data" on page 9-35.

2. From the Simulink **Tools** menu, select **Fixed-Point Tool**.

3. In the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem of interest.

4. If you have a floating-point model, use the Fixed-Point Advisor to prepare the model for conversion.

   a. From the Simulink **Tools** menu, select **Fixed-Point Tool**.

   b. In the Fixed-Point Tool **Model Hierarchy** pane, select the system or subsystem of interest.

   c. Run each task in the Fixed-Point Advisor. For more information, see Chapter 12, "Fixed-Point Advisor Reference".

   The Fixed-Point Advisor:

   - Checks the model against fixed-point guidelines.
   - Identifies unsupported blocks.
   - Removes output data type inheritance from blocks that use floating-point inheritance.
   - Allows you to promote simulation minimum and maximum values to design minimum and maximum values. This capability is useful if you have not specified design ranges and you have simulated the model with inputs that cover the full intended operating range. For more information, see "Specify block minimum and maximum values" on page 12-33.

5. In the **Settings for selected system** pane, set **Data type override** to `Double`, then click **Apply**.

   Using this setting, the Fixed-Point Tool derives ranges for the full range. Otherwise, the tool uses the representable range of the data type specified

on the block to derive a narrower range. The tool then propagates this narrower range through the model.

6 Optionally, in the **Data collection** pane **Store results in run** field, specify a run name. Specifying a unique run name avoids overwriting results from previous runs.

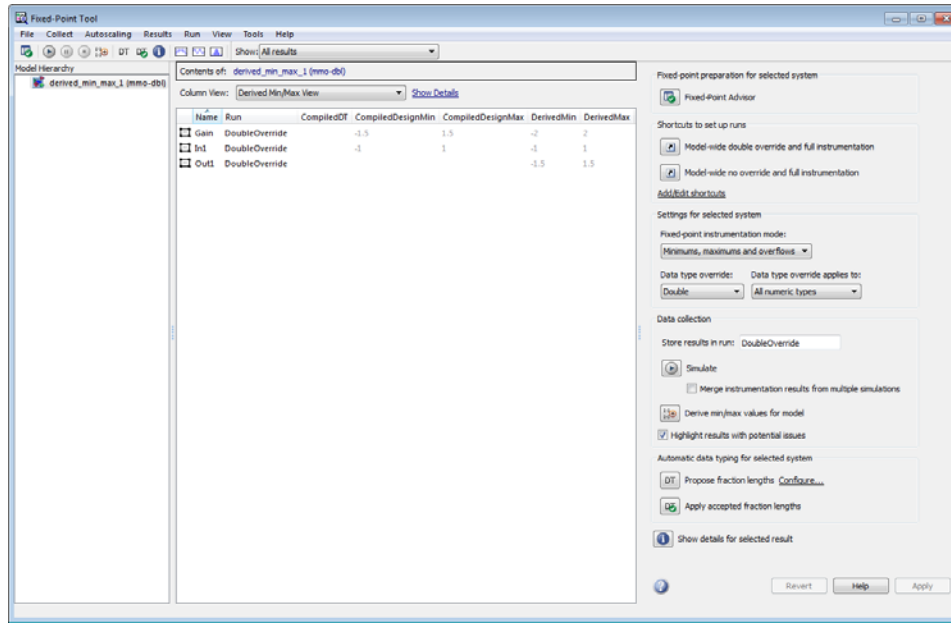
7 In the Fixed-Point Tool, click **Derive min/max values for model**.

The analysis runs and tries to derive range information for objects in the selected system. Your next steps depend on the analysis results.

Analysis Results	Fixed-Point Tool Behavior	Next Steps	For More Information
Successfully derives range data for the model.	Displays the derived minimum and maximum values for the blocks in the selected system.	Review the derived ranges to determine if the results are suitable for proposing data types. If not, you must specify additional design information and rerun the analysis.	"Viewing Derived Range Information in the Fixed-Point Tool" on page 10-10
Fails because the model contains blocks that the software does not support.	Generates an error and provides information about the unsupported blocks.	To fix the error, review the error message information and replace the unsupported blocks.	"Model Compatibility with Range Analysis" on page 10-6
Cannot derive range data because the model contains conflicting design range information.	Generates an error.	To fix this error, examine the design ranges specified in the model to identify inconsistent design specifications. Modify them to make them consistent.	"Fixing Design Range Conflicts" on page 10-22
Cannot derive range data for an object because there is insufficient design range information specified on the model.	Highlights the results for the object.	Examine the model to determine which design range information is missing.	"Providing More Design Range Information" on page 10-20

Viewing Derived Range Information in the Fixed-Point Tool

After you use the Fixed-Point Tool to derive ranges for a model, the Fixed-Point Tool **Contents** pane displays the derived minimum and maximum values for each object in the selected system.



If the analysis cannot derive a minimum or maximum value, the Fixed-Point Tool highlights the result. To fix the issue, examine the model to identify which objects have no specified design ranges and add this information. See "Insufficient Design Range Information" on page 10-18.

Contents of: ex_derived_min_max_4 (mmo-dbl)

Column View: Derived Min/Max View Show Details

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Gain	DoubleOverride	double	-1.5	1.5	-1.5	1.5
In1	DoubleOverride	double	-1		-1	Inf
Out1	DoubleOverride				-1.5	1.5

Range Analysis Examples

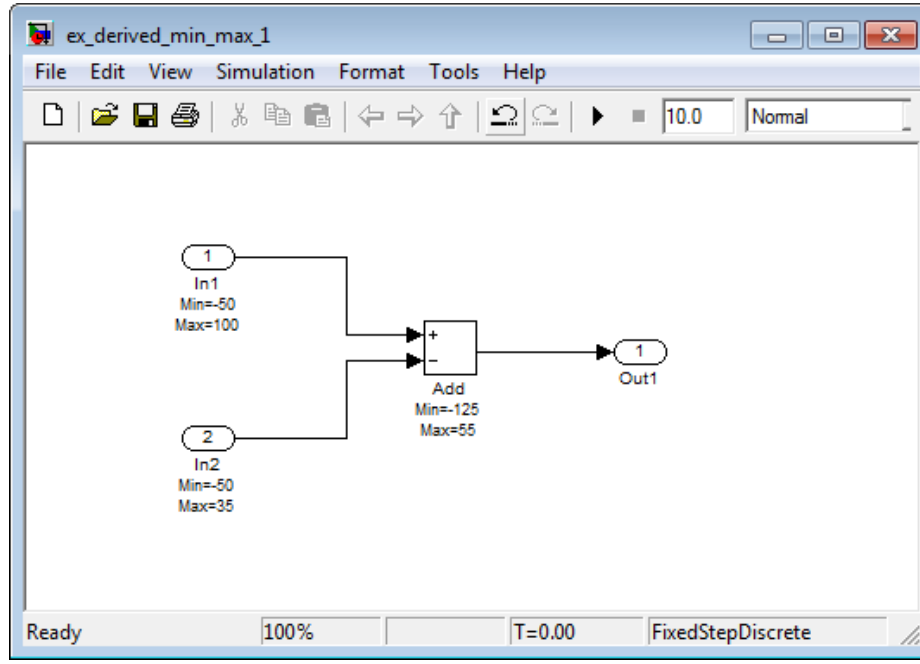
In this section...

Deriving Ranges for an Add Block

This example shows how the range analysis narrows the derived range for the Outport block. This range is based on the range derived for the **Add** block using the design ranges specified on the two Inport blocks and the design range specified for the Add block.

1. Open the `ex_derived_min_max_1` model. At the MATLAB command line, enter:

```
run(docpath(fullfile(docroot,'toolbox','fixpoint','examples','ex_derived_min_max_1.mdl')))
```



The model uses block annotations to display the specified design minimum and maximum values for each block.

- `In1` design range is `[-50,100]`.
- `In2` design range is `[-50,35]`.
- Add block design range is `[125,55]`.

Tip To edit block annotations, right-click the block and, from the context menu, select **Block Properties**. In the **Block Properties** dialog box, select the **Block Annotation** tab. For more information, see "Block Annotation Pane" in the Simulink documentation.

---

2 From the Simulink **Tools** menu, select **Fixed-Point Tool**.

3 In the **Settings for selected system** pane, set **Data type override** to `Double`, then click **Apply**.

The Fixed-Point Tool derives ranges for the full range. If you do not set **Data type override** to `Double`, the tool uses the representable range of the data type specified on the block to derive a narrower range and propagates this narrower range through the model.

4 In the Fixed-Point Tool, click the **Derive min/max values for model** button.

To calculate the derived range at the Add block input, the software uses the design minimum and maximum values specified for the Inport blocks, `[-50,100]` and `[-50,35]`. The derived range at the Add block input is `[-85,150]`.

In the **Contents** pane, the Fixed-Point Tool displays the derived and design minimum and maximum values for the blocks in the selected system.

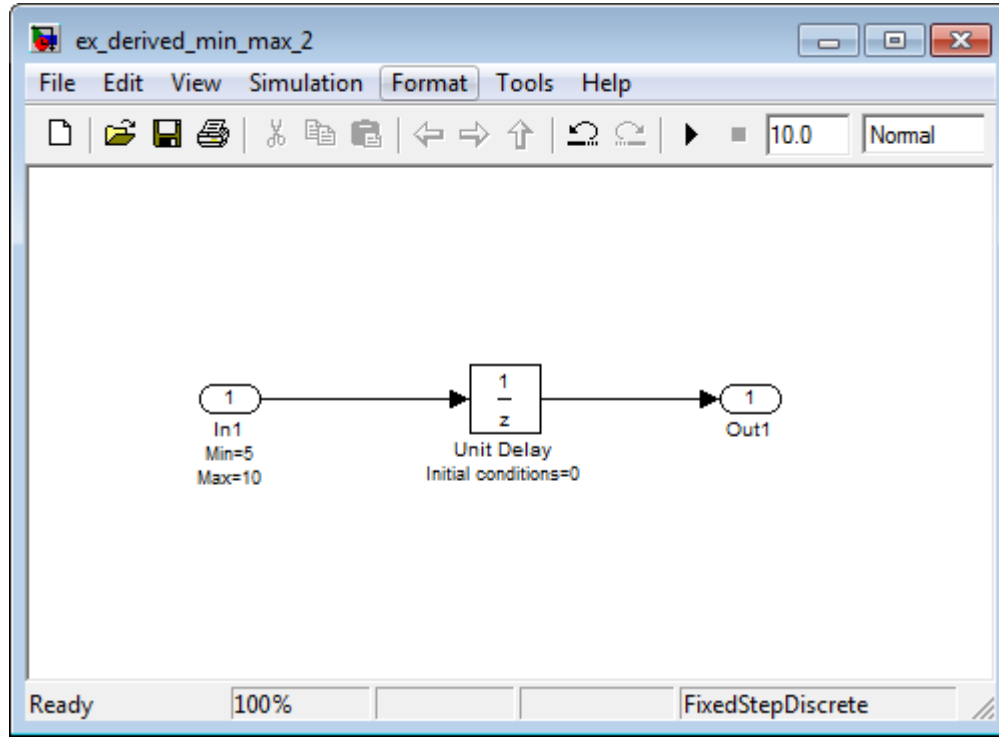
- The derived range for the Add block output signal is narrowed to `[-85,55]`. This derived range is the intersection of the range derived from the block inputs, `[-85,150]` and the design minimum and maximum values specified for the block output, `[-125,55]`.
- The derived range for the Outport block `Out1` is `[-85,55]`, the same as the Add block output.

Using Block Initial Conditions

This example shows how range analysis takes into account block initial conditions.

1 Open the `ex_derived_min_max_2` model. At the MATLAB command line, enter:

```
run(docpath(fullfile(docroot,'toolbox','fixpoint','examples','ex_derived_min_max_2.mdl')))
```



The model uses block annotations to display the specified design minimum and maximum values for the Inport block and the initial conditions for the Unit Delay block.

- `In1` design range is `[5,10]`.
- Unit Delay block initial condition is `0`.

Tip To edit block annotations, right-click the block and, from the context menu, select **Block Properties**. In the **Block Properties** dialog box, select the **Block Annotation** tab. For more information, see "Block Annotation Pane" in the Simulink documentation.

---

2 From the Simulink **Tools** menu, select **Fixed-Point Tool**.

3 In the **Settings for selected system** pane, set **Data type override** to `Double`, then click **Apply**.

   The Fixed-Point Tool derives ranges for the full range. If you do not set **Data type override** to `Double`, the tool uses the representable range of the data type specified on the block to derive a narrower range and propagates this narrower range through the model.

4 In the Fixed-Point Tool, click **Derive min/max values for model**.

   In the **Contents** pane, the Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the model.

   The derived minimum and maximum range for the Outport block, `Out1`, is `[0, 10]` . The range analysis derives this range by taking the union of the initial value, `0`, on the Unit Delay block and the design range on the block, `[5,10]`.

5 Change the initial value of the Unit Delay block to `7`.

   a Double-click the Unit Delay block.

   b In the **Block Parameters** dialog box, set **Initial conditions** to `7`, then click **OK**.

   c In the Fixed-Point Tool, click the **Derive min/max values for model** button.

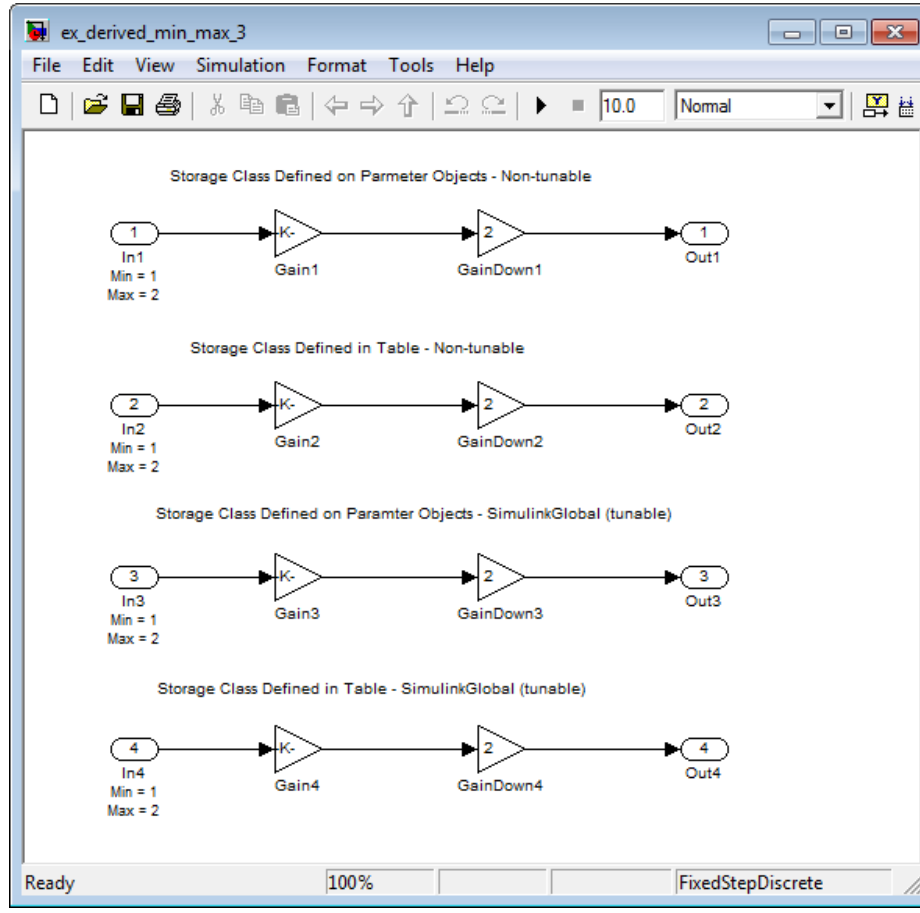
      Because the analysis takes the union of the initial conditions, `7`, and the design range, `[5,10]`, on the Unit Delay block, the derived range for the block is still `[5,10]`.

Using Design Range Information for Simulink.Parameter Objects

This example shows how the range analysis takes into account design range information for `Simulink.Parameter` objects only if they are global tunable parameters. (See "Global Tunable Parameters" in the Simulink documentation.) Otherwise, the analysis uses the value of the parameter.

1 Open the `ex_derived_min_max_3` model. At the MATLAB command line, enter:

```
run(docpath(fullfile(docroot,'toolbox','fixpoint','examples','ex_derived_min_max_3.mdl')))
```



The model uses block annotations to display the specified design minimum and maximum values for the Inport blocks. The design range for all Inport blocks is `[1,2]`.

Tip To edit block annotations, right-click the block and, from the context menu, select **Block Properties**. In the **Block Properties** dialog box, select the **Block Annotation** tab. For more information, see "Block Annotation Pane" in the Simulink documentation.

2 From the Simulink **Tools** menu, select **Fixed-Point Tool**.

3 In the **Settings for selected system** pane, set **Data type override** to `Double` , then click **Apply**.

The Fixed-Point Tool derives ranges for the full range. If you do not set **Data type override** to `Double`, the tool uses the representable range of the data type specified on the block to derive a narrower range and propagates this narrower range through the model.

4 In the Fixed-Point Tool, click the **Derive min/max values for model** button.

In the **Contents** pane, the Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the model.

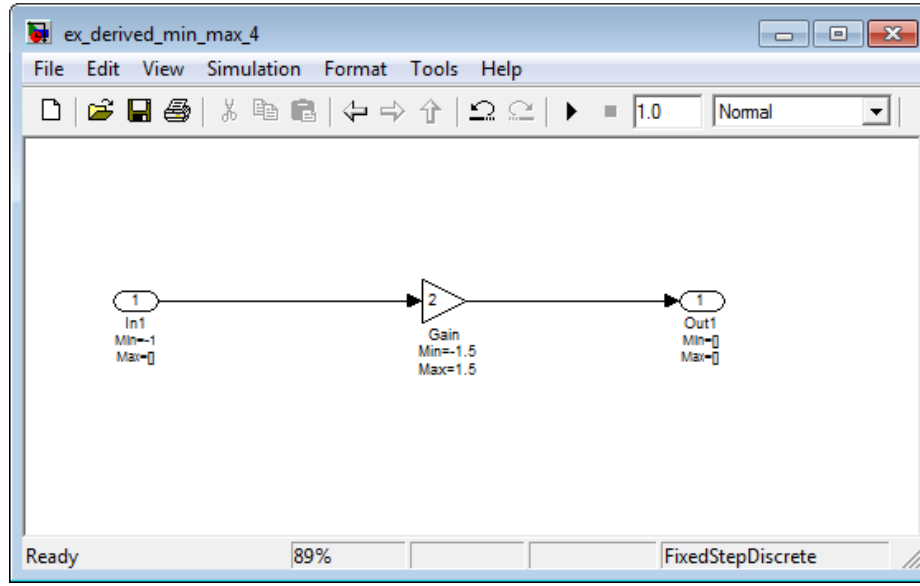
Block	Derived Range	Reason
`Gain1`	`[2,4]`	The gain parameters specified on Gain blocks `Gain1` and `Gain2` are `Simulink.Parameter` objects that have their storage class specified as `Auto`. They are non-tunable parameters. In this case, the range analysis uses the **value** of the `Simulink.Parameter` object, which is `2`, and ignores the design range specified for these parameters.
`Gain2`	`[2,4]`	
`Gain3`	`[1,20]`	The `Simulink.Parameter` objects that specify the gain parameters for these Gain blocks are tunable parameters. The range analysis takes into account the design range, `[1,10]`, specified for these parameters.
`Gain4`	`[1,20]`	

Insufficient Design Range Information

This example shows that if the analysis cannot derive range information because there is insufficient design range information, you can fix the issue by providing additional input design minimum and maximum values.

1 Open the `ex_derived_min_max_4` model. At the MATLAB command line, enter:

```
run(docpath(fullfile(docroot,'toolbox','fixpoint','examples','ex_derived_min_max_4.mdl')))
```



The model uses block annotations to display the specified design minimum and maximum values for the blocks in the model.

- The Inport block `In1` has a design minimum of `-1` but no specified maximum value, as shown by the annotation, `Max=[]`.
- The Gain block has a design range of `[-1.5,1.5]`.
- The Outport block `Out1` has no design range specified, as shown by the annotations, `Min=[]`, `Max=[]`.

Tip To edit block annotations, right-click the block and, from the context menu, select **Block Properties**. In the **Block Properties** dialog box, select the **Block Annotation** tab. For more information, see "Block Annotation Pane" in the Simulink documentation.

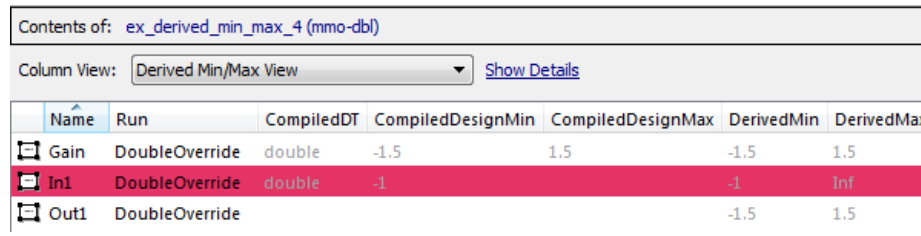
2 From the Simulink **Tools** menu, select **Fixed-Point Tool**.

3 In the **Settings for selected system** pane, set **Data type override** to `Double`, then click **Apply**.

The Fixed-Point Tool derives ranges for the full range. If you do not set **Data type override** to `Double`, the tool uses the representable range of the data type specified on the block to derive a narrower range and propagates this narrower range through the model.

4 In the Fixed-Point Tool, click the **Derive min/max values for model** button.

In the **Contents** pane, the Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the model. The range analysis is unable to derive a maximum value for the Inport block, `In1`. The tool highlights this result.



Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Gain	DoubleOverride	double	-1.5	1.5	-1.5	1.5
In1	DoubleOverride	double	-1		-1	Inf
Out1	DoubleOverride				-1.5	1.5

5 To fix the issue, specify a design maximum value for `In1`:

a In the model, double-click the Inport block, `In1`.

b In the block parameters dialog box, select the **Signal Attributes** tab.

c On this tab, set **Maximum** to `1` and click **OK**.

The model displays the updated maximum value in the block annotation for `In1`.

6 In the Fixed-Point Tool, click **Derive min/max values for model** to rerun the range analysis.

The range analysis can now derive ranges for the Inport and Gain blocks.

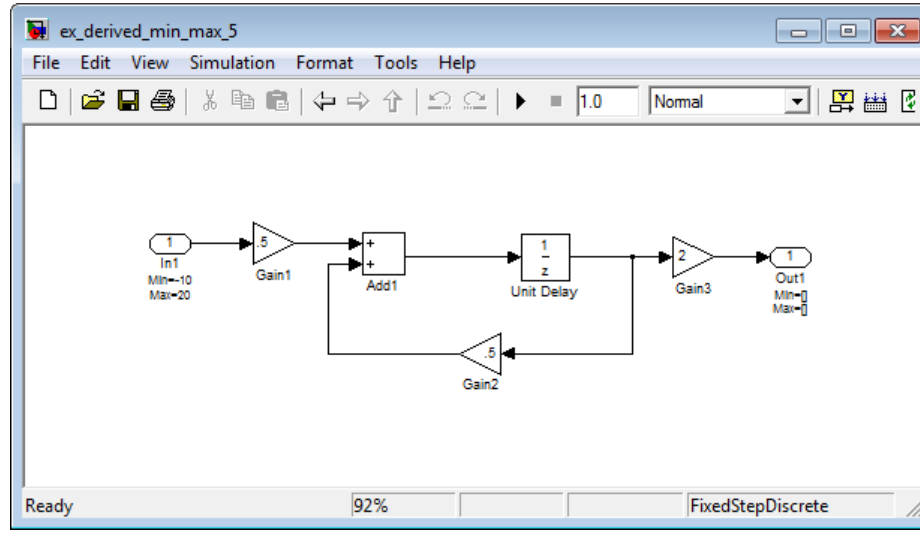
Block	Derived Range	Reason
Inport `In1`	`[-1,1]`	Uses specified design range on the block.
Gain	`[-1.5,1.5]`	The design range specified on the Gain block is`[-1.5,1.5]`. The derived range at the block input is `[-1,1]` (the derived range at the output of `In1`). Therefore, because the gain is `2`, the derived range at the Gain block output is the intersection of the propagated range, `[-2,2]`, and the design range, `[-1.5,1.5]`.
Outport `In2`	`[-1.5,1.5]`	Same as Gain block output because no locally specified design range on Outport block.

Providing More Design Range Information

This example shows that if the analysis cannot derive range information because there is insufficient design range information, you can fix the issue by providing additional output design minimum and maximum values.

1 Open the `ex_derived_min_max_5` model. At the MATLAB command line, enter:

```
run(docpath(fullfile(docroot,'toolbox','fixpoint','examples','ex_derived_min_max_5.mdl')))
```



The model uses block annotations to display the specified design minimum and maximum values for the blocks in the model.

- The Inport block `In1` has a design range of `-10,20`.
- The rest of the blocks in the model have no specified design range.

Tip To edit block annotations, right-click the block and, from the context menu, select **Block Properties**. In the **Block Properties** dialog box, select the **Block Annotation** tab. For more information, see "Block Annotation Pane" in the Simulink documentation.

---

2 From the Simulink **Tools** menu, select **Fixed-Point Tool**.

3 In the **Settings for selected system** pane, set **Data type override** to `Double` , then click **Apply**.

The Fixed-Point Tool derives ranges for the full range. If you do not set **Data type override** to `Double`, the tool uses the representable range of the data type specified on the block to derive a narrower range and propagates this narrower range through the model.

4 In the Fixed-Point Tool, click the **Derive min/max values for model** button.

In the **Contents** pane, the Fixed-Point Tool displays the derived minimum and maximum values for the blocks in the model. Because one of the Add block inputs is fed back from its output, the analysis is unable to derive an output range for the Add block or for any of the blocks connected to this output. The Fixed-Point Tool highlights these results.

Contents of: ex_derived_min_max_5* (dbl)

Column View: Derived Min/Max View Show Details

Name	Run	CompiledDT	CompiledDesignMin	CompiledDesignMax	DerivedMin	DerivedMax
Add1 : Output	Run 1	double			-7.0223e+305	Inf
Gain1	Run 1	double			-5	10
Gain2	Run 1	double			-7.0223e+305	Inf
Gain3	Run 1	double			-2.8089e+306	Inf
In1	Run 1	double	-10	20	-10	20
Out1	Run 1				-2.8089e+306	Inf
Unit Delay	Run 1	double			-1.4045e+306	Inf

5 To fix the issue, specify design minimum and maximum values for `Out1`:

a In the model, double-click the Outport block, `Out1`.

b In the block parameters dialog box, select the **Signal Attributes** tab.

c On this tab, set **Minimum** to `-20` and **Maximum** to `40` and click **OK**.

6 In the Fixed-Point Tool, click **Derive min/max values for model** to rerun the range analysis.

The range analysis uses the minimum and maximum values specified for `Out1`, `[-20,40]` and the gain value of `Gain3`, `2`, to derive an input range for `Gain3`, `[-10,20]`. Because the input of `Gain3` feeds back to the input of the Add block, the analysis now derives ranges for all objects in the model.

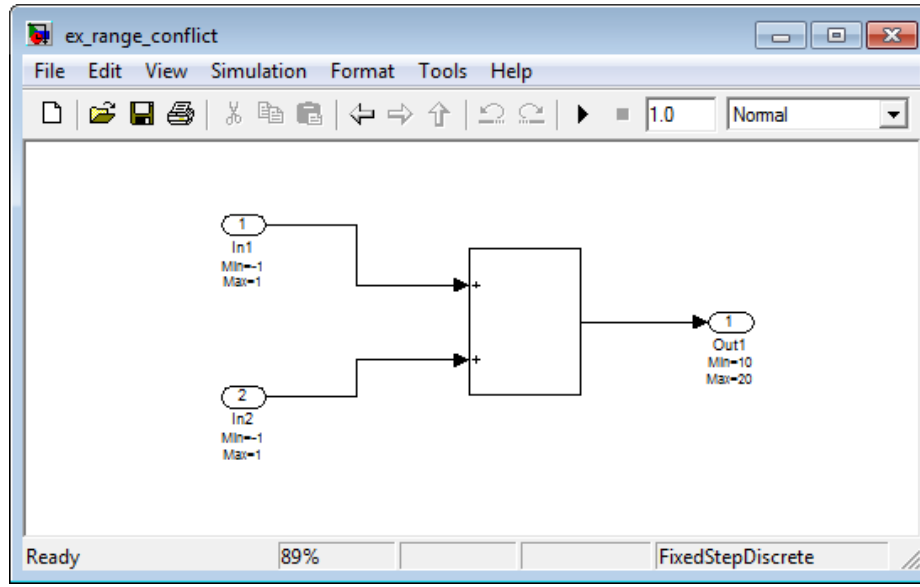
Fixing Design Range Conflicts

This example shows how to fix design range conflicts. If you specify conflicting design minimum and maximum values in your model, the range analysis software reports an error. To fix this error, examine the design ranges specified in the model to identify inconsistent design specifications. Modify them to make them consistent. In this example, the output design range

specified on the Outport block conflicts with the input design ranges specified on the Inport blocks.

1 Open the `ex_range_conflict` model. At the MATLAB command line, enter:

```
run(docpath(fullfile(docroot,'toolbox','fixpoint','examples','ex_range_conflict.mdl')))
```



The model uses block annotations to display the specified design minimum and maximum values for the blocks in the model.

- The Inport blocks `In1` and `In2` have a design range of `[-1,1]`.
- The Outport block `Out1` has a design range of `[10,20]`.

Tip To edit block annotations, right-click the block and, from the context menu, select **Block Properties**. In the **Block Properties** dialog box, select the **Block Annotation** tab. For more information, see "Block Annotation Pane" in the Simulink documentation.

---

2. From the Simulink **Tools** menu, select **Fixed-Point Tool**.

3. In the **Settings for selected system** pane, set **Data type override** to `Double` , then click **Apply**.

   The Fixed-Point Tool derives ranges for the full range. If you do not set **Data type override** to `Double`, the tool uses the representable range of the data type specified on the block to derive a narrower range and propagates this narrower range through the model.

4. In the Fixed-Point Tool, click the **Derive min/max values for model** button.

   The Fixed-Point Tool generates an error because the range analysis fails. It reports an error because the derived range for the Sum block, `[-2,2]` is outside the specified design range for the Outport block, `[10,20]`.

5. Close the error dialog box.

6. To fix the conflict, change the design range on the Outport block to [-10, 20] so that this range includes the derived range for the Sum block.

   a. In the model, double-click the Outport block.

   b. In the block parameters dialog box, click the **Signal Attributes** tab.

   c. On this tab, set **Minimum** to `-10` and click **OK**.

7. In the Fixed-Point Tool, click **Derive min/max values for model** to rerun the range analysis.

   The range analysis derives a minimum value of `-2` and a maximum value of `2` for the Outport block.

Unsupported Simulink Software Features

The software does not support the following Simulink software features. Avoid using these unsupported features.

Not Supported	Description
Variable-step solvers	The software supports only fixed-step solvers. For more information, see "Choosing a Fixed-Step Solver" in the Simulink documentation.
Callback functions	The software does not execute model callback functions during the analysis. The results that the analysis generates may behave inconsistently with the expected behavior. • If a model or any referenced model calls a callback function that changes any block parameters, model parameters, or workspace variables, the analysis does not reflect those changes. • Changing the storage class of base workspace variables on model callback functions or mask initializations is not supported. • Callback functions called prior to analysis, such as the `PreLoadFcn` or `PostLoadFcn` model callbacks, are fully supported.
Model callback functions	The software only supports model callback functions if the `InitFcn` callback of the model is empty.
Algebraic loops	The software does not support models that contain algebraic loops. For more information, see "Algebraic Loops" in the Simulink documentation.
Masked subsystem initialization functions	The software does not support models whose masked subsystem initialization modifies any attribute of any workspace parameter.

Not Supported	Description
Complex signals	The software supports only real signals. For more information, see "Complex Signals" in the Simulink documentation.
Variable-size signals	The software does not support variable-size signals. A variable-size signal is a signal whose size (number of elements in a dimension), in addition to its values, can change during model execution. For more information, see "Working with Variable-Size Signals" in the Simulink documentation.
Arrays of buses	The software does not support arrays of buses. For more information, see "Combining Buses into an Array of Buses" in the Simulink documentation.
Multiword fixed-point data types	The software does not support multiword fixed-point data types.
Nonfinite data	The software does not support nonfinite data (for example, `NaN` and `Inf`) and related operations.
Signals with nonzero sample time offset	The software does not support models with signals that have nonzero sample time offsets.
Models with no output ports	The software only supports models that have one or more output ports.

Supported and Unsupported Simulink Blocks

Overview of Simulink Block Support

The following tables summarize the analysis support for Simulink blocks. Each table lists all the blocks in each Simulink library and describes support information for that particular block. A dash (—) indicates that the software supports that block under all conditions. If the software does not support a given block, where possible, automatic stubbing considers the interface of the unsupported blocks, but not their behavior, during the analysis. However, if any of the unsupported blocks affect the simulation outcome, the analysis may achieve only partial results. If the analysis cannot use automatic stubbing for a block, the block is marked as "not stubbable". For more information, see "Automatic Stubbing" on page 10-5.

Additional Math and Discrete Library

The software supports all blocks in the Additional Math and Discrete library.

Commonly Used Blocks Library

The Commonly Used Blocks library includes blocks from other libraries. Those blocks are listed under their respective libraries.

Continuous Library

Block	Support Notes
Derivative	Not supported
Integrator	Not supported and not stubbable
Integrator Limited	Not supported and not stubbable
PID Controller	Not supported
PID Controller (2 DOF)	Not supported
Second Order Integrator	Not supported and not stubbable
Second Order Integrator Limited	Not supported and not stubbable
State-Space	Not supported

Block	Support Notes
Transfer Fcn	Not supported
Transport Delay	Not supported
Variable Time Delay	Not supported
Variable Transport Delay	Not supported
Zero-Pole	Not supported

Discontinuities Library

The software supports all blocks in the Discontinuities library.

Discrete Library

Block	Support Notes
Delay	—
Difference	—
Discrete Derivative	—
Discrete Filter	The software analyzes through the filter. It does not derive any range information for the filter.
Discrete FIR Filter	The software analyzes through the filter. It does not derive any range information for the filter.
Discrete PID Controller	—
Discrete PID Controller (2 DOF)	—
Discrete State-Space	Not supported
Discrete Transfer Fcn	—
Discrete Zero-Pole	Not supported
Discrete-Time Integrator	—
First-Order Hold	—
Memory	—

Block	Support Notes
Tapped Delay	—
Transfer Fcn First Order	—
Transfer Fcn Lead or Lag	—
Transfer Fcn Real Zero	—
Unit Delay	—
Zero-Order Hold	—

Logic and Bit Operations Library

The software supports all blocks in the Logic and Bit Operations library.

Lookup Tables Library

Block	Support Notes
Cosine	—
Direct Lookup Table (n-D)	—
Interpolation Using Prelookup	Not supported when: • The **Interpolation method** parameter is `Linear` and the **Number of table dimensions** parameter is greater than `4`. or • The **Interpolation method** parameter is `Linear` and the **Number of sub-table selection dimensions** parameter is not `0`.
1-D Lookup Table	Not supported when the **Interpolation method** or the **Extrapolation method** parameter is `Cubic Spline`.
2-D Lookup Table	Not supported when the **Interpolation method** or the **Extrapolation method** parameter is `Cubic Spline`.

Block	Support Notes
n-D Lookup Table	Not supported when: • The **Interpolation method** or the **Extrapolation method** parameter is `Cubic Spline`. or • The **Interpolation method** parameter is `Linear` and the **Number of table dimensions** parameter is greater than `5`.
Lookup Table Dynamic	—
Prelookup	Not supported when the block outputs an interval fraction of fixed-point data type.
Sine	—

Math Operations Library

Block	Support Notes
Abs	—
Add	—
Algebraic Constraint	—
Assignment	—
Bias	—
Complex to Magnitude-Angle	Not supported
Complex to Real-Imag	Not supported
Divide	—
Dot Product	—
Find Nonzero Elements	—
Gain	—
Magnitude-Angle to Complex	Not supported

<table>
<tr><th>Block</th><th>Support Notes</th></tr>
<tr><td>Math Function</td><td>All signal types support the following Function parameter settings.
<table>
<tr><td>conj</td><td>hermitian</td><td>magnitude^2</td><td>mod</td></tr>
<tr><td>rem</td><td>reciprocal</td><td>square</td><td>transpose</td></tr>
</table>
The software does not support the following Function parameter settings.
<table>
<tr><td>10^u</td><td>exp</td><td>hypot</td></tr>
<tr><td>log</td><td>log10</td><td>pow</td></tr>
</table></td></tr>
<tr><td>Matrix Concatenate</td><td>—</td></tr>
<tr><td>MinMax</td><td>—</td></tr>
<tr><td>MinMax Running Resettable</td><td>—</td></tr>
<tr><td>Permute Dimensions</td><td>—</td></tr>
<tr><td>Polynomial</td><td>—</td></tr>
<tr><td>Product</td><td>—</td></tr>
<tr><td>Product of Elements</td><td>—</td></tr>
<tr><td>Real-Imag to Complex</td><td>Not supported</td></tr>
<tr><td>Reciprocal Sqrt</td><td>Not supported</td></tr>
<tr><td>Reshape</td><td>—</td></tr>
<tr><td>Rounding Function</td><td>—</td></tr>
<tr><td>Sign</td><td>—</td></tr>
<tr><td>Signed Sqrt</td><td>Not supported</td></tr>
<tr><td>Sine Wave Function</td><td>Not supported</td></tr>
<tr><td>Slider Gain</td><td>—</td></tr>
<tr><td>Sqrt</td><td>Not supported</td></tr>
</table>

Block	Support Notes
Squeeze	—
Subtract	—
Sum	—
Sum of Elements	—
Trigonometric Function	Supported when **Function** is `sin`, `cos`, or `sincos` and **Approximation method** is `CORDIC`.
Unary Minus	—
Vector Concatenate	—
Weighted Sample Time Math	—

Model Verification Library

The software supports all blocks in the Model Verification library.

Model-Wide Utilities Library

Block	Support Notes
Block Support Table	—
DocBlock	—
Model Info	—
Timed-Based Linearization	Not supported
Trigger-Based Linearization	Not supported

Ports & Subsystems Library

Block	Support Notes
Atomic Subsystem	—
Code Reuse Subsystem	—

Block	Support Notes
Configurable Subsystem	—
Enable	—
Enabled Subsystem	—
Enabled and Triggered Subsystem	Not supported when the trigger control signal specifies a fixed-point data type.
For Each	Not supported
For Each Subsystem	Not supported
For Iterator Subsystem	—
Function-Call Feedback Latch	—
Function-Call Generator	—
Function-Call Split	—
Function-Call Subsystem	—
If Action Subsystem	—
Inport	—
Model	Supported except for the limitations described in "Limitations of Support for Model Blocks" on page 10-36.
Model Variants	Supported except for the limitations described in "Limitations of Support for Model Blocks" on page 10-36.
Outport	—
Subsystem	—
Switch Case	—
Switch Case Action Subsystem	—
Trigger	—
Triggered Subsystem	Not supported when the trigger control signal specifies a fixed-point data type.
Variant Subsystem	—
While Iterator Subsystem	—

Signal Attributes Library

The software supports all blocks in the Signal Attributes library.

Signal Routing Library

Block	Support Notes
Bus Assignment	—
Bus Creator	—
Bus Selector	—
Data Store Memory	—
Data Store Read	—
Data Store Write	—
Demux	—
Environment Controller	—
From	—
Goto	—
Goto Tag Visibility	—
Index Vector	—
Manual Switch	The Manual Switch block is compatible with the software, but the analysis ignores this block in a model.
Merge	—
Multiport Switch	—
Mux	—
Selector	—
Switch	—
Vector Concatenate	—

Sinks Library

Block	Support Notes
Display	—
Floating Scope	—
Outport (Out1)	—
Scope	—
Stop Simulation	Not supported and not stubbable
Terminator	—
To File	—
To Workspace	—
XY Graph	—

Sources Library

Block	Support Notes
Band-Limited White Noise	Not supported
Chirp Signal	Not supported
Clock	—
Constant	Supported unless **Constant value** is `inf`.
Counter Free-Running	—
Counter Limited	—
Digital Clock	—
Enumerated Constant	—
From File	Not supported
From Workspace	Not supported
Ground	—
Inport (In1)	—

Block	Support Notes
Pulse Generator	—
Ramp	—
Random Number	Not supported and not stubbable
Repeating Sequence	Not supported
Repeating Sequence Interpolated	Not supported
Repeating Sequence Stair	—
Signal Builder	Not supported
Signal Generator	Not supported
Sine Wave	Not supported
Step	—
Uniform Random Number	Not supported and not stubbable

User-Defined Functions Library

Block	Support Notes
Fcn	Supports all operators except `^`. Supports only the mathematical functions `abs`, `ceil`, `fabs`, `floor`, `rem`, and `sgn`.
Interpreted MATLAB Function	Not supported
MATLAB Function	The software analyzes through the MATLAB Function block.
Level-2 MATLAB S-Function	Not supported
S-Function	Not supported
S-Function Builder	Not supported

Limitations of Support for Model Blocks

The software supports the Model block, but with the following limitations. The software cannot analyze a model that contains one or more Model blocks if:

- The referenced model is protected. Protected referenced models are encoded to obscure their contents. This feature allows third parties to use the referenced model without being able to view the intellectual property that makes up the model.

Note For more information, see "Protecting Referenced Models" in the Simulink documentation.

- The parent model or any of the referenced models gives an error when you set one of the following model parameters in the Configuration Parameters dialog box to `error`:
  - **Diagnostics > Connectivity > Element name mismatch**
  - **Diagnostics > Connectivity > Mux blocks used to create bus signals**

  You can use the **Element name mismatch** diagnostic along with bus objects so that your model meets the bus element naming requirements imposed by some blocks.

  If your model contains Mux blocks that create bus signals, refer to "Tips" in "Mux blocks used to create bus signals" to resolve this problem.
- The Model block uses asynchronous function-call inputs.
- Any of the Model blocks in the model reference hierarchy creates an artificial algebraic loop. If this occurs, take the following steps:

  1. On the **Diagnostics** pane of the Configuration Parameters dialog box, set the **Minimize algebraic loop** parameter to `error` so that Simulink reports an algebraic loop error.
  2. On the **Model Referencing** Pane of the Configuration Parameters dialog box, select the Minimize algebraic loop occurrences parameter.

     Simulink tries to eliminate the artificial algebraic loop during simulation.
  3. Simulate the model.
  4. If Simulink cannot eliminate the artificial algebraic loop, highlight the location of the algebraic loop by selecting **Edit > Update Diagram**.

5 Eliminate the artificial algebraic loop so that the software can analyze the model. Break the loop with Unit Delay blocks so that the execution order is predictable.

Note For more information, see "Algebraic Loops" in the Simulink documentation.

11

Code Generation

Overview

You can generate C code with the Simulink Fixed Point software by using the Simulink Coder product. The code generated from fixed-point models uses only integer types and automatically includes all operations, such as shifts, needed to account for differences in fixed-point locations. You can use the generated code on embedded fixed-point processors or on rapid prototyping systems even if they contain a floating-point processor. For more information about code generation, refer to the Simulink Coder documentation.

You can generate code for testing on a rapid prototyping system using products such as xPC Target™, Real-Time Windows Target™, or dSPACE® software. The target compiler and processor may support floating-point operations in software or in hardware. In any case, the fixed-point portions of a model generate pure integer code and do not use floating-point operations. This allows valid bit-true testing even on a floating-point processor.

You can also generate code for non-real-time testing. For example, you can generate code to run in nonreal time on computers running any supported operating system. Even though the processors have floating-point hardware, the code generated by fixed-point blocks is pure integer code. The Generic Real-Time Target (GRT) in the Simulink Coder product and acceleration modes in the Simulink software are examples of where non-real-time code is generated and run.

Code Generation Support

In this section...

Introduction

All fixed-point blocks support code generation, except particular simulation features. The sections that follow describe the code generation support that the Simulink Fixed Point software provides.

Languages

C code generation is supported.

Data Types

Fixed-point code generation supports all integer and fixed-point data types that are supported by simulation. See "Data Type Support" on page 1-20.

Rounding Modes

All rounding modes—`Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, and `Zero` —are supported.

Overflow Handling

- Saturation and wrapping are supported.
- Wrapping generates the most efficient code.

- Currently, you cannot choose to exclude saturation code automatically when hardware saturation is available. Select wrapping in order for the Simulink Coder product to exclude saturation code.

Blocks

All blocks generate code for all operations with a few exceptions. The Lookup Table Dynamic block generates code for all lookup methods except `Interpolation-Extrapolation`.

Scaling

Any binary-point-only scaling and [Slope Bias] scaling that is supported in simulation is supported, bit-true, in code generation.

Accelerating Fixed-Point Models

If the model meets the code generation restrictions, you can use Simulink acceleration modes with your fixed-point model. The acceleration modes can drastically increase the speed of some fixed-point models. This is especially true for models that execute a very large number of time steps. The time overhead to generate code for a fixed-point model is generally larger than the time overhead to set up a model for simulation. As the number of time steps increases, the relative importance of this overhead decreases.

Note Rapid Accelerator mode does not support models with bus objects or 33+ bit fixed-point data types as parameters.

Every Simulink model is configured to have a start time and a stop time in the Configuration Parameters dialog box. Simulink simulations are usually configured for non-real-time execution, which means that the Simulink software tries to simulate the behavior from the specified start time to the stop time as quickly as possible. The time it takes to complete a simulation consists of two parts: overhead time and core simulation time, which is spent calculating changes from one time step to the next. For any model, the time it takes to simulate if the stop time is the same as the start time can be regarded as the overhead time. If the stop time is increased, the simulation takes longer. This additional time represents the core simulation time. Using an acceleration mode to simulate a model has an initially larger overhead time that is spent generating and compiling code. For any model, if the simulation stop time is sufficiently close to the start time, then Normal mode simulation is faster than an acceleration mode. But an acceleration mode can eliminate the overhead of code generation for subsequent simulations if structural changes to the model have not occurred.

In Normal mode, the Simulink software runs general code that can handle various situations. In an acceleration mode, code is generated that is tailored to the current usage. For fixed-point use, the tailored code is much leaner than the simulation code and executes much faster. The tailored code allows an acceleration mode to be much faster in the core simulation time. For any model, when the stop time is close to the start time, overhead dominates the overall simulation time. As the stop time is increased, there is a point at which the core simulation time dominates overall simulation time. Normal

mode has less overhead compared to an acceleration mode when fresh code generation is necessary. Acceleration modes are faster in the core simulation portion. For any model, there is a stop time for which Normal mode and acceleration mode with fresh code generation have the same overall simulation time. If the stop time is decreased, then Normal mode is faster. If the stop time is increased, then an acceleration mode has an increasing speed advantage. Eventually, the acceleration mode speed advantage is drastic.

Normal mode generally uses more tailored code for floating-point calculations compared to fixed-point calculations. Normal mode is therefore generally much faster for floating-point models than for similar fixed-point models. For acceleration modes, the situation often reverses and fixed point becomes significantly faster than floating point. As noted above, the fixed-point code goes from being general to highly tailored and efficient. Depending on the hardware, the integer-based fixed-point code can gain speed advantages over similar floating-point code. Many processors can do integer calculations much faster than similar floating-point operations. In addition, if the data bus is narrow, there can also be speed advantages to moving around 1-, 2-, or 4-byte integer signals compared to 4- or 8-byte floating-point signals.

See “Accelerating Models” in *Simulink User’s Guide* for more information.

Using External Mode or Rapid Simulation Target

In this section...

Introduction

If you are using the Simulink Coder external mode or rapid simulation (rsim) target (see *Simulink Coder User's Guide* for more information), there are situations where you might get unexpected errors when tuning block parameters. These errors can arise when you specify the `Best precision` scaling option for blocks that support constant scaling for best precision. See "Constant Scaling for Best Precision" on page 2-12 for a description of the constant scaling feature.

The sections that follow provide further details about the errors you might encounter. To avoid these errors, specify a scaling value instead of using the `Best precision` scaling option.

External Mode

If you change a parameter such that the binary point moves during an external mode simulation or during graphical editing, and you reconnect to the target, a checksum error occurs and you must rebuild the code. When you use `Best Precision` scaling, the binary point is automatically placed based on the value of a parameter. Each power of two roughly marks the boundary where a parameter value maps to a different binary point. For example, a parameter value of 1 to 2 maps to a particular binary point position. If you change the parameter to a value of 2 to 4, the binary point moves one place to the right, while if you change the parameter to a value of 0.5 to 1, it moves one place to the left.

For example, suppose a block has a parameter value of `-2`. You then build the code and connect in external mode. While connected, you change the parameter to `-4`. If the simulation is stopped and then restarted, this parameter change causes a binary point change. In external mode, the binary

point is kept fixed. If you keep the parameter value of `-4` and disconnect from the target, then when you reconnect, a checksum error occurs and you must rebuild the code.

Rapid Simulation Target

If a parameter change is great enough, and you are using the best precision mode for constant scaling, then you cannot use the rsim target.

If you change a block parameter by a sufficient amount (approximately a factor of two), the best precision mode changes the location of the binary point. Any change in the binary point location requires the code to be rebuilt because the model checksum is changed. This means that if best precision parameters are changed over a great enough range, you cannot use the rapid simulation target and a checksum error message occurs when you initialize the rsim executable.

Optimizing Your Generated Code

In this section...

Introduction

The sections listed in the following table discuss tips to help you to optimize your code generated from fixed-point blocks, in order to reduce ROM usage or model execution time:

Tips for Reducing ROM Consumption or Model Execution Time

Tip	Reduces ROM	Reduces Model Execution Time
"Restrict Data Type Word Lengths" on page 11-10	Yes	Yes
"Avoid Fixed-Point Scalings with Bias" on page 11-11	Yes	Yes
"Wrap and Round to Floor or Simplest" on page 11-11	Yes	Yes

Tips for Reducing ROM Consumption or Model Execution Time (Continued)

Tip	Reduces ROM	Reduces Model Execution Time
"Limit the Use of Custom Storage Classes" on page 11-13	Yes	No
"Limit the Use of Unevenly Spaced Lookup Tables" on page 11-13	Yes	Yes
"Minimize the Variety of Similar Fixed-Point Utility Functions" on page 11-13	Yes	No
"Handle Net Slope Correction" on page 11-14	Dependent on model configuration, compiler, and target hardware	Dependent on model configuration, compiler, and target hardware
"Optimize Generated Code Using Specified Minimum and Maximum Values" on page 11-28	Yes	Yes

Restrict Data Type Word Lengths

If possible, restrict the fixed-point data type word lengths in your model so that they are equal to or less than the integer size of your target microcontroller. This results in fewer mathematical instructions in the microcontroller, and reduces ROM and execution time.

This recommendation strongly applies to global variables that consume global RAM. For example, Unit Delay blocks have discrete states that have the same word lengths as their input and output signals. These discrete states are global variables that consume global RAM, which is a scarce resource on many embedded systems.

For temporary variables that only occupy a CPU register or stack location briefly, the space consumed by a `long` is less critical. However, depending on the operation, the use of `long` variables in math operations can be expensive. Addition and subtraction of long integers generally requires the same effort as adding and subtracting regular integers, so that operation is not a concern.

In contrast, multiplication and division with long integers can require significantly larger and slower code.

Avoid Fixed-Point Scalings with Bias

Whenever possible, avoid using fixed-point numbers with bias. In certain cases, if you choose biases carefully, you can avoid significant increases in ROM and execution time. Refer to "Recommendations for Arithmetic and Scaling" on page 3-34 for more information on how to choose appropriate biases in cases where it is necessary; for example if you are interfacing with a hardware device that has a built-in bias. In general, however, it is safer to avoid using fixed-point numbers with bias altogether.

Inputs to lookup tables are an important exception to this recommendation. If a lookup table input and the associated input data use the same bias, then there is no penalty associated with nonzero bias for that operation.

Wrap and Round to Floor or Simplest

For most fixed-point and integer operations, the Simulink software provides you with options on how overflows are handled and how calculations are rounded. Traditional handwritten code, especially for control applications, almost always uses the "no effort" rounding mode. For example, to reduce the precision of a variable, that variable is shifted right. For unsigned integers and two's complement signed integers, shifting right is equivalent to rounding to floor. To get results comparable to or better than what you expect from traditional handwritten code, you should round to floor in most cases.

The primary exception to this rule is the rounding behavior of signed integer division. The C language leaves this rounding behavior unspecified, but for most targets the "no effort" mode is round to zero. For unsigned division, everything is nonnegative, so rounding to floor and rounding to zero are identical.

You can improve code efficiency by setting the value of the **Configuration Parameters > Hardware Implementation > Embedded Hardware > Signed integer division rounds to** parameter to describe how your production target handles rounding for signed division. For Product blocks that are doing only division, setting the **Integer rounding mode** parameter to the rounding mode of your production target gives the best results. You

can also use the `Simplest` rounding mode on blocks where it is available. For more information, refer to "Rounding Mode: Simplest" on page 3-13.

The options for overflow handling also have a big impact on the efficiency of your generated code. Using software to detect overflow situations and saturate the results requires the code to be much bigger and slower compared to simply ignoring the overflows. When overflows are ignored for unsigned integers and two's complement signed integers, the results usually wrap around modulo 2^N, where N is the number of bits. Unhandled overflows that wrap around are highly undesirable for many situations.

However, because of code size and speed needs, traditional handwritten code contains very little software saturation. Typically, the fixed-point scaling is very carefully set so that overflow does not occur in most calculations. The code for these calculations safely ignores overflow. To get results comparable to or better than what you would expect from traditional handwritten code, the **Saturate on integer overflow** parameter should not be selected for Simulink blocks doing those calculations.

In a design, there might be a few places where overflow can occur and saturation protection is needed. Traditional handwritten code includes software saturation for these few places where it is needed. To get comparable generated code, the **Saturate on integer overflow** parameter should only be selected for the few Simulink blocks that correspond to these at-risk calculations.

A secondary benefit of using the most efficient options for overflow handling and rounding is that calculations often reduce from multiple statements requiring several lines of C code to small expressions that can be folded into downstream calculations. Expression folding is a code optimization technique that produces benefits such as minimizing the need to store intermediate computations in temporary buffers or variables. This can reduce stack size and make it more likely that calculations can be efficiently handled using only CPU registers. An automatic code generator can carefully apply expression folding across parts of a model and often see optimizations that might not be obvious. Automatic optimizations of this type often allow generated code to exceed the efficiency of typical examples of handwritten code.

Limit the Use of Custom Storage Classes

In addition to the tip mentioned in "Wrap and Round to Floor or Simplest" on page 11-11, to obtain the maximum benefits of expression folding you also need to make sure that the **Storage class** field in the Signal Properties dialog box is set to `Auto` for each signal. When you choose a setting other than `Auto`, you need to name the signal, and a separate statement is created in the generated code. Therefore, only use a setting other than `Auto` when it is necessary for global variables.

You can access the Signal Properties dialog box by selecting any connection between blocks in your model, and then selecting **Signal Properties** from the Simulink **Edit** menu.

Limit the Use of Unevenly Spaced Lookup Tables

If possible, use lookup tables with nontunable, evenly spaced axes. A table with an unevenly spaced axis requires a search routine and memory for each input axis, which increases ROM and execution time. However, keep in mind that an unevenly spaced lookup table might provide greater accuracy. You need to consider the needs of your algorithm to determine whether you can forgo some accuracy with an evenly spaced table in order to reduce ROM and execution time. Also note that this decision applies only to lookup tables with nontunable input axes, because tables with tunable input axes always have the potential to be unevenly spaced.

Minimize the Variety of Similar Fixed-Point Utility Functions

The Embedded Coder product generates fixed-point utility functions that are designed to handle specific situations efficiently. The Simulink Coder product can generate multiple versions of these optimized utility functions depending on what a specific model requires. For example, the division of `long` integers can, in theory, require eight varieties that are combinations of the output and the two inputs being signed or unsigned. A model that uses all these combinations can generate utility functions for all these combinations.

In some cases, it is possible to make small adjustments to a model that reduce the variety of required utility functions. For example, suppose that across most of a model signed data types are used, but in a small part of a model, a

local decision to use unsigned data types is made. If it is possible to switch that portion of the model to use signed data types, then the overall variety of generated utility functions can potentially be reduced.

The best way to identify these opportunities is to inspect the generated code. For each utility function that appears in the generated code, you can search for all the call sites. If relatively few calls to the function are made, then trace back from the call site to the Simulink model. By modifying those places in the Simulink model, it is possible for you to eliminate the few cases that need a rarely used utility function.

Handle Net Slope Correction

The Simulink Fixed Point software provides an optimization parameter, **Use integer division to handle net slopes that are reciprocals of integers**, that controls how the software handles net slope correction. To learn how to enable this optimization, see "How to Use Integer Division to Handle Net Slope Correction" on page 11-15.

When a change of fixed-point slope is not a power of two, net slope correction is necessary. Normally, net slope correction is implemented using an integer multiplication followed by shifts. Under some conditions, an alternate implementation requires just an integer division by a constant. One of the conditions is that the net slope can be accurately represented as the reciprocal of an integer. Under this condition, the division implementation gives more accurate numerical behavior. Depending on your compiler and embedded hardware, the division implementation might be more desirable than the multiplication and shifts implementation. The generated code for the division implementation might require less ROM or improve model execution time.

When to Use Integer Division to Handle Net Slope Correction

This optimization works if:

- The net slope is a reciprocal of an integer.
- Division is more efficient than multiplication followed by shifts on the target hardware.

Note The Simulink Fixed Point software is not aware of the target hardware. Before selecting this option, verify that division is more efficient than multiplication followed by shifts on your target hardware.

When Not to Use Integer Division to Handle Net Slope Correction

This optimization does not work if:

- The software cannot perform the division using the production target `long` data type and therefore must use multiword operations.

  Using multiword division does not produce code suitable for embedded targets. Therefore, do not use integer division to handle net slope correction in models that use multiword operations. If your model contains blocks that use multiword operations, change the word length of these blocks to avoid these operations.

- Net slope is a power of 2

  Binary-point-only scaling, where the net slope is a power of 2, involves moving the binary point within the fixed-point word. This scaling mode already minimizes the number of processor arithmetic operations.

How to Use Integer Division to Handle Net Slope Correction

To enable this optimization:

1. Select the **Optimization > Use integer division to handle net slopes that are reciprocals of integers** configuration parameter.

   For more information, see "Use integer division to handle net slopes that are reciprocals of integers" in the Simulink Graphical User Interface.

2. On the **Hardware Implementation > Embedded Hardware** pane, set the **Signed integer division rounds to** configuration parameter to `Floor` or `Zero`, as appropriate for your target hardware. The optimization does not occur if the **Signed integer division rounds to** parameter is `Undefined`.

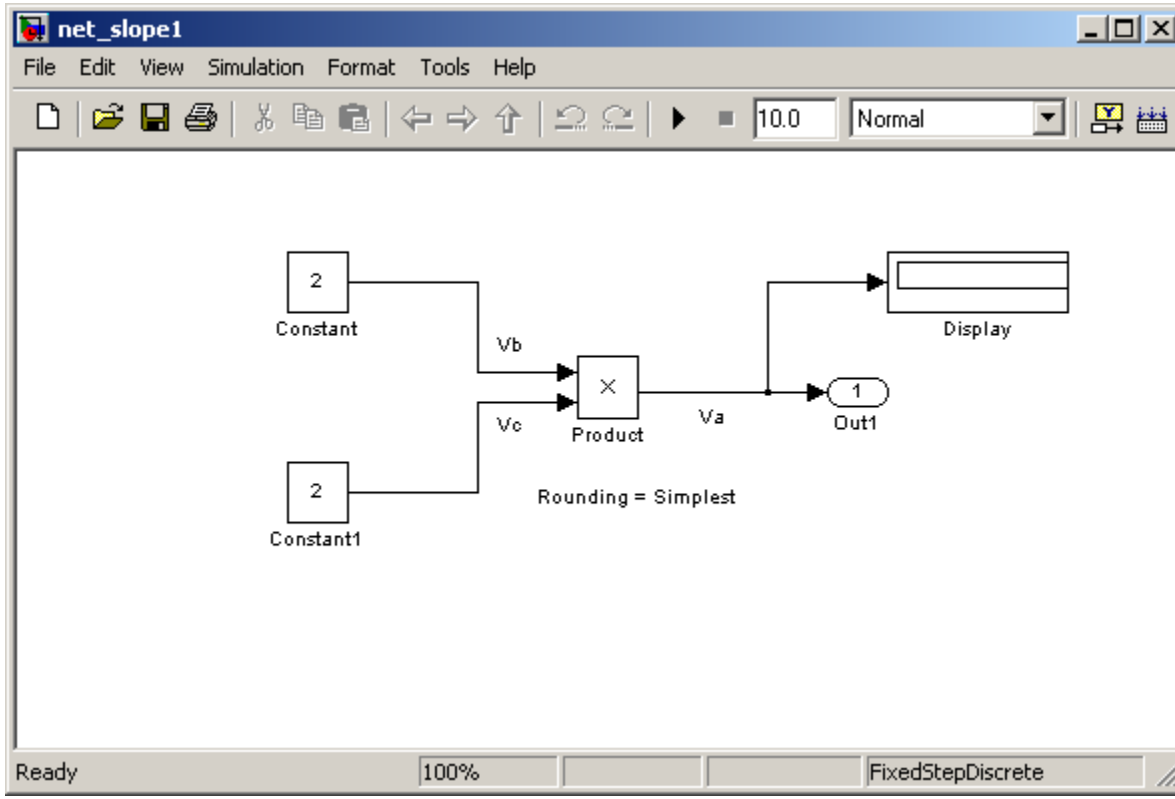
Note You must set this parameter to a value that is appropriate for the target hardware. Failure to do so might result in division operations that comply with the definition on the **Hardware Implementation** pane, but are inappropriate for the target hardware.

3 Set the **Integer rounding mode** of the blocks that require net slope correction (for example, Product, Gain, and Data Type Conversion) to `Simplest` or match the rounding mode of your target hardware.

Note You can use the Model Advisor to alert you if you have not configured your model correctly for this optimization. Open the Model Advisor and run the **Identify questionable fixed-point operations** check. For more information, see "Optimize Net Slope Correction" on page 11-44.

Example: Improving Numerical Accuracy of Simulation Results

This example illustrates how selecting the **Use integer division to handle net slopes that are reciprocals of integers** optimization parameter improves numerical accuracy. It uses the following model.



For the Product block in this model,

$$V_a = V_b \times V_c$$

These values are represented by the general [Slope Bias] encoding scheme described in "Scaling" on page 2-5: $V_i = S_i Q_i + B_i$.

Because there is no bias for the inputs or outputs:

$S_a Q_a = S_b Q_b . S_c Q_c$, or

$$Q_a = \frac{S_b S_c}{S_a} . Q_b Q_c$$

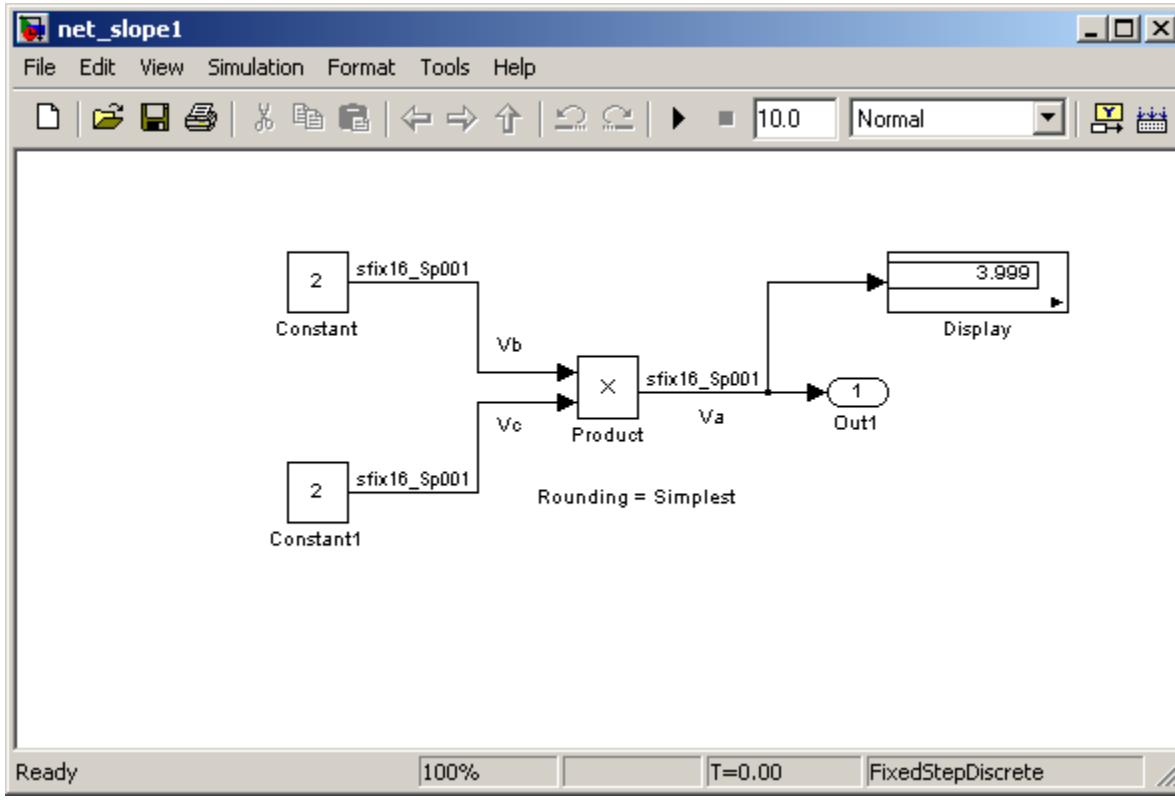
where the net slope is:

$$\frac{S_b S_c}{S_a}$$

The net slope for the Product block is `1/1000`. Because the net slope is the reciprocal of an integer, you can use the **Use integer division to handle net slopes that are reciprocals of integers** optimization parameter if your model and hardware configuration are suitable. For more information, see "When to Use Integer Division to Handle Net Slope Correction" on page 11-14.

To set up the model and run the simulation:

1. For the two Constant blocks, set the **Output data type** to `fixdt(1, 16, 1/1000, 0)`.
2. For the Product block, set the **Output data type** to `fixdt(1, 16, 1/1000, 0)`. Set the **Integer rounding mode** to `Simplest`.
3. Set the **Hardware Implementation > Embedded Hardware > Signed integer division rounds to** configuration parameter to `Zero`.
4. Clear the **Optimization > Use integer division to handle net slopes that are reciprocals of integers** configuration parameter.
5. In your Simulink model window, select **Simulation > Start**.



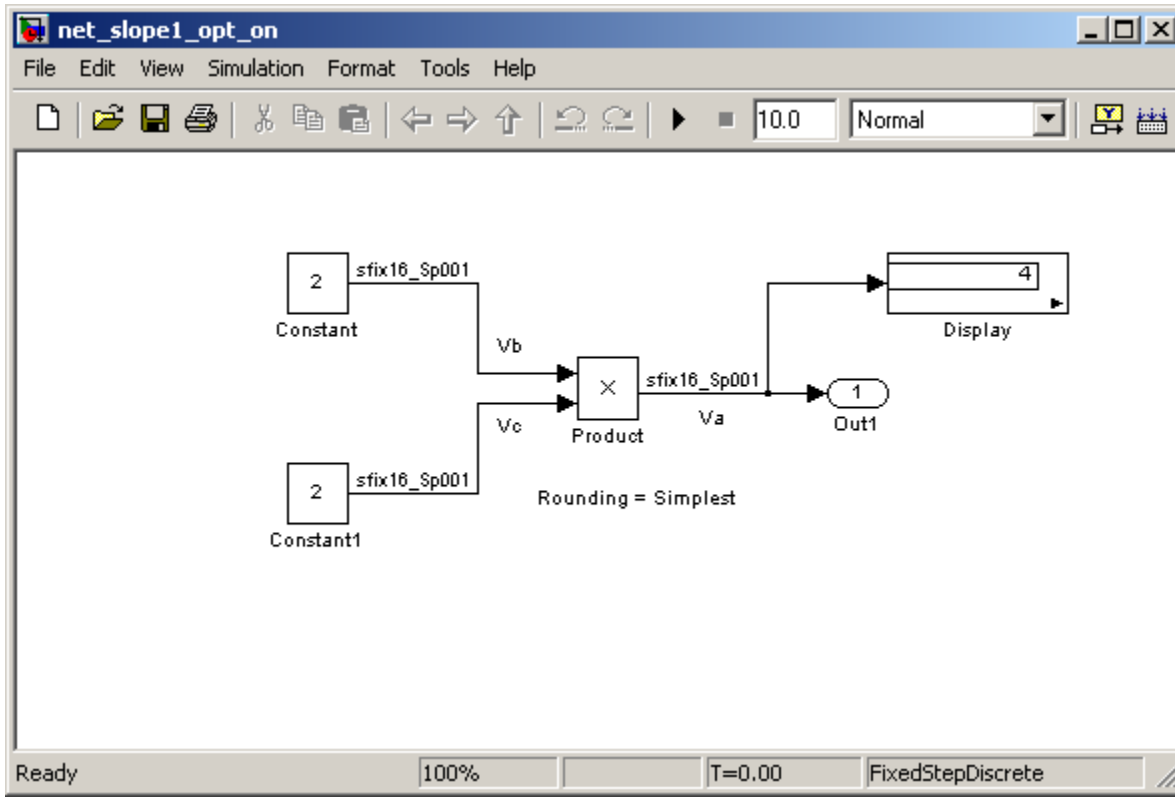
Because the simulation uses multiplication followed by shifts to handle the net slope correction, net slope precision loss occurs. This precision loss results in numerical inaccuracy: the calculated product is `3.999`, not `4`, as you expect.

Note You can set up the Simulink Fixed Point software to provide alerts when precision loss occurs in fixed-point constants. For more information, see "Net Slope and Net Bias Precision" on page 3-22.

---

6 Select the **Optimization > Use integer division to handle net slopes that are reciprocals of integers** configuration parameter, save your model, and simulate again.

The software implements the net slope correction using division instead of multiplication followed by shifts. The calculated product is 4, as you expect.



The optimization works for this model because:

- The net slope is a reciprocal of an integer.
- The **Hardware Implementation > Embedded Hardware > Signed integer division rounds to** configuration parameter is set to `Zero`.

Note This setting must match your target hardware rounding mode.

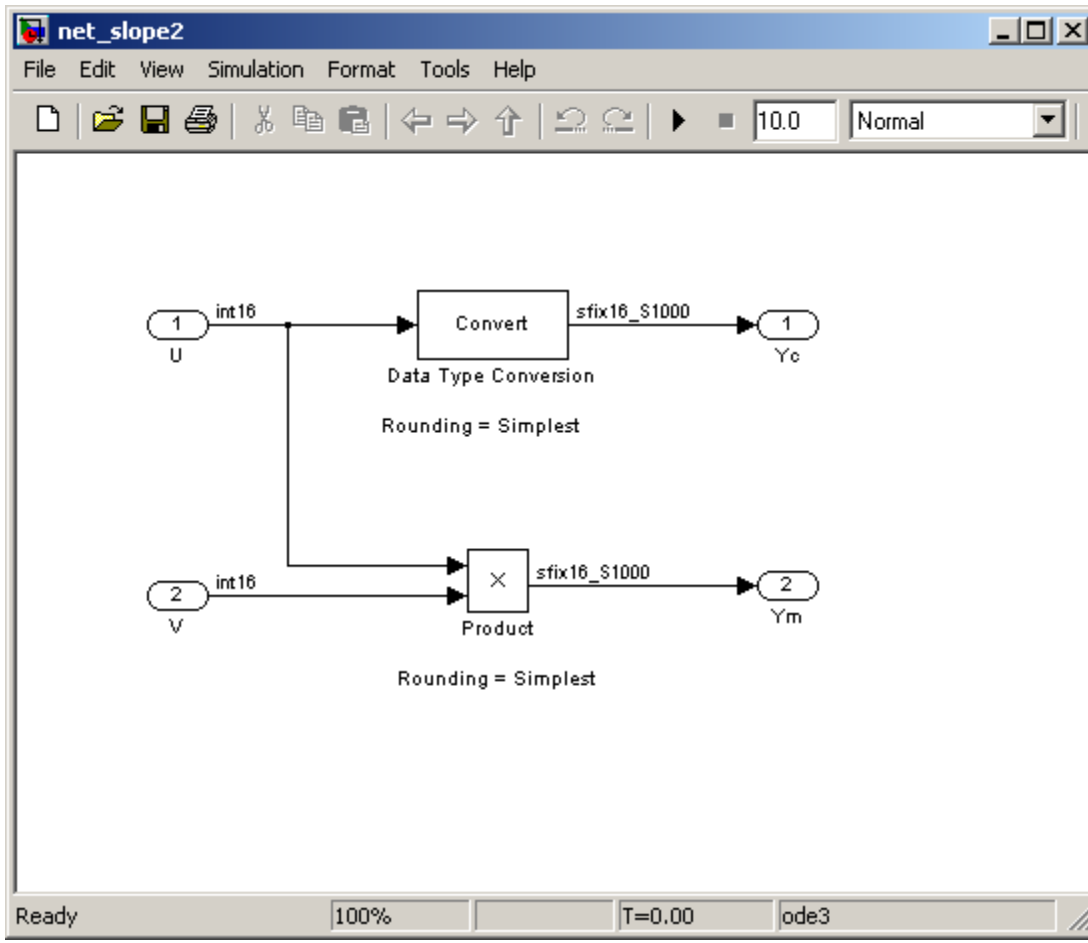
- The **Integer rounding mode** of the Product block in the model is set to `Simplest`.
- The model does not use multiword operations.

Example: Improving Efficiency of Generated Code

This example illustrates how selecting the **Use integer division to handle net slope correction** optimization parameter improves the efficiency of generated code.

Note The generated code is more efficient only if division is more efficient than multiplication followed by shifts on your target hardware.

This example uses the following model.



For the Product block in this model,

$$V_a = V_b \times V_c$$

These values are represented by the general [Slope Bias] encoding scheme described in "Scaling" on page 2-5: $V_i = S_i Q_i + B_i$.

Because there is no bias for the inputs or outputs:

$S_a Q_a = S_b Q_b . S_c Q_c$, or

$$Q_a = \frac{S_b S_c}{S_a} . Q_b Q_c$$

where the net slope is:

$$\frac{S_b S_c}{S_a}$$

The net slope for the Product block is `1/1000`.

Similarly, for the Data Type Conversion block in this model,

$$S_a Q_a + B_a = S_b Q_b + B_b$$

There is no bias. Therefore, the net slope is $\frac{S_b}{S_a}$. The net slope for this block is also `1/1000`.

Because the net slope is the reciprocal of an integer, you can use the **Use integer division to handle net slopes that are reciprocals of integers** optimization parameter if your model and hardware configuration are suitable. For more information, see "When to Use Integer Division to Handle Net Slope Correction" on page 11-14.

To set up the model and generate code:

1. For the two Inport blocks, `U` and `V`, set the **Data type** to `int16`.
2. For the Data Type Conversion block, set the **Integer rounding mode** to `Simplest`. Set the **Output data type** to `fixdt(1, 16, 1000, 0)`.
3. For the Product block, set the **Integer rounding mode** to `Simplest`. Set the **Output data type** to `fixdt(1, 16, 1000, 0)`.
4. Set the **Hardware Implementation > Embedded Hardware > Signed integer division rounds to** configuration parameter to `Zero`.
5. Clear the **Optimization > Use integer division to handle net slopes that are reciprocals of integers** configuration parameter.

6 From the Simulink model menu, select **Tools > Code Generation > Build Model**.

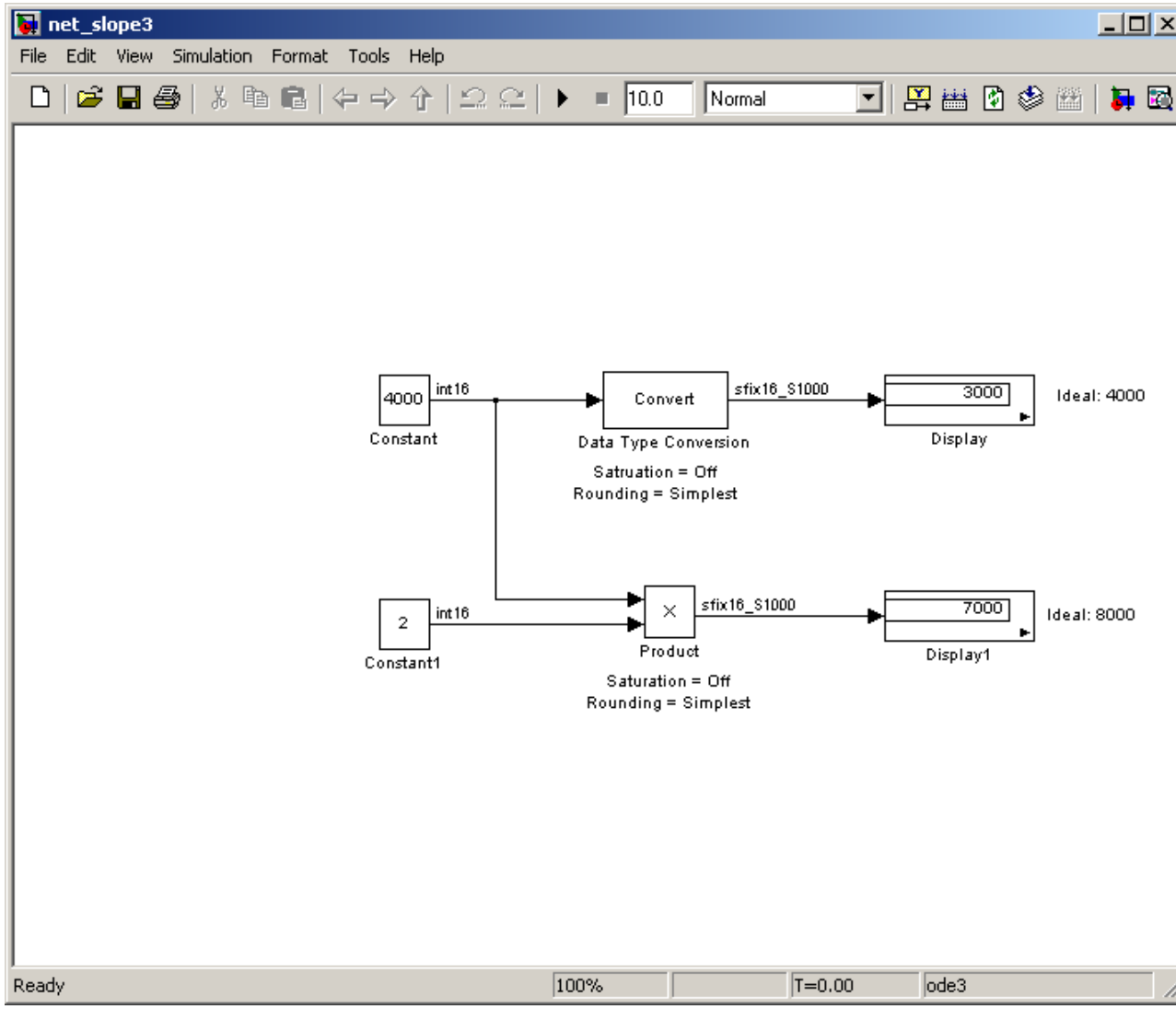
Conceptually, the net slope correction is `1/1000` or `0.001`:

```
Yc = 0.001 * U;
Ym = 0.001 * U * V;
```

The generated code uses multiplication with shifts:

```
% For the conversion
Yc = (int16_T)U * 16777 >> 24;
% For the multiplication
Ym = (int16_T)((int16_T)(U * V >> 10) * 16777 >> 14);
```

The ideal value of the net slope correction is `0.001`. In the generated code, the approximate value of the net slope correction is `16777L >> 24 = 16777/2^24 = 0.000999987125396729`. This approximation introduces numerical inaccuracy. For example, using the same model with constant inputs produces the following results.

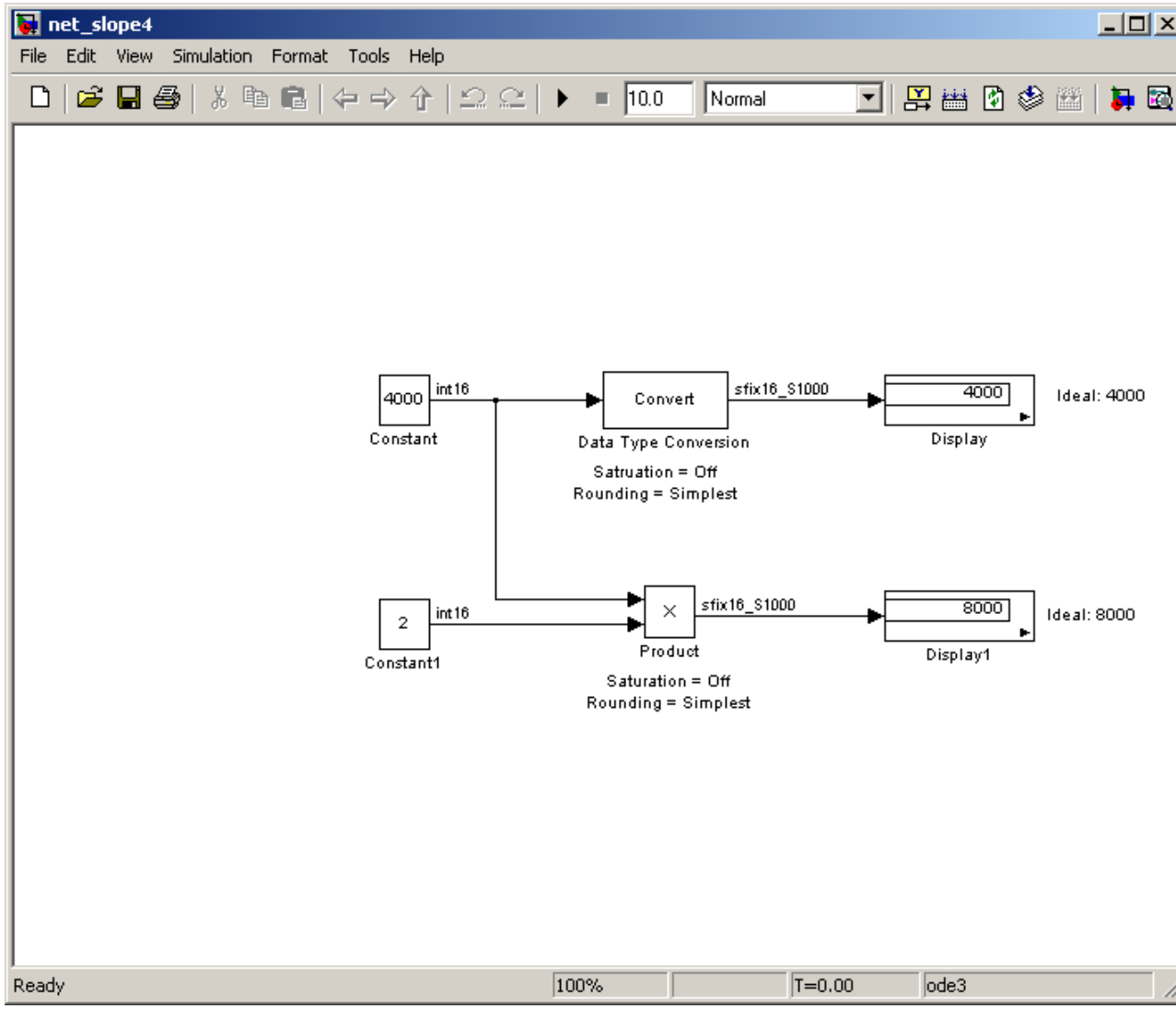


7 Select the **Optimization > Use integer division to handle net slopes that are reciprocals of integers** optimization parameter, update diagram, and generate code again.

The generated code now uses integer division instead of multiplication followed by shifts:

```
% For the conversion
Yc = (int16_T)(U / 1000);
% For the multiplication
Ym = (int16_T)(U * V / 1000);
```

8 In the generated code, the value of the net slope correction is now the ideal value of `0.001`. Using division, the results are numerically accurate.



The optimization works for this model because the:

- Net slope is a reciprocal of an integer.
- **Hardware Implementation > Embedded Hardware > Signed integer division rounds to** configuration parameter is set to `Zero`.

Note This setting must match your target hardware rounding mode.

- For the Product and Data Type Conversion blocks in the model, the **Integer rounding mode** is set to `Simplest`.
- Model does not use multiword operations.

Optimize Generated Code Using Specified Minimum and Maximum Values

The Simulink Fixed Point software uses representable minimum and maximum values and constant values to determine if it is possible to optimize the generated code, for example, by eliminating unnecessary utility functions and saturation code from the generated code.

This optimization results in:

- Reduced ROM and RAM consumption
- Improved execution speed

When you select the **Optimize using specified minimum and maximum values** configuration parameter, the software takes into account input range information, also known as *design minimum and maximum*, that you specify for signals and parameters in your model. It uses these minimum and maximum values to derive range information for downstream signals in the model and then uses this derived range information to simplify mathematical operations in the generated code whenever possible.

Prerequisites

The **Optimize using specified minimum and maximum values** parameter appears for ERT-based targets only and requires an Embedded Coder license when generating code.

How to Configure Your Model

To make optimization more likely:

- Provide as much design minimum and maximum information as possible. Specify minimum and maximum values for signals and parameters in the model for:
  - Inport and Outport blocks
  - Block outputs
  - Block inputs, for example, for the MATLAB Function and Stateflow Chart blocks
  - `Simulink.Signal` objects
- Ensure that the minimum and maximum values for signals and parameters are accurate and trustworthy. Otherwise, optimization might result in numerical mismatch with simulation.

  Before generating code, test the specified values by simulating your model with simulation range checking enabled. If errors or warnings occur, fix these issues before generating code.

  #### How to Enable Simulation Range Checking

  1. In your model, select **Simulation > Configuration Parameters** to open the Configuration Parameters dialog box.
  2. In the Configuration Parameters dialog box, select **Diagnostics > Data Validity**.
  3. On the **Data Validity** pane, under **Signals**, set **Simulation range checking** to `warning` or `error`.
- Use fixed-point data types with binary-point-only (power-of-two) scaling.
- Provide design minimum and maximum information upstream of blocks as close to the inputs of the blocks as possible. If you specify minimum and maximum values for a block output, these values are most likely to affect the outputs of the blocks immediately downstream. For more information, see "Example: Using Specified Minimum and Maximum Values to Eliminate Unnecessary Utility Functions" on page 11-30.

How to Enable Optimization

1. Set the **Code Generation > System target file** to select an Embedded Real-Time (`ERT`) target (requires an Embedded Coder license).

2. Specify design minimum and maximum values for signals and parameters in your model using the tips in "How to Configure Your Model" on page 11-28.

3. Select the **Optimization > Optimize using specified minimum and maximum values** configuration parameter.

   For more information, see "Optimize using the specified minimum and maximum values" in the Simulink Graphical User Interface.

Example: Using Specified Minimum and Maximum Values to Eliminate Unnecessary Utility Functions

This example demonstrates how the Simulink Fixed Point software uses the input range for a division operation to determine whether it can eliminate unnecessary utility functions from the generated code. It uses the `fxpdemo_min_max_optimization` demo model. First, you generate code without using the specified minimum and maximum values to see that the generated code contains utility functions to ensure that division by zero does not occur. You then turn on the optimization, and generate code again. With the optimization, the generated code does not contain the utility function because it is not necessary for the input range.

Generate Code Without Using Minimum and Maximum Values. First, generate code without taking into account the design minimum and maximum values for the first input of the division operation to show the code without the optimization. In this case, the software uses the representable ranges for the two inputs, which are both `uint16`. With these input ranges, it is not possible to implement the division with the specified precision using shifts, so the generated code includes a division utility function.

1. Run the demo. At the MATLAB command line, enter:

   ```
   fxpdemo_min_max_optimization
   ```

2. In the demo window, double-click the **View Optimization Configuration** button.

   The Optimization pane of the Configuration Parameters dialog box appears.

Note that the **Optimize using specified minimum and maximum values** parameter is not selected.

3 Double-click the **Generate Code** button.

The code generation report appears.

4 In the model, right-click the `Division with increased fraction length output type` block.

The context menu appears.

5 From the context menu, select **Code Generation > Navigate to code**.

The code generation report highlights the code generated for this block. The generated code includes a call to the `div_repeat_u32` utility function.

```
rtY.Out3 = div_repeat_u32((uint32_T)rtU.In5 << 16,
  (uint32_T)rtU.In6, 1U);
```

6 Click the `div_repeat_u32` link to view the utility function, which contains code for handling division by zero.

Generate Code Using Minimum and Maximum Values. Next, generate code for the same division operation, this time taking into account the design minimum and maximum values for the first input of the Product block. These minimum and maximum values are specified on the Inport block directly upstream of the Product block. With these input ranges, the generated code implements the division by simply using a shift. It does not need to generate a division utility function, reducing both memory usage and execution time.

1 Double-click the Inport block labelled `5` to open the block parameters dialog box.

2 On the block parameters dialog box, select the **Signal Attributes** pane and note that:

- The **Minimum** value for this signal is `1`.
- The **Maximum** value for this signal is `100`.

3 Click **OK** to close the dialog box.

4 Double-click the **View Optimization Configuration** button.

   The Optimization pane of the Configuration Parameters dialog box appears.

5 On this pane, select the **Optimize using specified minimum and maximum values** parameter and click **Apply**.

6 Double-click the **Generate Code** button.

   The code generation report appears.

7 In the model, right-click the `Division with increased fraction length output type` block.

   The context menu appears.

8 From the context menu, select **Code Generation > Navigate to code**.

   The code generation report highlights the code generated for this block. This time the generated code implements the division with a shift operation and there is no division utility function.

```
tmp = rtU.In6;
rtY.Out3 = (uint32_T)tmp ==
  (uint32_T)0 ? MAX_uint32_T : ((uint32_T)rtU.In5 << 17) /
    (uint32_T)tmp;
```

Modify the Specified Minimum and Maximum Values. Finally, modify the minimum and maximum values for the first input to the division operation so that its input range is too large to guarantee that the value does not overflow when shifted. Here, you cannot shift a 16-bit number 17 bits to the right without overflowing the 32-bit container. Generate code for the division operation, again taking into account the minimum and maximum values. With these input ranges, the generated code includes a division utility function to ensure that no overflow occurs.

1 Double-click the Inport block labelled `5` to open the block parameters dialog box.

2 On the block parameters dialog box, select the **Signal Attributes** pane and set the **Maximum** value to `40000`, then click **OK** to close the dialog box.

3. Double-click the **Generate Code** button.

   The code generation report appears.

4. In the model, right-click the `Division with increased fraction length output type` block.

   The context menu appears.

5. From the context menu, select **Code Generation > Navigate to code**.

   The code generation report highlights the code generated for this block. The generated code includes a call to the `div_repeat_32` utility function.

```
rtY.Out3 = div_repeat_u32((uint32_T)rtU.In5 << 16,
  (uint32_T)rtU.In6, 1U);
```

Limitations

- This optimization does not occur for:
  - Multiword operations
  - Fixed-point data types with slope and bias scaling
  - Addition unless the fraction length is zero
- This optimization does not take into account minimum and maximum values for:
  - Merge block inputs. To work around this issue, use a `Simulink.Signal` object on the Merge block output and specify the range on this object.
  - Bus elements.
  - Conditionally-executed subsystem (such as a triggered subsystem) block outputs that are directly connected to an Outport block.

    Outport blocks in conditionally-executed subsystems can have an initial value specified for use only when the system is not triggered. In this case, the optimization cannot use the range of the block output because the range might not cover the initial value of the block.

- There are limitations on precision because you specify the minimum and maximum values as double-precision values. If the true value of a minimum or maximum value cannot be represented as a double, ensure that you round the minimum and maximum values correctly so that they cover the true design range.
- If your model contains multiple instances of a reusable subsystem and each instance uses input signals with different specified minimum and maximum values, this optimization might result in different generated code for each subsystem so code reuse does not occur. Without this optimization, the Simulink Coder software generates code once for the subsystem and shares this code among the multiple instances of the subsystem.

Optimizing Your Generated Code with the Model Advisor

In this section...

Introduction

You can use the Simulink Model Advisor to help you configure your fixed-point models to achieve a more efficient design and optimize your generated code. To use the Model Advisor to check your fixed-point models:

1. From the **Tools** menu of the model you want to analyze, select **Model Advisor**. The Model Advisor appears.
2. In the left pane, expand the **By Product** node and select **Embedded Coder**.
3. From the Model Advisor **Edit** menu, select **Select All** to enable all Model Advisor checks associated with the selected node. For fixed-point code generation, the most important check boxes to select are **Identify questionable fixed-point operations**, **Identify blocks that generate expensive saturation and rounding code**, and **Check the hardware implementation**.
4. Click **Run Selected Checks**. Any tips for improving the efficiency of your fixed-point model appear in the Model Advisor window.

The sections that follow discuss possible messages that might be returned when you use the Model Advisor check titled **Identify questionable fixed-point operations**. The sections explain the messages, discuss their importance in fixed-point code generation, and offer suggestions for tweaking your model to optimize the code.

Optimize Lookup Table Data

Efficiency trade-offs related to lookup table data are described in "Effects of Spacing on Speed, Error, and Memory Usage" on page 8-22. Based on these trade-offs, the Model Advisor identifies blocks where there is potential for efficiency improvements. Messages like the following are shown in the browser to alert you to these cases:

- Lookup table input data is not evenly spaced. An evenly spaced table might be more efficient. See `fixpt_look1_func_approx`.
- The lookup table input data is *not* evenly spaced when quantized, but it is very close to being evenly spaced. If the data is not tunable, then it is strongly recommended that you consider adjusting the table to be evenly spaced. See `fixpt_evenspace_cleanup`.
- Lookup table input data is evenly spaced, but the spacing is not a power of two. A simplified implementation could result if the table could be reimplemented with even power-of-two spacing. See `fixpt_look1_func_approx`.

Reduce Cumbersome Multiplications

"Targeting an Embedded Processor" on page 4-4 discusses the capabilities and limitations of embedded processors. "Design Rules" on page 4-5 recommends that inputs to a multiply operation should not have word lengths larger than the base integer type of your processor. Multiplication with larger word lengths can always be handled in software, but that approach requires much more code and is much slower. The Model Advisor identifies blocks where undesirable software multiplications are required. Visual inspection of the generated code, including the generated multiplication utility function, will make the cost of these operations clear. It is strongly recommended that you adjust the model to avoid these operations. Messages like the following are shown in the browser to alert you to this situation:

- A very cumbersome multiplication is required by this block. The first input has 8 bits. The second input has 32 bits. The ideal product has 40 bits. The largest integer size for the target has only 32 bits. Saturation is ON, so it is necessary to determine all 40 bits of the ideal product in the C code. The C code required to do this multiplication is large and slow. For this target, restricting multiplications to 16 bits times 16 bits is strongly recommended.
- A very cumbersome multiplication is required by this block. The first input has 8 bits. The second input has 32 bits. The ideal product has 40 bits. The largest integer size for the target has only 32 bits. The relative scaling of the inputs and the output requires that some of the 8 most significant bits of the ideal product be determined in the C code. The C code required to do this multiplication is large and slow. For this target, restricting multiplications to 16 bits times 16 bits is strongly recommended.

Optimize the Number of Multiply and Divide Operations

The number of multiplications and divisions that a block performs can have a big impact on accuracy and efficiency. The Model Advisor detects some, but not all, situations where rearranging the operations can improve accuracy, efficiency, or both.

One such situation is when a calculation using more than one division operation is computed. A browser message will identify Product blocks that are doing multiple divisions. Note that multiple divisions spread over a series of blocks are not detected by Model Advisor:

- This Product block is configured to do more than one division operation. A general guideline from the field of numerical analysis is to multiply all the denominator terms together first, then do one and only one division. This improves accuracy and often speed in floating-point and especially fixed-point. This can be accomplished in Simulink by cascading Product blocks.

Another situation is when a single Product block is configured to do more than one multiplication or division operation. A browser message will identify Product blocks doing multiple operations:

- This Product block is configured to do more than one multiplication or division operation. This is supported, but if the output data type is

integer or fixed-point, then better result are likely if this operation is split across several blocks each doing one multiplication or one division. Using several blocks allows the user to control the data type and scaling used for intermediate calculations. The choice of data types for intermediate calculations affects precision, range errors, and efficiency.

Reduce Multiplies and Divides with Nonzero Bias

"Rules for Arithmetic Operations" on page 3-50 discusses the implementation details of fixed-point multiplication and division. That section shows the significant increase in complexity that occurs when signals with nonzero biases are involved in multiplication and division. The Model Advisor puts a message in the browser that identifies blocks that require these complicated operations. It is strongly recommended that you make changes to eliminate the need for these complicated operations:

- This block is multiplying signals with nonzero bias. It is recommended that this be avoided when possible. Extra steps are required to implement the multiplication (if possible). Inserting a Data Type Conversion block before and after the block doing the multiplication allows the biases to be removed and allows the user to control data type and scaling for intermediate calculations. In many cases the Data Type Conversion blocks can be move to the "edges" of a (sub)system. The conversion is only done once and all blocks can benefit from simpler bias-free math.

Eliminate Mismatched Scaling

Scaling adjustment is an extremely common operation in fixed-point designs. In the vast majority of cases, shifts left or shifts right are sufficient to handle the scaling adjustment. This occurs when the slope adjustment is an exact power of two, and the bias adjustment term is zero. Situations where shifts are not sufficient to handle scaling adjustments are called mismatched scaling. Cases of mismatched scaling can involve either mismatched slopes or mismatched biases.

For mismatched slopes, it is necessary to multiply by an integer correction term in addition to shifting. The need for this extra multiplication often represents a design oversight. The extra multiplication requires extra code, slows down the speed of execution, and usually introduces additional precision loss. By adjusting the scaling of the inputs or outputs, you can eliminate mismatched slopes. The most efficient designs minimize the number of places

where mismatched slopes occur. The need to handle mismatched slopes can occur in many Simulink blocks, including Product, Sum, Relational Operator, and MinMax. A browser message will identify these blocks. The Data Type Conversion block can also face mismatched slopes, but it is assumed that this explicit conversion is intentional, so no Model Advisor messages are issued:

- This block is multiplying signals with mismatched slope adjustment terms. The first input has slope adjustment 1.01. The second input has slope adjustment 1. The output has slope adjustment 1. The net slope adjustment is 1.01. This mismatch causes the overall operation to involve two multiply instructions rather than just one as expected. The mismatch can be removed by changing the data type of the output.
- This Sum block has a mismatched slope adjustment term between an input and the output. The input has slope adjustment 1.5. The output has slope adjustment 1. The net slope adjustment is 1.5. This mismatch causes the Sum block to require a multiply operation each time the input is converted to the outputs data type and scaling. The mismatch can be removed by changing the scaling of the output or the input.
- This MinMax block has mismatched slope adjustment terms between an input and the output. The input has slope adjustment 1.125. The output has slope adjustment 1. The net slope adjustment is 1.125. This mismatch causes the MinMax block to require a multiply operation each time the input is converted to the data type and scaling of the output. The mismatch can be removed by changing the scaling of either the input or output.
- This Relational Operator block has mismatched slope adjustment terms between the first and second input. The first input has slope adjustment 1. The second input has slope adjustment 1.125. The net slope adjustment is 1.125. This mismatch causes the relational operator block to require a multiply operation each time the nondominant input is converted to the data type and scaling of the dominant input. The mismatch can be removed by changing the scaling of either of the inputs.

For mismatched bias, it is usually necessary to add or subtract an integer correction term as a separate step in addition to the normal shifting. Like slope mismatch, the need to do this extra addition often represents a design oversight. Except for the Data Type Conversion block, Model Advisor assumes mismatched bias is an oversight. A message such as the following appears

in the browser, identifying blocks that could be made more efficient by eliminating mismatched biases:

- For this Sum block, the addition and subtraction of the input biases do not cancel with the output bias. The implementation will include one extra addition or subtraction instruction to correctly account for the net bias adjustment. Changing the bias of the output scaling can make the net bias adjustment zero and eliminate the need for the extra operation.

Minimize Internal Conversion Issues

Many fixed-point operations need to do internal data type and scaling conversions. Fixed-point operations are based upon lower-level operations, such as integer addition and integer comparisons, that require the arguments to have the same data type and scaling. This is why blocks built on these operations, such as Sum, Relational Operator, and MinMax, must do internal conversions. There can be issues related to these internal conversions, such as range errors, that lead to overflows and loss of efficiency. Model Advisor warns separately about these two issues with messages like the following:

- For this Relational Operator block, the first input has the greater positive range. The second input is converted to the data type and scaling of the first input prior to performing the relational operation. The first input has range 0 to 255.996 but the second input has range -4 to 3.96875 so a range error can occur when casting.
- For this MinMax block, an input is converted to the data type and scaling of the output prior to performing the relational operation. The input has range 0 to 255.996 but the output has range -256 to 255.992, so a range error can occur when casting.
- For this Relational Operator block, the second input has the greater positive range. The first input is converted to the data type and scaling of the second input prior to performing the relational operation. The first input has range -4 to 3.96875 but the second input has range 0 to 255.996, so a range error can occur when casting.
- The Sum block can have a range error prior to the addition or subtraction operation being performed. For simplicity of design, the sum block always casts each input to the output's data type and scaling prior to performing the addition or subtraction. One of the inputs has range -128 to 127.996 but the output has range -32 to 31.999 so a range error can occur when

casting the input to the outputs data type. Users can get any addition subtraction their application requires by inserting data type conversion blocks before and/or after the sum block. For example, suppose the inputs were a combination of signed and unsigned 8 bits with binary points that differed by at most 5 places. The output of the sum block could be set to signed 16 bit with scaling that matched the most precise input. When the inputs were cast to the outputs data type there would be no loss of range or precision. A conversion block after the sum block would allow the final result to be put in whatever data type was desired.

- The Sum block can have a range error prior to the addition or subtraction operation being performed. For simplicity of design, the sum block always casts each input to the output's data type and scaling prior to performing the addition or subtraction. Note, for better accuracy and efficiency, nonzero bias terms are handled separately and are not included in the conversion from input to output. The ranges given below for the input and output exclude their biases. One of the inputs has range -4 to 3.96875 but the output has range 0 to 63.999 so a range error can occur when casting the input to the outputs data type. Users can get any addition subtraction their application requires by inserting data type conversion blocks before and/or after the sum block. For example, suppose the inputs were a combination of signed and unsigned 8 bits with binary points that differed by at most 5 places. The output of the sum block could be set to signed 16 bit with scaling that matched the most precise input. When the inputs were cast to the outputs data type there would be no loss of range or precision. A conversion block after the sum block would allow the final result to be put in whatever data type was desired.

For some operations, the need to do an internal conversion can represent a design oversight. The impact of this oversight is a loss of efficiency, and possibly a loss of accuracy. As an example, consider the comparison of a signal against a constant using a Relational Operator block. To compare a fixed-point signal against a constant, the underlying implementation should directly compare the stored integer of the input signal against an invariant stored integer. If the scaling or data type of the signal and constant are different, then it is also necessary to do a conversion operation. This extra conversion work is usually inefficient and is often unexpected. The Model Advisor warns about these situations with messages like the following:

- For this MinMax block, an input is converted to the data type and scaling of the output prior to performing the relational operation. The input has precision 0.00390625. The output has precision 0.0078125, so there can be a precision loss each time the conversion is performed.
- For this relational operator block, the data types of the first and second inputs are not the same. A conversion operation is required every time the block is executed. If one of the inputs is invariant (sample time color magenta), then changing the data type and scaling of the invariant input to match the other input is a good opportunity for improving the efficiency of your model.
- For this MinMax block, the data types of the output and an input are not the same. A conversion operation is required every time the block is executed.

Use the Most Efficient Rounding

How to Specify Rounding for Fixed-Point Operations

Specify rounding options for fixed-point operations by setting a combination of these parameters:

- **Integer rounding mode**

  Use the **Integer rounding mode** parameter on your model's blocks to simulate the rounding behavior of the C compiler that you use to compile code generated from the model. This setting appears on the **Signal Attributes** pane of the parameter dialog boxes of blocks that can perform signed integer arithmetic, such as the Product block.

- **Signed integer division rounds to** parameter

  This parameter is available from **Configuration Parameters > Hardware Implementation > Embedded Hardware**. It describes how to produce a signed integer quotient for the production hardware. For most blocks, the value of **Integer rounding mode** completely defines rounding behavior. For blocks that support fixed-point data and the `Simplest` rounding mode, the value of **Signed integer division rounds to** also affects rounding. For details, see Simplest rounding.

How to Choose the Most Efficient Rounding

Traditional handwritten code, especially for control applications, almost always uses "no effort" rounding. For example, for unsigned integers and two's complement signed integers, shifting right and dropping the bits is equivalent to rounding to floor. To get results comparable to, or better than, what you expect from traditional handwritten code, use the simplest rounding mode. In general the simplest mode provides the minimum cost solution with no overflows. If the simplest mode is not available, round to floor.

The primary exception to this rule is the rounding behavior of signed integer division. The C standard leaves this rounding behavior unspecified, but for most production targets the "no effort" mode is to round to zero. For unsigned division, everything is nonnegative, so rounding to floor and rounding to zero are identical. To improve rounding efficiency, set **Configuration Parameters > Hardware Implementation > Embedded Hardware > Signed integer division rounds to** using the mode that your production target uses.

For more information on properties to consider when choosing a rounding mode, see "Choosing a Rounding Method" in the Fixed-Point Toolbox documentation.

Model Advisor Rounding Mode Checks

Use the Model Advisor to alert you when rounding optimizations are available.

1. From the **Tools** menu of the model you want to analyze, select **Model Advisor**. The Model Advisor appears.
2. In the left pane, expand the **By Product** node and select **Embedded Coder**.
3. Select **Identify blocks that generate expensive saturation and rounding code**.
4. Click **Run Selected Checks**. Any tips for improving the rounding efficiency of your fixed-point model appear in the Model Advisor window.

The Model Advisor alerts you when rounding optimizations are available.

- To obtain the most efficient generated code, change the **Integer rounding mode** parameter of the following block to `Simplest` or to `Floor` if `Simplest` is not available.
- Integer division generated code could be more efficient. C language standards do not fully specify the rounding behavior of signed integer division. When faced with this lack of specification, the code generated for division can be large to ensure bit-true agreement between simulation and code generation.

  Configuration Parameters > Hardware Implementation > Embedded Hardware > Signed integer division rounds to allows you to describe the rounding behavior of signed integer division for your production target. The rounding behavior for this model is currently set to `Undefined`. You can reduce the size of the code generated for division by setting this parameter. The most common behavior is that signed integer division rounds to zero.

Optimize Net Slope Correction

When a change of fixed-point slope is not a power of two, net slope correction is necessary. Normally, net slope correction is implemented using an integer multiplication followed by shifts. Under some conditions, an alternate implementation requires just an integer division by a constant. One of the conditions is that the net slope can be very accurately represented as the reciprocal of an integer. When this condition is met, the division implementation produces more accurate numerical behavior. Depending on your compiler and embedded hardware, the division implementation might be more desirable than the multiplication and shifts implementation. The generated code might be more efficient in either ROM size or model execution size.

How the Model Advisor Helps You Optimize Net Slope Correction

The Model Advisor alerts you when:

- You select the **Use integer division to handle net slopes that are reciprocals of integers** optimization parameter, but your model configuration is not compatible with this selection.

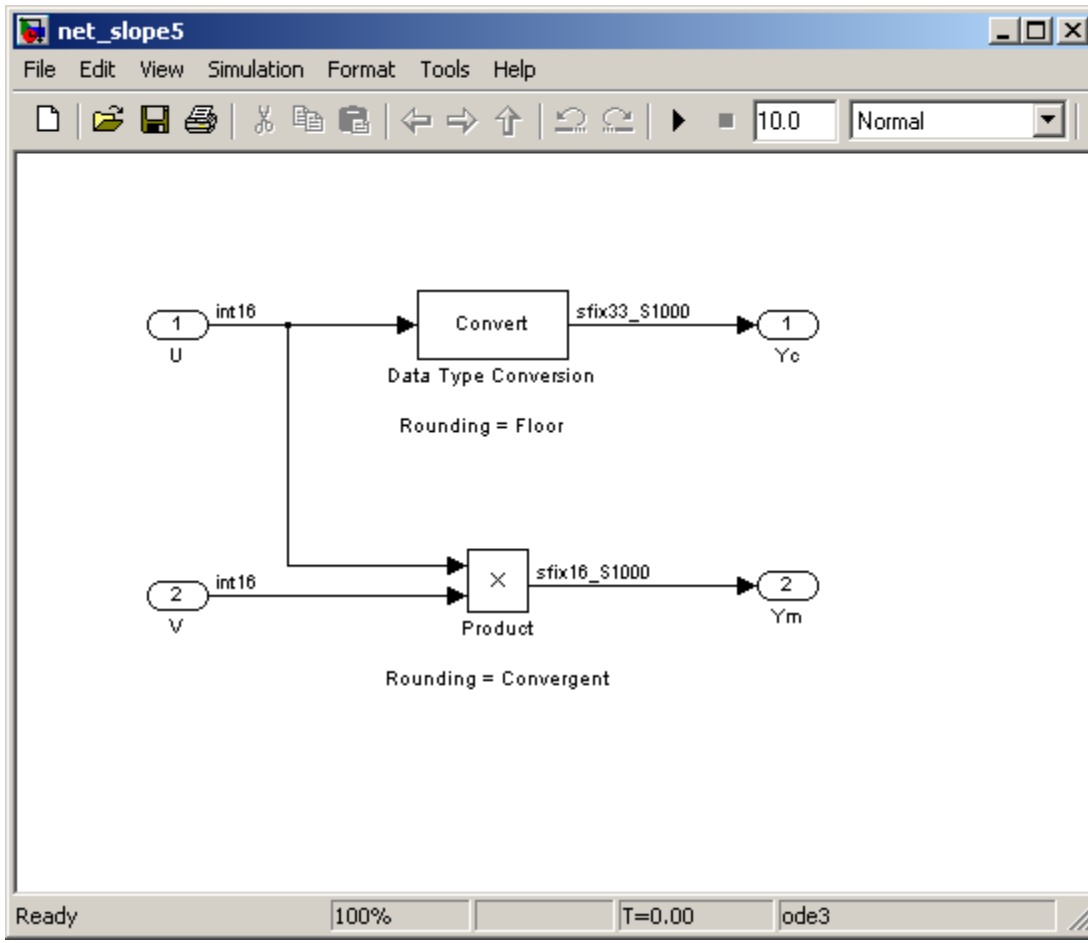
See "Example: Using the Model Advisor to Verify that Your Model Configuration is Suitable for Using Integer Division for Net Slope Correction" on page 11-45.

- Your model configuration is suitable for using integer division to handle net slope correction, but you do not select the **Use integer division to handle net slopes that are reciprocals of integers** optimization parameter.

  See "Example: Using the Model Advisor to Detect When to Use Integer Division for Net Slope Correction" on page 11-50.

Example: Using the Model Advisor to Verify that Your Model Configuration is Suitable for Using Integer Division for Net Slope Correction

This example uses the following model.



In this model, the net slope for the Data Type Conversion and Product blocks is `1/1000`.

To set up the model:

1. For the two Inport blocks, `U` and `V`, set the **Data type** to `int16`.
2. For the Data Type Conversion block, set the **Integer rounding mode** to `Floor`. Set the **Output data type** to `fixdt(1, 33, 1000, 0)`.

Note Setting the **Output data type** word length greater than the length of the `long` data type results in multiword operations.

---

3. For the Product block, set the **Integer rounding mode** to `Convergent`. Set the **Output data type** to `fixdt(1, 16, 1000, 0)`.

4. Set the **Hardware Implementation > Embedded Hardware > Signed integer division rounds to** configuration parameter to `Zero`.

5. Select the **Optimization > Use integer division to handle net slopes that are reciprocals of integers** configuration parameter.

6. Save the model.

To run the Model Advisor check:

1. From the model menu, select **Tools > Model Advisor**.

   The Model Advisor appears.

2. In the left pane, expand the **By Product** node and then expand the **Embedded Coder** node.

3. Select **Identify questionable fixed-point operations**.

4. Click **Run This Check**.

   The Model Advisor provides warnings that your model configuration is not compatible with the use of division for net slope correction. It also provides recommendations on how to change your model configuration to make it compatible:

   - The Product block is not using the correct rounding mode. Change the **Integer rounding mode** parameter to `Simplest` or to match the configuration parameter setting, **Hardware Implementation > Embedded Hardware > Signed integer division rounds to**.
   - The Data Type Conversion block is not using the correct rounding mode. Change the **Integer rounding mode** parameter to `Simplest`

or to match the configuration parameter setting, **Hardware Implementation > Embedded Hardware > Signed integer division rounds to**.

- The Data Type Conversion block is using multiword operations. Change the word length of the block to avoid multiword operations.

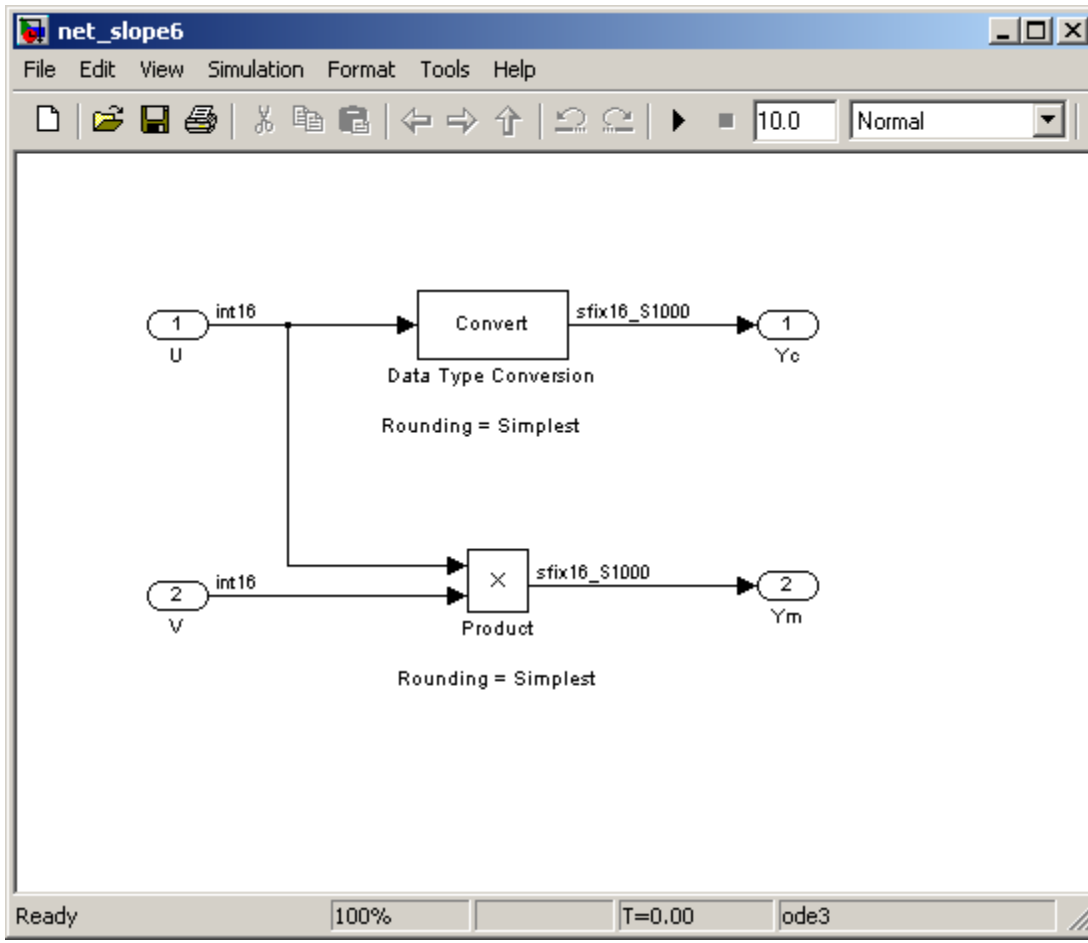
5 Make the suggested changes:

a For the Product and Data Type Conversion blocks, change the rounding mode to `Simplest`.

b For the Data Type Conversion block, change the **Output data type** from `fixdt(1, 33, 1000, 0)` to `fixdt(1, 16, 1000, 0)` to avoid multiword operations.

c Save the model.

This is your model configuration.

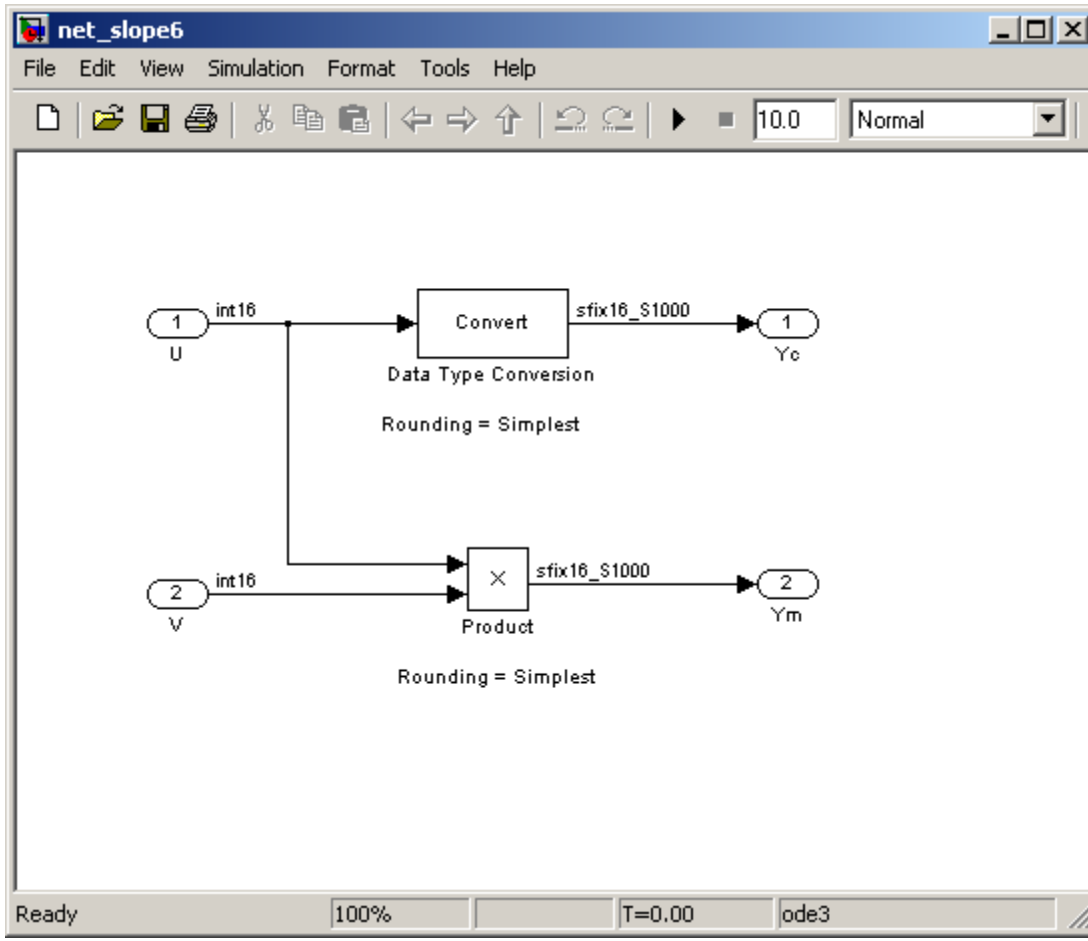


6 Rerun the Model Advisor **Identify questionable fixed-point operations** check.

The Model Advisor no longer reports the warnings about rounding mode and multiword operations. Your model configuration is compatible with using integer division to handle net slope correction.

Example: Using the Model Advisor to Detect When to Use Integer Division for Net Slope Correction

This example uses the following model.



In this model, the net slope for the Data Type Conversion and Product blocks is 1/1000.

To set up the model:

1. For the two Inport blocks, `U` and `V`, set the **Data type** to `int16`.
2. For the Data Type Conversion block , set the **Integer rounding mode** to `Simplest`. Set the **Output data type** to `fixdt(1, 16, 1000, 0)`.
3. For the Product block, set the **Integer rounding mode** to `Simplest`. Set the **Output data type** to `fixdt(1, 16, 1000, 0)`.
4. Set the **Hardware Implementation > Embedded Hardware > Signed integer division rounds to** configuration parameter to `Zero`.
5. Clear the **Optimization > Use integer division to handle net slopes that are reciprocals of integers** configuration parameter.
6. Save the model.

To run the Model Advisor check:

1. From the model menu, select **Tools > Model Advisor**.

   The Model Advisor appears.
2. In the left pane, expand the **By Product** node and then expand the **Embedded Coder** node.
3. Select **Identify questionable fixed-point operations**.
4. Click **Run This Check**.

   The Model Advisor warns that your model configuration is not optimal and provides the following recommendation:

   The Product and Data Type Conversion blocks are not using integer division for net slope correction. Selecting **Optimization > Use integer division to handle net slopes that are reciprocals of integers** might generate more efficient code.

Note The generated code is more efficient only if division is more efficient than multiplication followed by shifts on your target hardware.

---

5 Select the **Use integer division to handle net slopes that are reciprocals of integers** optimization parameter.

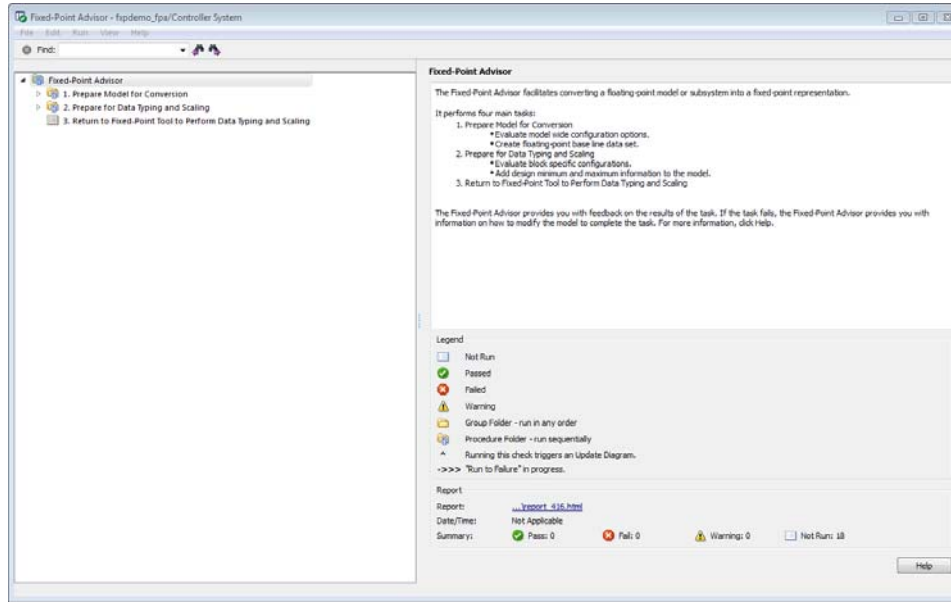
6 Rerun the Model Advisor **Identify questionable fixed-point operations** check.

The Model Advisor no longer reports the warning. Your model now uses integer division to handle net slope correction. This configuration results in more efficient code if division is more efficient than multiplication followed by shifts on your target hardware. For more information, see "Handle Net Slope Correction" on page 11-14.

12

Fixed-Point Advisor Reference

Fixed-Point Advisor



Fixed-Point Advisor Overview

The Fixed-Point Advisor is a tool you can use to prepare your model for conversion from an unknown floating-point data type to a known fixed-point data type. The Fixed-Point Advisor workflow allows you to complete your first iteration through the conversion process without accepting all the recommendations. However, before using the Fixed-Point Tool to autoscale your model using simulation data, you must accept all the recommendations.

Description

Use the Fixed-Point Advisor to:

- Set model-wide configuration options.
- Set block-specific dialog parameters.
- Check for unsupported blocks.

Procedures

Automatically Run Tasks. The following steps list how you can automatically run all tasks within a folder.

1. Click the **Run to Failure** button. The tasks run in order until a task fails.
2. Fix the failure:
   - Manually fix the problem using the **Explore Result** button, if present.
   - Manually fix the problem by modifying the model as instructed in the Analysis Result box.
   - Automatically fix the problem using the **Modify All** button, if available.
3. Continue the run to failure by selecting **Run > Continue**.

Run Individual Tasks. The following steps list how you can run an individual task.

1. Specify **Input Parameters**, if present.
2. Run the task by clicking **Run This Task**.

3 Review Results. The possible results are:

Pass: Move on to the next task.
Warning: Review results, decide whether to move on or fix.
Fail: Review results, do not move on without fixing.

4 If Status is **Warning** or **Fail**, you can:

- Manually fix the problem using the **Explore Result** button, if present.
- Manually fix the problem by modifying the model.
- Automatically fix the problem using the **Modify All** button, if available.

5 Once you have fixed a **Warning** or **Failed** task, rerun the task by clicking **Run This Task**.

Run to Selected Task. If you know that a particular task causes a failure, you might want to run all the tasks prior to this task and save a restore point before continuing the run. For more information about restore points, see "Saving a Restore Point" on page 5-11. To run all tasks up to and including the currently selected task:

1 Select the last task that you want to run.

2 Right click this task to open the context menu.

3 From the context menu, select `Run to Selected Task` to run all tasks up to and including the selected task.

Note If a task before the selected task fails, the Fixed-Point Advisor stops the run at the failed task.

Rerun a Task. You might want to rerun a task to see if changes you make result in a different answer. To rerun a task that you have run before:

1 Select the task that you want to rerun.

2 Specify input parameters, if present.

3 Run the task by clicking **Run This Task**.

The task reruns.

Caution All downstream tasks are reset to **Not Run** if:

- The task fails.
- You click the **Modify All** button.

View a Run Summary. To view a complete run summary of **Pass**, **Failed**, **Warning**, and **Not Run** tasks:

1. Select the **Fixed-Point Advisor** folder.
2. Click the path link listed for Report. A report containing a summary of all tasks is displayed.

See Also

- "Best Practices for Using the Fixed-Point Advisor" on page 5-2
- "Working with the Fixed-Point Advisor" on page 5-2
- "Converting a Model from Floating- to Fixed-Point Using Simulation Data" on page 5-14

Prepare Model for Conversion

In this section...

Prepare Model for Conversion Overview

This folder contains tasks for configuring and setting up the model for data logging.

Description

Validate model-wide settings and create simulation reference data for downstream tasks.

See Also

- "Working with the Fixed-Point Advisor" on page 5-2
- "Converting a Model from Floating- to Fixed-Point Using Simulation Data" on page 5-14

Verify model simulation settings

Validate that model simulation settings allow signal logging and disable data type override to facilitate conversion to fixed point. Logged signals are used for analysis and comparison in later tasks.

Description

Ensures that fixed-point data can be logged in downstream tasks.

Results and Recommended Actions

Conditions	Recommended Action
The following Fixed-Point Tool setting is not set to the correct value: • **Data type override**	Set **Data type override** to `Use local settings`
The model Configuration Parameters **Data Import/Export** > **Signal logging** check box is `off`.	Set to `on`
The `fipref` `DataTypeOverride` property is not set to `Off`.	Set `DataTypeOverride` to `Off`

Action Results

Clicking **Modify All** configures the model for recommended simulation settings and `fipref` objects. A table displays the current and previous block settings.

See Also

- "Data Type Override" on page 9-58
- "Signal logging"
- "Using fipref Objects to Set Data Type Override Preferences"

Verify update diagram status

Verify update diagram succeeds.

Description

A model update diagram action is necessary for most down stream tasks.

Results and Recommended Actions

Conditions	Recommended Action
The model diagram does not update.	Fix the model. Make sure needed `mat` files are loaded.

See Also

"Updating a Block Diagram" in the Simulink documentation

Address unsupported blocks

Identify blocks that do not support fixed-point data types.

Description

Blocks that do not support fixed-point data types cannot be converted.

Results and Recommended Actions

Conditions	Recommended Action
Blocks that do not support fixed-point data types and cannot be converted exist in model.	• Replace the block with the block specified in the Result pane by right-clicking the block and selecting the replacement from the context menu. **Note** The Fixed-Point Advisor provides a preview of the replacement block. To view the replacement and verify its settings, click the Preview link. If the settings on the replacement block differ from the settings on the original block, set up the replacement block to match the original block. • Isolate the block by right-clicking the block and selecting **Insert Data Type Conversion > All Ports**.

Tips

- Before inserting a replacement block, use the Preview link to view the replacement block. If necessary, update the settings on the replacement block to match the settings on the original block.
- If the Fixed-Point Advisor does not recommend a corresponding fixed-point block, replace the unsupported block with a number of lower-level blocks to provide the same functionality.
- The goal is to replace all blocks that do not support fixed-point data types. Using Data Type Conversion blocks to isolate blocks at this stage enables you to continue running through the conversion process. However, this will cause the **Summarize data type** task to fail downstream. To fix this failure, you must replace the block that does not support fixed-point data types.

See Also

The Simulink Block Data Type Support table summarizes characteristics of blocks in the Simulink and Simulink Fixed Point block libraries, including whether or not they support fixed-point data types. To view this table, enter the following command at the MATLAB command line:

```
showblockdatatypetable
```

Set up signal logging

Specify at least one signal of interest to log during simulation. Logged signals are used for analysis and comparison in other tasks. Suggested signals to log are model inports and outports.

Description

The Fixed-Point Advisor uses logged signals to compare the initial data type to the fixed-point data type.

Analysis Result and Recommended Actions

Conditions	Recommended Action
No signals are logged.	If you are using simulation minimum and maximum values, specify at least one signal to be logged. Otherwise, ignore this warning.

Tips

Log inports and outports of the system under conversion.

Create simulation reference data

Simulate the model using the current solver settings, and create reference data to use for comparison and analysis. If necessary, you can stop the simulation by selecting the waitbar and then pressing Ctrl+C. To set **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`, click the **Modify All** button.

Description

Simulate the model using the current solver settings, create and archive reference signal data to use for comparison and analysis in downstream tasks.

Input Parameters

Merge instrumentation results from multiple simulations

Merges new simulation minimum and maximum results with existing simulation results in the active run. Allows you to collect complete range information from multiple test benches. Does not merge signal logging results.

Results and Recommended Actions

Conditions	Recommended Action
Simulation does not run.	Fix errors so simulation will run.
Fixed-point instrumentation mode is not set to `Minimums, maximums and overflows`	If you are using simulation minimum and maximum values, set **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`. Otherwise, ignore this warning.

Action Results

Clicking **Modify All** sets **Fixed-point instrumentation mode** to `Minimums, maximums and overflows`. A table displays the current and previous block settings.

Tips

- If the simulation is set up to have a long simulation time, after starting the run of this task you can stop the simulation by selecting the waitbar and then pressing **Ctrl+C**. This allows you to change the simulation time and continue without having to wait for the long simulation to complete.
- Specifying small simulation run times reduces task processing times. You can change the simulation run time in the Configuration Parameters dialog box. See “Start time” and “Stop time” in the Simulink reference for more information.

Verify Fixed-Point Conversion Guidelines Overview

Verify modeling guidelines related to fixed-point conversion goals.

Description

Validate model-wide settings.

See Also

- "Working with the Fixed-Point Advisor" on page 5-2
- "Converting a Model from Floating- to Fixed-Point Using Simulation Data" on page 5-14

Check model configuration data validity diagnostic parameters settings

Verify that model **Configuration Parameters** > **Diagnostic** > **Data Validity** parameters are not set to error.

Description

If the **Configuration Parameters** > **Diagnostic** > **Data Validity** parameters are set to error, the model update diagram action fails in downstream tasks.

Results and Recommended Actions

Conditions	Recommended Action
Detect downcast is set to `error`. **Detect overflow** is set to `error`. **Detect underflow** is set to `error`. **Detect precision loss** is set to `error`. **Detect loss of tunability** is set to `error`.	Set all **Configuration Parameters** > **Diagnostics** > **Data Validity** > **Parameters** options to `warning`.

Action Results

Clicking **Modify All** sets all **Configuration Parameters** > **Diagnostics** > **Data Validity** > **Parameters** options to `warning`. A table displays the current and previous settings.

Implement logic signals as Boolean data

Confirm that Simulink simulations are configured to treat logic signals as Boolean data.

Description

Configuring logic signals as Boolean data optimizes the code generated in downstream tasks.

Results and Recommended Actions

Conditions	Recommended Action
Implement logic signals as Boolean data is set to `off`.	Set **Configuration Parameters** > **Optimization** > **Implement logic signals as Boolean data** to on.

Action Results

Clicking **Modify All** selects the model **Configuration Parameters** > **Optimization** > **Implement logic signals as Boolean data** check box. A table displays the current and previous parameter settings.

Check for proper bus usage

Identify any Mux block used as a bus creator and any bus signal treated as a vector.

Description

This task identifies:

- Mux blocks that are bus creators
- Bus signals that the top-level model treats as vectors

Results and Recommended Actions

Conditions	Recommended Action
The Fixed-Point Advisor is not operating on a top-level model	If this task is important to your conversion, start the Fixed-Point Advisor on the top-level model.
The model is not configured to detect future changes that might result in improper bus usage.	Set **Configuration Parameters** > **Diagnostics** > **Connectivity** > **Buses** > **Bus signal treated as vector** to `error`.

Note This task is a Simulink task. For more information, see "Check for proper bus usage" in the Simulink documentation.

Simulation range checking

Verify that model **Configuration Parameters** > **Diagnostics** > **Simulation range checking** is not set to `none`.

Description

If **Configuration Parameters** > **Diagnostics** > **Simulation range checking** is set to `none`, the simulation does not generate any range checking warnings.

Results and Recommended Actions

Conditions	Recommended Action
Configuration Parameters > **Diagnostics** > **Simulation range checking** is set to `none`.	Set **Configuration Parameters** > **Diagnostics** > **Simulation range checking** to `warning`.

Action Results

Clicking **Modify All** sets **Configuration Parameters** > **Diagnostics** > **Simulation range checking** to `warning`.

Check for implicit signal resolution

Check if model uses implicit signal resolution.

Description

Models with implicit signal resolution attempt to resolve all named signals and states to Simulink signal objects, which is inefficient and slows incremental code generation and model reference. This task identifies those signals and states for which you may turn off implicit signal resolution and enforce resolution.

Results and Recommended Actions

Conditions	Recommended Action
Model uses implicit signal resolution.	• Set the model **Configuration Parameters** > **Diagnostics** > **Data Validity** > **Signal resolution** to `Explicit only`. • Enforce resolution for each of the signals and states in the model by selecting **Signal name must resolve to Simulink signal object**.

Action Results

Clicking **Modify All** sets the model **Configuration Parameters** > **Diagnostics** > **Data Validity** > **Signal resolution** to `Explicit only` and enforces resolution for each of the signals and states in the model. Tables display the current and previous settings.

See Also

"Resolving Signal Objects for Output Data" in the Simulink documentation

Prepare for Data Typing and Scaling

In this section...

Prepare for Data Typing and Scaling Overview

Configure blocks with data type inheritance or constraints to avoid data type propagation errors.

Description

The block settings from this folder simplifies the initial scaling. The optimal block configuration is achieved in later stages. The tasks in this folder are preparation for scaling, not actual scaling.

Tips

Block output and parameter minimum and maximum values can be specified in this step.

See Also

- "Working with the Fixed-Point Advisor" on page 5-2
- "Converting a Model from Floating- to Fixed-Point Using Simulation Data" on page 5-14

Review locked data type settings

Review blocks that currently have their scaling locked down and will be excluded from autoscaling.

Description

When blocks have their scaling locked, the Fixed-Point Advisor excludes them from autoscaling. This task identifies blocks that have locked scaling so that you can unlock them.

Results and Recommended Actions

Conditions	Recommended Action
Blocks have locked scaling.	Unlock scaling on blocks that currently have locked scaling.

Action Results

Clicking **Modify All** unlocks scaling on blocks that currently have locked scaling.

Remove output data type inheritance

Identify blocks with an inherited output signal data type.

Description

Inherited data types might lead to data type propagation errors.

For floating-point inheritance blocks with floating-point inputs or outputs, the Fixed-Point Advisor replaces the inheritance with the fixed-point data type specified by the user. For floating-point inheritance blocks with fixed-point output and other Simulink and DSP System Toolbox and Communications System Toolbox™ blocks, the Fixed-Point Advisor now detects inheritance and replaces it with the compiled data type.

What are Floating-Point Inheritance Blocks?

For floating-point inheritance blocks, when inputs are floating-point, all internal and output data types are floating point.

Note This task is preparation for scaling, not actual scaling.

Input Parameters

Data type for blocks with floating-point inheritance

Enter a default fixed-point data type to use for floating-point inheritance blocks, or select one from the list:

```
undefined
int8
uint8
int16
uint16
int32
uint32
fixdt(1,16,4)
```

Results and Recommended Actions

Conditions	Recommended Action
An input parameter is invalid.	Enter or select a valid value for the **Data type for blocks with floating-point inheritance** input parameter.
The system or subsystems contain floating-point inheritance blocks that have floating-point inputs.	Set the block output data type to the recommended data type. Remove floating-point inheritance for these blocks by explicitly configuring the **Output data type** or **Output data type mode** parameter to the recommended value where possible.
Blocks or Stateflow output data in the current system or subsystems have inherited output data types.	Remove output data type inheritance for blocks by explicitly configuring the **Output data type** or **Output data type mode** parameter to the recommended value where possible. Remove output data type inheritance for Logical Operator blocks by clearing the **Require all inputs and outputs to have the same data type parameter** parameter. Remove Stateflow output data type inheritance by explicitly configuring the output data **Type** property.

Action Results

Clicking **Modify All** explicitly configures the output data types to the recommended values where possible. Tables list the previous and current data types for the reconfigured blocks.

Relax input data type settings

Identify blocks with input data type constraints.

Description

Blocks that have input data type constraints might lead to data type propagation errors.

Note This task is preparation for scaling, not actual scaling.

Results and Recommended Actions

Conditions	Recommended Action
The input data types of blocks or Stateflow charts in the current system or subsystems have constraints.	Explicitly configure flexible input data types for blocks by setting the `InputSameDT` parameter to `off` where possible. Explicitly configure Logical Operator blocks to have flexible input data types by setting the `AllPortsSameDT` parameter to `off`. Explicitly configure flexible Stateflow chart input data types by setting the `Type` method to `Inherited`. Select the **Use Strong Data Typing with Simulink I/O** chart property.

Action Results

Clicking **Modify All** explicitly configures the specified settings to the recommended value where possible. A table lists the previous and current settings for the reconfigured blocks.

Tip

Removing unnecessary data setting restrictions makes it more likely that the **Propose data types** task will succeed downstream.

Verify Stateflow charts have strong data typing with Simulink

Verify all Stateflow charts are configured to have strong data typing with Simulink I/O.

Description

Identify mismatches between input or output fixed-point data in Stateflow charts and their counterparts in Simulink models.

Note This task is preparation for scaling, not actual scaling.

Results and Recommended Actions

Conditions	Recommended Action
Stateflow charts do not have strong data typing with Simulink I/O.	Select the **Use Strong Data Typing with Simulink I/O** check box in the chart properties dialog.

Action Results

Clicking **Modify All** configures all Stateflow charts to have strong data typing with Simulink I/O.

Remove redundant specification between signal objects and blocks

Identify and remove redundant data type specification originating from blocks and Simulink signal objects.

Description

This task prepares your model for scaling by identifying and removing redundant data type specification originating from blocks and Simulink signal objects.

Note You must rerun this task whenever you delete or manipulate a Simulink signal object in the base workspace.

This task is preparation for scaling.

Input Parameters

Remove redundant specification from

Select from the list:

`Blocks`
Identify and remove redundant data type specification from blocks.

`Signal objects`
Identify and remove redundant data type specification from Simulink signal objects.

Results and Recommended Actions

Conditions	Recommended Action
Blocks associated with Simulink signal objects do not have their data type specification set to a passive mode.	Set the data type specification of these blocks to a passive mode, such as `Inherit via back propagation`.
Simulink signal objects associated with blocks do not have their data type specification set to a passive mode.	Set the data type specification of these Simulink signal objects to `Auto`.

Action Results

Clicking **Modify All** explicitly configures the properties of the blocks or Simulink signal objects to the recommended value where possible. A table displays the current and previous settings.

Verify hardware selection

Verify target hardware setting.

Description

Review the hardware device settings and verify they are the settings you intend to use.

Input Parameters

Default type of all floating-point signals

Enter a default fixed-point data type to use for all floating-point signals, or select one from the list. For FPGA/ASIC targets, specify the type explicitly.

`Remain floating-point`
Use this setting if you are converting only part of the model to fixed point and want to leave the rest of the model as floating point.

`Same as embedded hardware integer`
Use this setting if the hardware device specified is a microprocessor.

`int8`

`int16`

`int32`

`fixdt(1,16,4)`

Results and Recommended Actions

Conditions	Recommended Action
The model's **Configuration Parameters** > **Hardware Implementation** device parameters are not specified.	Provide values for the **Configuration Parameters** > **Hardware Implementation** > **Device vendor** and **Device type** parameters.
Default data type of all floating-point signals is set to `Remain floating-point`	For microprocessors, set to `Same as embedded hardware integer`. For FPGA/ASIC, set the data type explicitly. The Fixed-Point Advisor uses the sign and word length of this data type.

See Also

- "Device type"
- "Device vendor"

Specify block minimum and maximum values

Specify block output and parameter minimum and maximum values.

Description

Block output and parameter minimum and maximum values are used for fixed-point scaling in other tasks. Typically, they are determined during the design process based on the system you are creating.

Note This task is preparation for scaling, not actual scaling.

Results and Recommended Actions

Conditions	Recommended Action
Minimum and maximum values are not specified for Inport blocks.	Specify minimum and maximum values for Inport blocks.
Warning if no simulation minimum or maximum for any signals.	If you are using simulation minimum and maximum data, return to "Create simulation reference data" to set up signal logging.

Tips

- In this task, you can specify minimum and maximum values for any block.
- You can promote simulation minimum and maximum values to output minimum and maximum values using the Model Advisor Result Explorer, launched by clicking the **Explore Result** button. In the center pane of the Model Advisor Result Explorer, use the check boxes in the **PromoteSimMinMax** column to promote values.
- If you do not specify block minimum and maximum values, the **Propose data types** task might fail later in the conversion.

See Also

"Batch-Fixing Warnings or Failures" in the Simulink documentation.

Return to the Fixed-Point Tool to Perform Data Typing and Scaling

Close the Fixed-Point Advisor and return to the Fixed-Point Tool to autoscale your model.

See Also

- "Working with the Fixed-Point Advisor" on page 5-2
- "Converting a Model from Floating- to Fixed-Point Using Simulation Data" on page 5-14

A

Writing Fixed-Point S-Functions

This appendix discusses the API for user-written fixed-point S-functions, which enables you to write Simulink C S-functions that directly handle fixed-point data types. Note that the API also provides support for standard floating-point and integer data types. You can find the files and demos associated with this API in the following locations:

- *matlabroot*/simulink/include/
- *matlabroot*/toolbox/simulink/fixedandfloat/fxpdemos/

Data Type Support

In this section...

Supported Data Types

The API for user-written fixed-point S-functions provides support for a variety of Simulink and Simulink Fixed Point data types, including

- Built-in Simulink data types
  - `single`
  - `double`
  - `uint8`
  - `int8`
  - `uint16`
  - `int16`
  - `uint32`
  - `int32`
- Fixed-point Simulink data types, such as
  - `sfix16_En15`
  - `ufix32_En16`
  - `ufix128`
  - `sfix37_S3_B5`
- Data types resulting from a data type override with `Scaled double`, such as
  - `flts16`
  - `flts16_En15`

- `fltu32_S3_B5`

For more information, see "Fixed-Point Data Type and Scaling Notation" on page 2-15.

The Treatment of Integers

The API treats integers as fixed-point numbers with trivial scaling. In [Slope Bias] representation, fixed-point numbers are represented as

$$real\text{-}world\ value = (slope \times integer) + bias.$$

In the trivial case, $slope = 1$ and $bias = 0$.

In terms of binary-point-only scaling, the binary point is to the right of the least significant bit for trivial scaling, meaning that the fraction length is zero:

$$real\text{-}world\ value = integer \times 2^{-fraction\ length} = integer \times 2^{0}.$$

In either case, trivial scaling means that the real-world value is equal to the stored integer value:

$$real\text{-}world\ value = integer.$$

All integers, including Simulink built-in integers such as `uint8`, are treated as fixed-point numbers with trivial scaling by this API. However, Simulink built-in integers are different in that their use does not cause a Simulink Fixed Point software license to be checked out.

Data Type Override

The Fixed-Point Tool enables you to perform various data type overrides on fixed-point signals in your simulations. This API can handle signals whose data types have been overridden in this way:

- A signal that has been overridden with `Single` is treated as a Simulink built-in `single`.
- A signal that has been overridden with `Double` is treated as a Simulink built-in `double`.

- A signal that has been overridden with `Scaled double` is treated as being of data type `ScaledDouble`.

`ScaledDouble` signals are a hybrid between floating-point and fixed-point signals, in that they are stored as `doubles` with the scaling, sign, and word length information retained. The value is stored as a floating-point `double`, but as with a fixed-point number, the distinction between the stored integer value and the real-world value remains. The scaling information is applied to the stored integer `double` to obtain the real-world value. By storing the value in a `double`, overflow and precision issues are almost always eliminated. Refer to any individual API function reference page at the end of this appendix to learn how that function treats `ScaledDouble` signals.

For more information about the Fixed-Point Tool and data type override, refer to Chapter 6, "Fixed-Point Tool" and the `fxptdlg` reference page in the Simulink documentation.

Structure of the S-Function

The following diagram shows the basic structure of an S-function that directly handles fixed-point data types.

Include fixedpoint.h after simstruc.h

Include fixedpoint.c after simulink.c

```
/* Copyright 1994-2006 The MathWorks, Inc.
 * $Revision: $
 * $Date: $
 *
 * File:   sfun_user_fxp_bare.c
 *
 * Abstract:
 *      Bare S-function that supports fixed-point.
 */

/*=========================================*
 * Required setup for C MEX S-function   *
 *=========================================*/
#define S_FUNCTION_NAME sfun_user_fxp_bare
#define S_FUNCTION_LEVEL 2

#include <math.h>
#include "simstruc.h"
#include "fixedpoint.h"

static void mdlInitializeSizes(SimStruct *S)
{
}

static void mdlInitializeSampleTimes(SimStruct *S)
{
}

static void mdlOutputs(SimStruct *S, int_T tid)
{
}

static void mdlTerminate(SimStruct *S)
{
}

/*=========================================*
 * Required closing for C MEX S-function *
 *=========================================*/
#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#include "fixedpoint.c"
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

The callouts in the diagram alert you to the fact that you must include `fixedpoint.h` and `fixedpoint.c` at the appropriate places in the S-function. The other elements of the S-function displayed in the diagram follow the

standard requirements for S-functions. If you need more information on this topic, refer to *Developing S-Functions* in the Simulink documentation.

To learn how to create a MEX-file for your user-written fixed-point S-function, see "Creating MEX-Files" on page A-23.

Storage Containers

In this section...

Introduction

While coding with the API for user-written fixed-point S-functions, it is important to keep in mind the difference between storage container size, storage container word length, and signal word length. The sections that follow discuss the containers used by the API to store signals in simulation and code generation.

Storage Containers in Simulation

In simulation, signals are stored in one of several types of containers of a specific size.

Storage Container Categories

During simulation, fixed-point signals are held in one of the types of storage containers, as shown in the following table. In many cases, signals are represented in containers with more bits than their specified word length.

Fixed-Point Storage Containers

Container Category	Signal Word Length	Container Word Length	Container Size
`FXP_STORAGE_INT8` (signed) `FXP_STORAGE_UINT8` (unsigned)	1 to 8 bits	8 bits	1 byte
`FXP_STORAGE_INT16` (signed) `FXP_STORAGE_UINT16` (unsigned)	9 to 16 bits	16 bits	2 bytes
`FXP_STORAGE_INT32` (signed) `FXP_STORAGE_UINT32` (unsigned)	17 to 32 bits	32 bits	4 bytes

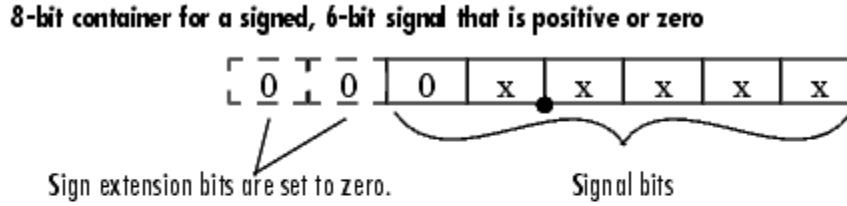
Fixed-Point Storage Containers (Continued)

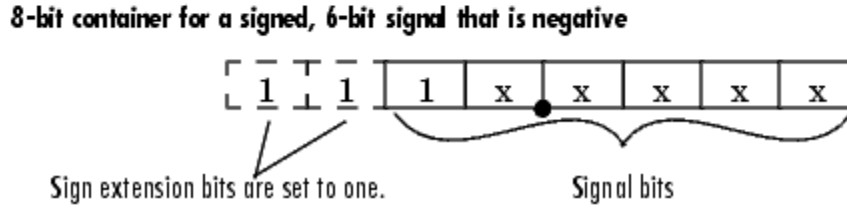
Container Category	Signal Word Length	Container Word Length	Container Size
`FXP_STORAGE_OTHER_SINGLE_WORD`	33 to word length of `long` data type	Length of `long` data type	Length of `long` data type
`FXP_STORAGE_MULTIWORD`	Greater than the word length of `long` data type to 128 bits	Multiples of length of `long` data type to 128 bits	Multiples of length of `long` data type to 128 bits

When the number of bits in the signal word length is less than the size of the container, the word length bits are always stored in the least significant bits of the container. The remaining container bits must be sign extended:

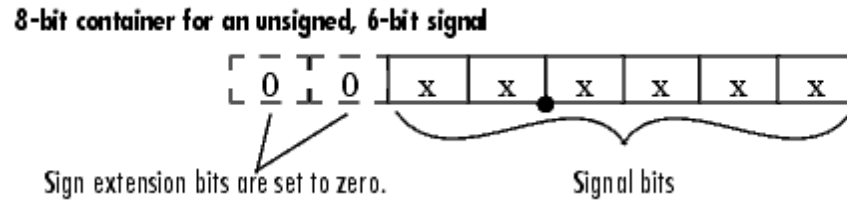
- If the data type is unsigned, the sign extension bits must be cleared to zero.
- If the data type is signed, the sign extension bits must be set to one for strictly negative numbers, and cleared to zero otherwise.

For example, a signal of data type `sfix6_En4` is held in a `FXP_STORAGE_INT8` container. The signal is held in the six least significant bits. The remaining two bits are set to zero when the signal is positive or zero, and to one when it is negative.





A signal of data type `ufix6_En4` is held in a `FXP_STORAGE_UINT8` container. The signal is held in the six least significant bits. The remaining two bits are always cleared to zero.



The signal and storage container word lengths are returned by the `ssGetDataTypeFxpWordLength` and `ssGetDataTypeFxpContainWordLen` functions, respectively. The storage container size is returned by the `ssGetDataTypeStorageContainerSize` function. The container category is returned by the `ssGetDataTypeStorageContainCat` function, which in addition to those in the table above, can also return the following values.

Other Storage Containers

Container Category	Description
FXP_STORAGE_UNKNOWN	Returned if the storage container category is unknown
FXP_STORAGE_SINGLE	The container type for a Simulink single
FXP_STORAGE_DOUBLE	The container type for a Simulink double
FXP_STORAGE_SCALEDDOUBLE	The container type for a data type that has been overridden with Scaled double

Storage Containers in Simulation Example

An sfix24_En10 data type has a word length of 24, but is actually stored in 32 bits during simulation. For this signal,

- ssGetDataTypeStorageContainCat returns FXP_STORAGE_INT32.
- ssGetDataTypeStorageContainerSize or sizeof( ) returns 4, which is the storage container size in bytes.
- ssGetDataTypeFxpContainWordLen returns 32, which is the storage container word length in bits.
- ssGetDataTypeFxpWordLength returns 24, which is the data type word length in bits.

Storage Containers in Code Generation

The storage containers used by this API for code generation are not always the same as those used for simulation. During code generation, a native C data type is always used. Floating-point data types are held in C double or float. Fixed-point data types are held in C signed and unsigned char, short, int, or long.

Emulation

Because it is valuable for rapid prototyping and hardware-in-the-loop testing, the emulation of smaller signals inside larger containers is supported in code generation. For example, a 29-bit signal is supported in code generation if there is a C data type available that has at least 32 bits. The rules for

placing a smaller signal into a larger container, and for dealing with the extra container bits, are the same in code generation as for simulation.

If a smaller signal is emulated inside a larger storage container in simulation, it is not necessarily emulated in code generation. For example, a 24-bit signal is emulated in a 32-bit storage container in simulation. However, some DSP chips have native support for 24-bit quantities. On such a target, the C compiler can define an `int` or a `long` to be exactly 24 bits. In this case, the 24-bit signal is held in a 32-bit container in simulation, and in a 24-bit container in code generation.

Conversely, a signal that was not emulated in simulation might need to be emulated in code generation. For example, some DSP chips have minimal support for integers. On such chips, `char`, `short`, `int`, and `long` might all be defined to 32 bits. In that case, it is necessary to emulate 8- and 16-bit fixed-point data types in code generation.

Storage Container TLC Functions

Since the mapping of storage containers in simulation to storage containers in code generation is not one-to-one, the Target Language Compiler (TLC) functions for storage containers are different from those in simulation:

- `FixPt_DataTypeNativeType`
- `FixPt_DataTypeStorageDouble`
- `FixPt_DataTypeStorageSingle`
- `FixPt_DataTypeStorageScaledDouble`
- `FixPt_DataTypeStorageSInt`
- `FixPt_DataTypeStorageUInt`
- `FixPt_DataTypeStorageSLong`
- `FixPt_DataTypeStorageULong`
- `FixPt_DataTypeStorageSShort`
- `FixPt_DataTypeStorageUShort`
- `FixPt_DataTypeStorageMultiword`

The first of these TLC functions, `FixPt_DataTypeNativeType`, is the closest analogue to `ssGetDataTypeStorageContainCat` in simulation. `FixPt_DataTypeNativeType` returns a TLC string that specifies the type of the storage container, and the Simulink Coder product automatically inserts a `typedef` that maps the string to a native C data type in the generated code.

For example, consider a fixed-data type that is held in `FXP_STORAGE_INT8` in simulation. `FixPt_DataTypeNativeType` will return `int8_T`. The `int8_T` will be `typdef`'d to a `char`, `short`, `int`, or `long` in the generated code, depending upon what is appropriate for the target compiler.

The remaining TLC functions listed above return `TRUE` or `FALSE` depending on whether a particular standard C data type is used to hold a given API-registered data type. Note that these functions do not necessarily give mutually exclusive answers for a given registered data type, due to the fact that C data types can potentially overlap in size. In C,

$$\text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}.$$

One or more of these C data types can be, and very often are, the same size.

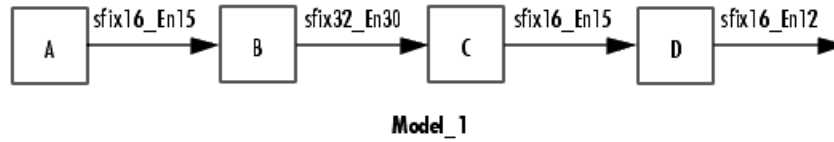
Data Type IDs

In this section...

The Assignment of Data Type IDs

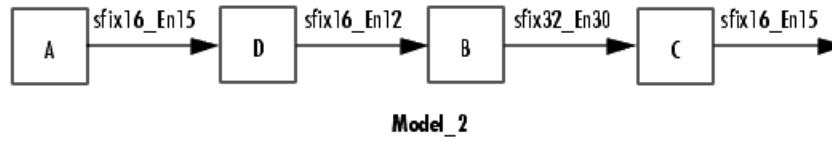
Each data type used in your S-function is assigned a data type ID. You should always use data type IDs to get and set information about data types in your S-function.

In general, the Simulink software assigns data type IDs during model initialization on a "first come, first served" basis. For example, consider the generalized schema of a block diagram below.



The Simulink software assigns a data type ID for each output data type in the diagram in the order it is requested. For simplicity, assume that the order of request occurs from left to right. Therefore, the output of block A may be assigned data type ID 13, and the output of block B may be assigned data type ID 14. The output data type of block C is the same as that of block A, so the data type ID assigned to the output of block C is also 13. The output of block D is assigned data type ID 15.

Now if the blocks in the model are rearranged,



The Simulink software still assigns the data type IDs in the order in which they are used. Therefore each data type might end up with a different data type ID. The output of block A is still assigned data type ID 13. The output of block D is now next in line and is assigned data type ID 14. The output of block B is assigned data type ID 15. The output data type of block C is still the same as that of block A, so it is also assigned data type ID 13.

This table summarizes the two cases described above.

Block	Data Type ID in Model_1	Data Type ID in Model_2
A	13	13
B	14	15
C	13	13
D	15	14

This example illustrates that there is no strict relationship between the attributes of a data type and the value of its data type ID. In other words, the data type ID is not assigned based on the characteristics of the data type it is representing, but rather on when that data type is first needed.

Note Because of the nature of the assignment of data type IDs, you should always use API functions to extract information from a data type ID about a data type in your S-function.

Registering Data Types

The functions in the following table are available in the API for user-written fixed-point S-functions for registering data types in simulation. Each of these

functions will return a data type ID. To see an example of a function being used, go to the file and line indicated in the table.

Data Type Registration Functions

Function	Description	Example of Use
`ssRegisterDataTypeFxpBinaryPoint`	Register a fixed-point data type with binary-point-only scaling and return its data type ID	`sfun_user_fxp_asr.c` Line 252
`ssRegisterDataTypeFxpFSlopeFixExpBias`	Register a fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias, and return its data type ID	Not Available
`ssRegisterDataTypeFxpScaledDouble`	Register a scaled double data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias, and return its data type ID	Not Available
`ssRegisterDataTypeFxpSlopeBias`	Register a data type with [Slope Bias] scaling and return its data type ID	`sfun_user_fxp_dtprop.c` Line 319

Preassigned Data Type IDs

The Simulink software registers its built-in data types, and those data types always have preassigned data type IDs. The built-in data type IDs are given by the following tokens:

- `SS_DOUBLE`

- SS_SINGLE
- SS_INT8
- SS_UINT8
- SS_INT16
- SS_UINT16
- SS_INT32
- SS_UINT32
- SS_BOOLEAN

You do not need to register these data types. If you attempt to register a built-in data type, the registration function simply returns the preassigned data type ID.

Setting and Getting Data Types

Data type IDs are used to specify the data types of input and output ports, run-time parameters, and DWork states. To set fixed-point data types for quantities in your S-function, the procedure is as follows:

1. Register a data type using one of the functions listed in the table Data Type Registration Functions on page A-15. A data type ID is returned to you.

   Alternately, you can use one of the preassigned data type IDs of the Simulink built-in data types.

2. Use the data type ID to set the data type for an input or output port, run-time parameter, or DWork state using one of the following functions:
   - ssSetInputPortDataType
   - ssSetOutputPortDataType
   - ssSetRunTimeParamInfo
   - ssSetDWorkDataType

To get the data type ID of an input or output port, run-time parameter, or DWork state, use one of the following functions:

- ssGetInputPortDataType
- ssGetOutputPortDataType
- ssGetRunTimeParamInfo
- ssGetDWorkDataType

Getting Information About Data Types

You can use data type IDs with functions to get information about the built-in and registered data types in your S-function. The functions in the following tables are available in the API for extracting information about registered data types. To see an example of a function being used, go to the file and line indicated in the table. Note that data type IDs can also be used with all the standard data type access methods in simstruc.h, such as ssGetDataTypeSize.

Storage Container Information Functions

Function	Description	Example of Use
ssGetDataTypeFxpContainWordLen	Return the word length of the storage container of a registered data type	sfun_user_fxp_ ContainWordLenProbe.c Line 181
ssGetDataTypeStorageContainCat	Return the storage container category of a registered data type	sfun_user_fxp_asr.c Line 294
ssGetDataTypeStorageContainerSize	Return the storage container size of a registered data type	sfun_user_fxp_ StorageContainSizeProbe.c Line 171

Signal Data Type Information Functions

Function	Description	Example of Use
ssGetDataTypeFxpIsSigned	Determine whether a fixed-point registered data type is signed or unsigned	sfun_user_fxp_asr.c Line 254
ssGetDataTypeFxpWordLength	Return the word length of a fixed-point registered data type	sfun_user_fxp_asr.c Line 255
ssGetDataTypeIsFixedPoint	Determine whether a registered data type is a fixed-point data type	sfun_user_fxp_const.c Line 127
ssGetDataTypeIsFloatingPoint	Determine whether a registered data type is a floating-point data type	sfun_user_fxp_ IsFloatingPointProbe.c Line 176
ssGetDataTypeIsFxpFltApiCompat	Determine whether a registered data type is supported by the API for user-written fixed-point S-functions	sfun_user_fxp_asr.c Line 184
ssGetDataTypeIsScalingPow2	Determine whether a registered data type has power-of-two scaling	sfun_user_fxp_asr.c Line 203
ssGetDataTypeIsScalingTrivial	Determine whether the scaling of a registered data type is slope = 1, bias = 0	sfun_user_fxp_ IsScalingTrivialProbe.c Line 171

Signal Scaling Information Functions

Function	Description	Example of Use
ssGetDataTypeBias	Return the bias of a registered data type	sfun_user_fxp_dtprop.c Line 243
ssGetDataTypeFixedExponent	Return the exponent of the slope of a registered data type	sfun_user_fxp_dtprop.c Line 237

Signal Scaling Information Functions (Continued)

Function	Description	Example of Use
`ssGetDataTypeFracSlope`	Return the fractional slope of a registered data type	`sfun_user_fxp_dtprop.c` Line 234
`ssGetDataTypeFractionLength`	Return the fraction length of a registered data type with power-of-two scaling	`sfun_user_fxp_asr.c` Line 256
`ssGetDataTypeTotalSlope`	Return the total slope of the scaling of a registered data type	`sfun_user_fxp_dtprop.c` Line 240

Converting Data Types

The functions in the following table allow you to convert values between registered data types in your fixed-point S-function.

Data Type Conversion Functions

Function	Description	Example of Use
`ssFxpConvert`	Convert a value from one data type to another data type.	Not Available
`ssFxpConvertFromRealWorldValue`	Convert a value of data type `double` to another data type.	Not Available
`ssFxpConvertToRealWorldValue`	Convert a value of any data type to a `double`.	Not Available

Overflow Handling and Rounding Methods

In this section...

Tokens for Overflow Handling and Rounding Methods

The API for user-written fixed-point S-functions provides functions for some mathematical operations, such as conversions. When these operations are performed, a loss of precision or overflow may occur. The tokens in the following tables allow you to control the way an API function handles precision loss and overflow. The data type of the overflow handling methods is `fxpModeOverflow`. The data type of the rounding modes is `fxpModeRounding`.

Overflow Handling Tokens

Token	Description	Example of Use
`FXP_OVERFLOW_SATURATE`	Saturate overflows	Not Available
`FXP_OVERFLOW_WRAP`	Wrap overflows	Not Available

Rounding Method Tokens

Token	Description	Example of Use
`FXP_ROUND_CEIL`	Round to the closest representable number in the direction of positive infinity	Not Available
`FXP_ROUND_CONVERGENT`	Round toward nearest integer with ties rounding to nearest even integer	Not Available
`FXP_ROUND_FLOOR`	Round to the closest representable number in the direction of negative infinity	Not Available

Rounding Method Tokens (Continued)

Token	Description	Example of Use
FXP_ROUND_NEAR	Round to the closest representable number, with the exact midpoint rounded in the direction of positive infinity	Not Available
FXP_ROUND_NEAR_ML	Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers	Not Available
FXP_ROUND_SIMPLEST	Automatically chooses between round toward floor and round toward zero to produce generated code that is as efficient as possible	Not Available
FXP_ROUND_ZERO	Round to the closest representable number in the direction of zero	Not Available

Overflow Logging Structure

Math functions of the API, such as `ssFxpConvert`, can encounter overflows when carrying out an operation. These functions provide a mechanism to log the occurrence of overflows and to report that log back to the caller.

You can use a fixed-point overflow logging structure in your S-function by defining a variable of data type `fxpOverflowLogs`. Some API functions, such as `ssFxpConvert`, accept a pointer to this structure as an argument. The function initializes the logging structure and maintains a count of each the following events that occur while the function is being performed:

- Overflows
- Saturations
- Divide-by-zeros

When a function that accepts a pointer to the logging structure is invoked, the function initializes the event counts of the structure to zero. The requested math operations are then carried out. Each time an event is detected, the appropriate event count is incremented by one.

The following fields contain the event-count information of the structure:

- `OverflowOccurred`
- `SaturationOccurred`
- `DivisionByZeroOccurred`

Creating MEX-Files

In this section...

Introduction

In addition to including `fixedpoint.c` and `fixedpoint.h` (see "Structure of the S-Function" on page A-5), the sections that follow describe additional steps you need to take to create a MEX-file for your user-written fixed-point S-function.

For general information on creating MEX-files, refer to Writing Wrapper S-Functions in the Simulink Coder documentation.

MEX-Files on UNIX

On UNIX® systems, to create a MEX-file for a user-written fixed-point S-function, you need to pass an extra argument, `-lfixedpoint`, to the `mex` command. For example,

```
mex('sfun_user_fxp_asr.c','-lfixedpoint')
```

MEX-Files on Windows

On Windows® systems, to create a MEX-file for a user-written fixed-point S-function, you need to pass an extra argument to the `mex` command that identifies the appropriate version of `libfixedpoint.lib`. The version of `libfixedpoint.lib` that you use depends on the compiler that you specify when `mex -setup` is run. The possible versions, which are installed with Simulink software, are as follows:

- `extern\lib\win32\lcc\libfixedpoint.lib`
- `extern\lib\win32\microsoft\libfixedpoint.lib`
- `extern\lib\win32\watcom\libfixedpoint.lib`

For example, if Version 6.0 of the Microsoft® Visual C++® software is the compiler that the `mex` command is configured to use,

```
mex('sfun_user_fxp_asr.c',[matlabroot,'\extern\lib\win32\...
microsoft\libfixedpoint.lib'])
```

If your C compiler is not listed, consider the following options:

- Use the LCC freeware compiler. LCC is normally installed when the MATLAB software is installed. When you run `mex -setup` on a Windows system, LCC should be listed as an option.
- Your compiler may provide tools for translating one of the `libfixedpoint.lib` files or `extern/include/libfixedpoint.def` to the required format.

Fixed-Point S-Function Examples

In this section...

List of Fixed-Point S-Function Examples

The following files in *matlabroot*/toolbox/simulink/fixedandfloat/fxpdemos/ are examples of S-functions written with the API for user-written fixed-point S-functions:

- sfun_user_fxp_asr.c
- sfun_user_fxp_BiasProbe.c
- sfun_user_fxp_const.c
- sfun_user_fxp_ContainWordLenProbe.c
- sfun_user_fxp_dtprop.c
- sfun_user_fxp_FixedExponentProbe.c
- sfun_user_fxp_FracLengthProbe.c
- sfun_user_fxp_FracSlopeProbe.c
- sfun_user_fxp_IsFixedPointProbe.c
- sfun_user_fxp_IsFloatingPointProbe.c
- sfun_user_fxp_IsFxpFltApiCompatProbe.c
- sfun_user_fxp_IsScalingPow2Probe.c
- sfun_user_fxp_IsScalingTrivialProbe.c

- sfun_user_fxp_IsSignedProbe.c
- sfun_user_fxp_prodsum.c
- sfun_user_fxp_StorageContainCatProbe.c
- sfun_user_fxp_StorageContainSizeProbe.c
- sfun_user_fxp_TotalSlopeProbe.c
- sfun_user_fxp_U32BitRegion.c
- sfun_user_fxp_WordLengthProbe.c

The sections that follow present smaller portions of code that focus on specific kinds of tasks you might want to perform within your S-function.

Getting the Input Port Data Type

Within your S-function, you might need to know the data types of different ports, run-time parameters, and DWorks. In each case, you will need to get the data type ID of the data type, and then use functions from this API to extract information about the data type.

For example, suppose you need to know the data type of your input port. To do this,

1. Use ssGetInputPortDataType. The data type ID of the input port is returned.
2. Use API functions to extract information about the data type.

The following lines of example code are from sfun_user_fxp_dtprop.c.

In lines 191 and 192, ssGetInputPortDataType is used to get the data type ID for the two input ports of the S-function:

```
dataTypeIdU0 = ssGetInputPortDataType( S, 0 );
dataTypeIdU1 = ssGetInputPortDataType( S, 1 );
```

Further on in the file, the data type IDs are used with API functions to get information about the input port data types. In lines 205 through 226, a check is made to see whether the input port data types are single or double:

```
storageContainerU0 = ssGetDataTypeStorageContainCat( S,
     dataTypeIdU0 );
storageContainerU1 = ssGetDataTypeStorageContainCat( S,
     dataTypeIdU1 );

    if ( storageContainerU0 == FXP_STORAGE_DOUBLE ||
         storageContainerU1 == FXP_STORAGE_DOUBLE )
    {
        /* Doubles take priority over all other rules.
         * If either of first two inputs is double,
         * then third input is set to double.
         */
        dataTypeIdU2Desired = SS_DOUBLE;
    }
    else if ( storageContainerU0 == FXP_STORAGE_SINGLE ||
              storageContainerU1 == FXP_STORAGE_SINGLE )
    {
        /* Singles take priority over all other rules,
         * except doubles.
         * If either of first two inputs is single
         * then third input is set to single.
         */
        dataTypeIdU2Desired = SS_SINGLE;
    }
    else
```

In lines 227 through 244, additional API functions are used to get information about the data types if they are neither `single` nor `double`:

```
{
    isSignedU0 = ssGetDataTypeFxpIsSigned( S, dataTypeIdU0 );
    isSignedU1 = ssGetDataTypeFxpIsSigned( S, dataTypeIdU1 );

    wordLengthU0 = ssGetDataTypeFxpWordLength( S, dataTypeIdU0 );
    wordLengthU1 = ssGetDataTypeFxpWordLength( S, dataTypeIdU1 );

    fracSlopeU0 = ssGetDataTypeFracSlope( S, dataTypeIdU0 );
    fracSlopeU1 = ssGetDataTypeFracSlope( S, dataTypeIdU1 );

    fixedExponentU0 = ssGetDataTypeFixedExponent( S,dataTypeIdU0 );
```

```
    fixedExponentU1 = ssGetDataTypeFixedExponent( S,dataTypeIdU1 );

    totalSlopeU0 = ssGetDataTypeTotalSlope( S, dataTypeIdU0 );
    totalSlopeU1 = ssGetDataTypeTotalSlope( S, dataTypeIdU1 );

    biasU0 = ssGetDataTypeBias( S, dataTypeIdU0 );
    biasU1 = ssGetDataTypeBias( S, dataTypeIdU1 );
}
```

The functions used above return whether the data types are signed or unsigned, as well as their word lengths, fractional slopes, exponents, total slopes, and biases. Together, these quantities give full information about the fixed-point data types of the input ports.

Setting the Output Port Data Type

You may want to set the data type of various ports, run-time parameters, or DWorks in your S-function.

For example, suppose you want to set the output port data type of your S-function. To do this,

1. Register a data type by using one of the functions listed in the table Data Type Registration Functions on page A-15. A data type ID is returned.

   Alternately, you can use one of the predefined data type IDs of the Simulink built-in data types.

2. Use `ssSetOutputPortDataType` with the data type ID from Step 1 to set the output port to the desired data type.

In the example below from lines 336 - 352 of `sfun_user_fxp_const.c`, `ssRegisterDataTypeFxpBinaryPoint` is used to register the data type. `ssSetOutputPortDataType` then sets the output data type either to the given data type ID, or to be dynamically typed:

```
/* Register data type
     */
    if ( notSizesOnlyCall )
    {
        DTypeId DataTypeId = ssRegisterDataTypeFxpBinaryPoint(
```

```
            S,
            V_ISSIGNED,
            V_WORDLENGTH,
            V_FRACTIONLENGTH,
    1 /* true means obey data type override setting for
      this subsystem */ );

            ssSetOutputPortDataType( S, 0, DataTypeId );
     }
     else
     {
         ssSetOutputPortDataType( S, 0, DYNAMICALLY_TYPED );
     }
```

Interpreting an Input Value

Suppose you need to get the value of the signal on your input port to use in your S-function. You should write your code so that the pointer to the input value is properly typed, so that the values read from the input port are interpreted correctly. To do this, you can use these steps, which are shown in the example code below:

1. Create a void pointer to the value of the input signal.
2. Get the data type ID of the input port using `ssGetInputPortDataType`.
3. Use the data type ID to get the storage container type of the input.
4. Have a case for each input storage container type you want to handle. Within each case, you will need to perform the following in some way:
   - Create a pointer of the correct type according to the storage container, and cast the original void pointer into the new fully typed pointer (see **a** and **c**).
   - You can now store and use the value by dereferencing the new, fully typed pointer (see **b** and **d**).

For example,

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
```

```
const void *pVoidIn =
 (const void *)ssGetInputPortSignal( S, 0 ); (1)

DTypeId dataTypeIdU0 = ssGetInputPortDataType( S, 0 ); (2)

fxpStorageContainerCategory storageContainerU0 =
 ssGetDataTypeStorageContainCat( S, dataTypeIdU0 ); (3)

switch ( storageContainerU0 )
{
  case FXP_STORAGE_UINT8: (4)
    {
        const uint8_T *pU8_Properly_Typed_Pointer_To_U0; (a)

        uint8_T u8_Stored_Integer_U0; (b)

        pU8_Properly_Typed_Pointer_To_U0 =
 (const uint8_T  *)pVoidIn; (c)

        u8_Stored_Integer_U0 =
 *pU8_Properly_Typed_Pointer_To_U0; (d)

        <snip: code that uses input when it's in a uint8_T>
    }
    break;

  case FXP_STORAGE_INT8: (4)
    {
        const int8_T *pS8_Properly_Typed_Pointer_To_U0; (a)

        int8_T s8_Stored_Integer_U0; (b)

        pS8_Properly_Typed_Pointer_To_U0 =
 (const int8_T  *)pVoidIn; (c)

        s8_Stored_Integer_U0 =
 *pS8_Properly_Typed_Pointer_To_U0; (d)

        <snip: code that uses input when it's in a int8_T>
    }
```

```
        break;
```

Writing an Output Value

Suppose you need to write the value of the output signal to the output port in your S-function. You should write your code so that the pointer to the output value is properly typed. To do this, you can use these steps, which are followed in the example code below:

1. Create a void pointer to the value of the output signal.
2. Get the data type ID of the output port using `ssGetOutputPortDataType`.
3. Use the data type ID to get the storage container type of the output.
4. Have a case for each output storage container type you want to handle. Within each case, you will need to perform the following in some way:
   - Create a pointer of the correct type according to the storage container, and cast the original void pointer into the new fully typed pointer (see **a** and **c**).
   - You can now write the value by dereferencing the new, fully typed pointer (see **b** and **d**).

For example,

```
static void mdlOutputs(SimStruct *S, int_T tid)
{
    <snip>

    void *pVoidOut = ssGetOutputPortSignal( S, 0 ); (1)

    DTypeId dataTypeIdY0 = ssGetOutputPortDataType( S, 0 ); (2)

    fxpStorageContainerCategory storageContainerY0 =
     ssGetDataTypeStorageContainCat( S,
     dataTypeIdY0 ); (3)

    switch ( storageContainerY0 )
    {
      case FXP_STORAGE_UINT8: (4)
```

```
    {
        const uint8_T *pU8_Properly_Typed_Pointer_To_Y0; (a)

        uint8_T u8_Stored_Integer_Y0; (b)

      <snip: code that puts the desired output stored integer
value in to temporary variable u8_Stored_Integer_Y0>

        pU8_Properly_Typed_Pointer_To_Y0 =
 (const uint8_T  *)pVoidOut; (c)

        *pU8_Properly_Typed_Pointer_To_Y0 =
 u8_Stored_Integer_Y0; (d)

    }
    break;

  case FXP_STORAGE_INT8: (4)
    {
        const int8_T *pS8_Properly_Typed_Pointer_To_Y0; (a)

        int8_T s8_Stored_Integer_Y0; (b)

      <snip: code that puts the desired output stored integer
value in to temporary variable s8_Stored_Integer_Y0>

        pS8_Properly_Typed_Pointer_To_Y0 =
 (const int8_T  *)pVoidY0; (c)

        *pS8_Properly_Typed_Pointer_To_Y0 =
 s8_Stored_Integer_Y0; (d)

    }
    break;

<snip>
```

Using the Input Data Type to Determine the Output Data Type

The following sample code from lines 243 through 261 of `sfun_user_fxp_asr.c` gives an example of using the data type of the input to your S-function to calculate the output data type. Notice that in this code

- The output is signed or unsigned to match the input **(a)**.
- The output is the same word length as the input **(b)**.
- The fraction length of the output depends on the input fraction length and the number of shifts **(c)**.

```
#define MDL_SET_INPUT_PORT_DATA_TYPE
static void mdlSetInputPortDataType(SimStruct *S, int port,
     DTypeId dataTypeIdInput)
{
    if ( isDataTypeSupported( S, dataTypeIdInput ) )
    {
        DTypeId dataTypeIdOutput;

        ssSetInputPortDataType( S, port, dataTypeIdInput );

        dataTypeIdOutput = ssRegisterDataTypeFxpBinaryPoint(
     S,
   ssGetDataTypeFxpIsSigned( S, dataTypeIdInput ), (a)
   ssGetDataTypeFxpWordLength( S, dataTypeIdInput ), (b)
   ssGetDataTypeFractionLength( S, dataTypeIdInput )
     - V_NUM_BITS_TO_SHIFT_RGHT, (c)
            0 /* false means do NOT obey data type override
                setting for this subsystem */ );

        ssSetOutputPortDataType( S, 0, dataTypeIdOutput );
    }
}
```

API Function Reference

Purpose

Convert value from one data type to another

Syntax

```
extern void ssFxpConvert  (SimStruct *S,
                           void *pVoidDest,
                           size_t sizeofDest,
                           DTypeId dataTypeIdDest,
                           const void *pVoidSrc,
                           size_t sizeofSrc,
                           DTypeId dataTypeIdSrc,
                           fxpModeRounding roundMode,
                           fxpModeOverflow overflowMode,
                           fxpOverflowLogs *pFxpOverflowLogs)
```

Arguments

`S`
: SimStruct representing an S-function block.

`pVoidDest`
: Pointer to the converted value.

`sizeofDest`
: Size in memory of the converted value.

`dataTypeIdDest`
: Data type ID of the converted value.

`pVoidSrc`
: Pointer to the value you want to convert.

`sizeofSrc`
: Size in memory of the value you want to convert.

`dataTypeIdSrc`
: Data type ID of the value you want to convert.

`roundMode`
: Rounding mode you want to use if a loss of precision is necessary during the conversion. Possible values are `FXP_ROUND_CEIL`, `FXP_ROUND_CONVERGENT`, `FXP_ROUND_FLOOR`, `FXP_ROUND_NEAR`, `FXP_ROUND_NEAR_ML`, `FXP_ROUND_SIMPLEST` and `FXP_ROUND_ZERO`.

`overflowMode`
: Overflow mode you want to use if overflow occurs during the conversion. Possible values are `FXP_OVERFLOW_SATURATE` and `FXP_OVERFLOW_WRAP`.

`pFxpOverflowLogs`
: Pointer to the fixed-point overflow logging structure.

Description

This function converts a value of any registered built-in or fixed-point data type to any other registered built-in or fixed-point data type.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

None

See Also

`ssFxpConvertFromRealWorldValue`, `ssFxpConvertToRealWorldValue`

Purpose

Convert value of data type `double` to another data type

Syntax

```
extern void ssFxpConvertFromRealWorldValue
                                (SimStruct *S,
                                 void *pVoidDest,
                                 size_t sizeofDest,
                                 DTypeId dataTypeIdDest,
                                 double dblRealWorldValue,
                                 fxpModeRounding roundMode,
                                 fxpModeOverflow overflowMode,
                                 fxpOverflowLogs *pFxpOverflowLogs)
```

Arguments

`S`
: SimStruct representing an S-function block.

`pVoidDest`
: Pointer to the converted value.

`sizeofDest`
: Size in memory of the converted value.

`dataTypeIdDest`
: Data type ID of the converted value.

`dblRealWorldValue`
: Double value you want to convert.

`roundMode`
: Rounding mode you want to use if a loss of precision is necessary during the conversion. Possible values are `FXP_ROUND_CEIL`, `FXP_ROUND_CONVERGENT`, `FXP_ROUND_FLOOR`, `FXP_ROUND_NEAR`, `FXP_ROUND_NEAR_ML`, `FXP_ROUND_SIMPLEST` and `FXP_ROUND_ZERO`.

`overflowMode`
: Overflow mode you want to use if overflow occurs during the conversion. Possible values are `FXP_OVERFLOW_SATURATE` and `FXP_OVERFLOW_WRAP`.

`pFxpOverflowLogs`
: Pointer to the fixed-point overflow logging structure.

ssFxpConvertFromRealWorldValue

Description

This function converts a `double` value to any registered built-in or fixed-point data type.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

None

See Also

`ssFxpConvert`, `ssFxpConvertToRealWorldValue`

Purpose

Convert value of any data type to `double`

Syntax

```
extern double ssFxpConvertToRealWorldValue (SimStruct *S,
                                            const void *pVoidSrc,
                                            size_t sizeofSrc,
                                            DTypeId dataTypeIdSrc)
```

Arguments

`S`
: SimStruct representing an S-function block.

`pVoidSrc`
: Pointer to the value you want to convert.

`sizeofSrc`
: Size in memory of the value you want to convert.

`dataTypeIdSrc`
: Data type ID of the value you want to convert.

Description

This function converts a value of any registered built-in or fixed-point data type to a `double`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

None

See Also

`ssFxpConvert`, `ssFxpConvertFromRealWorldValue`

ssFxpGetU32BitRegion

Purpose

Return stored integer value for 32-bit region of real, scalar signal element

Syntax

```
extern uint32 ssFxpGetU32BitRegion(SimStruct *S,
                                   const void *pVoid
                                   DTypeId dataTypeId
                                   unsigned int regionIndex)
```

Arguments

`S`
: SimStruct representing an S-function block.

`pVoid`
: Pointer to the storage container of the real, scalar signal element in which the 32-bit region of interest resides.

`dataTypeId`
: Data type ID of the registered data type corresponding to the signal.

`regionIndex`
: Index of the 32-bit region whose stored integer value you want to retrieve, where 0 accesses the least significant 32-bit region.

Description

This function returns the stored integer value in the 32-bit region specified by `regionIndex`, associated with the fixed-point data type designated by `dataTypeId`. You can use this function with any fixed-point data type, including those with word sizes less than 32 bits. If the fixed-point word size is less than 32 bits, the remaining bits are sign extended.

This function generates an error if `dataTypeId` represents a floating-point data type.

To view a demo model whose S-functions use the `ssFxpGetU32BitRegion` function, at the MATLAB prompt, enter `fxpdemo_sfun_user_U32BitRegion`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages C

See Also `ssFxpSetU32BitRegion`

ssFxpGetU32BitRegionCompliant

Purpose

Determine whether S-function is compliant with the U32 bit region interface

Syntax

```
extern ssFxpSGetU32BitRegionCompliant(SimStruct *S,
                                      int *result)
```

Arguments

`S`

SimStruct representing an S-function block.

`result`

- `1` if S-function calls ssFxpSetU32BitRegionCompliant to declare compliance with memory footprint for fixed-point data types with 33 or more bits
- `0` if S-function does not call ssFxpSetU32BitRegionCompliant

Description

This function checks whether the S-function calls ssFxpSetU32BitRegionCompliant to declare compliance with the memory footprint for fixed-point data types with 33 or more bits. Before calling any other Simulink Fixed Point API function on data with 33 or more bits, you must call ssFxpSetU32BitRegionCompliant as follows:

```
ssFxpSetU32BitRegionCompliant(S,1);
```

Note The Simulink Fixed Point software assumes that S-functions that use fixed-point data types with 33 or more bits without calling `ssFxpSetU32BitRegionCompliant` are using the obsolete memory footprint that existed until R2007b. Either redesign these S-functions or isolate them using the library `fixpt_legacy_sfun_support.mdl`.

.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages C

See Also `ssFxpSetU32BitRegionCompliant`

ssFxpSetU32BitRegion

Purpose

Set stored integer value for 32-bit region of real, scalar signal element

Syntax

```
extern ssFxpSetU32BitRegion(SimStruct *S,
                            void *pVoid
                            DTypeId dataTypeId
                            uint32 regionValue
                            unsigned int regionIndex)
```

Arguments

`S`
: SimStruct representing an S-function block.

`pVoid`
: Pointer to the storage container of the real, scalar signal element in which the 32-bit region of interest resides.

`dataTypeId`
: Data type ID of the registered data type corresponding to the signal.

`regionValue`
: Stored integer value that you want to assign to a 32-bit region.

`regionIndex`
: Index of the 32-bit region whose stored integer value you want to set, where 0 accesses the least significant 32-bit region.

Description

This function sets `regionValue` as the stored integer value of the 32-bit region specified by `regionIndex`, associated with the fixed-point data type designated by `dataTypeId`. You can use this function with any fixed-point data type, including those with word sizes less than 32 bits. If the fixed-point word size is less than 32 bits, ensure that the remaining bits are sign extended.

This function generates an error if `dataTypeId` represents a floating-point data type, or if the stored integer value that you set is invalid.

To view a demo model whose S-functions use the `ssFxpSetU32BitRegion` function, at the MATLAB prompt, enter `fxpdemo_sfun_user_U32BitRegion`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

See Also

`ssFxpGetU32BitRegion`

ssFxpSetU32BitRegionCompliant

Purpose

Declare compliance with the U32 bit region interface for fixed-point data types with 33 or more bits

Syntax

```
extern ssFxpSetU32BitRegionCompliant(SimStruct *S,
                                     int Value)
```

Arguments

`S`

SimStruct representing an S-function block.

`Value`

- `1` declare compliance with memory footprint for fixed-point data types with 33 or more bits.

Description

This function declares compliance with the Simulink Fixed Point bit region interface for data types with 33 or more bits. The memory footprint for data types with 33 or more bits varies between MATLAB host platforms and might change between software releases. To make an S-function robust to memory footprint changes, use the U32 bit region interface. You can use identical source code on different MATLAB host platforms and with any software release from R2008b. If the memory footprint changes between releases, you do not have to recompile U32 bit region compliant S-functions.

To make an S-function U32 bit region compliant, before calling any other Simulink Fixed Point API function on data with 33 or more bits, you must call this function as follows:

```
ssFxpSetU32BitRegionCompliant(S,1);
```

If an S-function block contains a fixed-point data type with 33 or more bits, call this function in mdlInitializeSizes().

Note The Simulink Fixed Point software assumes that S-functions that use fixed-point data types with 33 or more bits without calling `ssFxpSetU32BitRegionCompliant` are using the obsolete memory footprint that existed until R2007b. Either redesign these S-functions or isolate them using the library `fixpt_legacy_sfun_support.mdl`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

See Also

`ssFxpGetU32BitRegionCompliant`

ssGetDataTypeBias

Purpose

Return bias of registered data type

Syntax

```
extern double ssGetDataTypeBias(SimStruct *S, DTypeId
                                dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know the bias.

Description

Fixed-point numbers can be represented as

real-world value = (*slope* × *integer*) + *bias*.

This function returns the bias of a registered data type:

- For both trivial scaling and power-of-two scaling, `0` is returned.
- If the registered data type is `ScaledDouble`, the bias returned is that of the nonoverridden data type.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeBias`

See Also

`ssGetDataTypeFixedExponent`, `ssGetDataTypeFracSlope`, `ssGetDataTypeTotalSlope`

Purpose

Return exponent of slope of registered data type

Syntax

```
extern int ssGetDataTypeFixedExponent (SimStruct *S, DTypeId
                                        dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know the exponent.

Description

Fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times integer) + bias,$$

where the slope can be expressed as

$$slope = fractional\ slope \times 2^{exponent}.$$

This function returns the exponent of a registered fixed-point data type:

- For power-of-two scaling, the exponent is the negative of the fraction length.
- If the data type has trivial scaling, including for data types `single` and `double`, the exponent is `0`.
- If the registered data type is `ScaledDouble`, the exponent returned is that of the nonoverridden data type.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeFixedExponent`

See Also

`ssGetDataTypeBias`, `ssGetDataTypeFracSlope`, `ssGetDataTypeTotalSlope`

Purpose

Return fractional slope of registered data type

Syntax

```
extern double ssGetDataTypeFracSlope(SimStruct *S, DTypeId
                                     dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know the fractional slope.

Description

Fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times integer) + bias,$$

where the slope can be expressed as

$$slope = fractional\ slope \times 2^{exponent}.$$

This function returns the fractional slope of a registered fixed-point data type. To get the total slope, use `ssGetDataTypeTotalSlope`:

- For power-of-two scaling, the fractional slope is `1`.
- If the data type has trivial scaling, including data types `single` and `double`, the fractional slope is `1`.
- If the registered data type is `ScaledDouble`, the fractional slope returned is that of the nonoverridden data type.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeFracSlope`

See Also

`ssGetDataTypeBias`, `ssGetDataTypeFixedExponent`, `ssGetDataTypeTotalSlope`

Purpose

Return fraction length of registered data type with power-of-two scaling

Syntax

```
extern int ssGetDataTypeFractionLength (SimStruct *S, DTypeId
                                        dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know the fraction length.

Description

This function returns the fraction length, or the number of bits to the right of the binary point, of the data type designated by `dataTypeId`.

This function errors out when `ssGetDataTypeIsScalingPow2` returns `FALSE`.

This function also errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeFractionLength`

See Also

`ssGetDataTypeFxpWordLength`

ssGetDataTypeFxpContainWordLen

Purpose

Return word length of storage container of registered data type

Syntax

```
extern int ssGetDataTypeFxpContainWordLen (SimStruct *S,
                                           DTypeId dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know the container word length.

Description

This function returns the word length, in bits, of the storage container of the fixed-point data type designated by `dataTypeId`. This function does not return the size of the storage container or the word length of the data type. To get the storage container size, use `ssGetDataTypeStorageContainerSize`. To get the data type word length, use `ssGetDataTypeFxpWordLength`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

Examples

An `sfix24_En10` data type has a word length of 24, but is actually stored in 32 bits during simulation. For this signal,

- `ssGetDataTypeFxpContainWordLen` returns `32`, which is the storage container word length in bits.
- `ssGetDataTypeFxpWordLength` returns `24`, which is the data type word length in bits.
- `ssGetDataTypeStorageContainerSize` or `sizeof( )` returns `4`, which is the storage container size in bytes.

TLC Functions

`FixPt_DataTypeFxpContainWordLen`

See Also

`ssGetDataTypeFxpWordLength`, `ssGetDataTypeStorageContainCat`, `ssGetDataTypeStorageContainerSize`

ssGetDataTypeFxpIsSigned

Purpose

Determine whether fixed-point registered data type is signed or unsigned

Syntax

```
extern int ssGetDataTypeFxpIsSigned (SimStruct *S, DTypeId
                                     dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered fixed-point data type for which you want to know whether it is signed.

Description

This function determines whether a registered fixed-point data type is signed:

- If the fixed-point data type is signed, the function returns `TRUE`. If the fixed-point data type is unsigned, the function returns `FALSE`.
- If the registered data type is `ScaledDouble`, the function returns `TRUE` or `FALSE` according to the signedness of the nonoverridden data type.
- If the registered data type is `single` or `double`, this function errors out.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeFxpIsSigned`

Purpose

Return word length of fixed-point registered data type

Syntax

```
extern int ssGetDataTypeFxpWordLength (SimStruct *S, DTypeId
                                       dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered fixed-point data type for which you want to know the word length.

Description

This function returns the word length of the fixed-point data type designated by `dataTypeId`. This function does not return the word length of the container of the data type. To get the container word length, use `ssGetDataTypeFxpContainWordLen`:

- If the registered data type is fixed point, this function returns the total word length including any sign bits, integer bits, and fractional bits.
- If the registered data type is `ScaledDouble`, this function returns the word length of the nonoverridden data type.
- If registered data type is `single` or `double`, this function errors out.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

Examples

An `sfix24_En10` data type has a word length of 24, but is actually stored in 32 bits during simulation. For this signal,

- `ssGetDataTypeFxpWordLength` returns `24`, which is the data type word length in bits.
- `ssGetDataTypeFxpContainWordLen` returns `32`, which is the storage container word length in bits.
- `ssGetDataTypeStorageContainerSize` or `sizeof( )` returns `4`, which is the storage container size in bytes.

TLC Functions

`FixPt_DataTypeFxpWordLength`

See Also

`ssGetDataTypeFxpContainWordLen`, `ssGetDataTypeFractionLength`, `ssGetDataTypeStorageContainerSize`

Purpose

Determine whether registered data type is fixed-point data type

Syntax

```
extern int ssGetDataTypeIsFixedPoint(SimStruct *S, DTypeId
                                     dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know whether it is fixed-point.

Description

This function determines whether a registered data type is a fixed-point data type:

- This function returns `TRUE` if the registered data type is fixed-point, and `FALSE` otherwise.
- If the registered data type is a pure Simulink integer, such as `int8`, this function returns `TRUE`.
- If the registered data type is `ScaledDouble`, this function returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeIsFixedPoint`

See Also

`ssGetDataTypeIsFloatingPoint`

ssGetDataTypeIsFloatingPoint

Purpose

Determine whether registered data type is floating-point data type

Syntax

```
extern int ssGetDataTypeIsFloatingPoint (SimStruct *S, DTypeId
                                                        dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know whether it is floating-point.

Description

This function determines whether a registered data type is `single` or `double`:

- If the registered data type is either `single` or `double`, this function returns `TRUE`, and `FALSE` is returned otherwise.
- If the registered data type is `ScaledDouble`, this function returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeIsFloatingPoint`

See Also

`ssGetDataTypeIsFixedPoint`

Purpose

Determine whether registered data type is supported by API for user-written fixed-point S-functions

Syntax

```
extern int ssGetDataTypeIsFxpFltApiCompat(SimStruct *S, DTypeId
                                          dataTypeId)
```

Arguments

`S`

SimStruct representing an S-function block.

`dataTypeId`

Data type ID of the registered data type for which you want to determine compatibility with the API for user-written fixed-point S-functions.

Description

This function determines whether the registered data type is supported by the API for user-written fixed-point S-functions. The supported data types are all standard Simulink data types, all fixed-point data types, and data type override data types.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

None. Checking for API-compatible data types is done in simulation. Checking for API-compatible data types is not supported in TLC.

ssGetDataTypeIsScalingPow2

Purpose

Determine whether registered data type has power-of-two scaling

Syntax

```
extern int ssGetDataTypeIsScalingPow2 (SimStruct *S, DTypeId
                                        dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know whether the scaling is strictly power-of-two.

Description

This function determines whether the registered data type is scaled strictly by a power of two. Fixed-point numbers can be represented as

$$\textit{real-world value} = (\textit{slope} \times \textit{integer}) + \textit{bias},$$

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}.$$

When *bias* = 0 and *fractional slope* = 1, the only scaling factor that remains is a power of two:

$$\textit{real-world value} = (2^{\textit{exponent}} \times \textit{integer}) = (2^{-\textit{fraction length}} \times \textit{integer}).$$

Trivial scaling is considered a case of power-of-two scaling, with the exponent being equal to zero.

Note Many fixed-point algorithms are designed to accept only power-of-two scaling. For these algorithms, you can call `ssGetDataTypeIsScalingPow2` in `mdlSetInputPortDataType` and `mdlSetOutputPortDataType`, to prevent unsupported data types from being accepted.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

`FixPt_DataTypeIsScalingPow2`

See Also

`ssGetDataTypeIsScalingTrivial`

ssGetDataTypeIsScalingTrivial

Purpose

Determine whether scaling of registered data type is slope = 1, bias = 0

Syntax

```
extern int ssGetDataTypeIsScalingTrivial (SimStruct *S, DTypeId
                                          dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know whether the scaling is trivial.

Description

This function determines whether the scaling of a registered data type is trivial. In [Slope Bias] representation, fixed-point numbers can be represented as

$$\textit{real-world value} = (\textit{slope} \times \textit{integer}) + \textit{bias}.$$

In the trivial case, $\textit{slope} = 1$ and $\textit{bias} = 0$.

In terms of binary-point-only scaling, the binary point is to the right of the least significant bit for trivial scaling, meaning that the fraction length is zero:

$$\textit{real-world value} = \textit{integer} \times 2^{-\textit{fraction length}} = \textit{integer} \times 2^0.$$

In either case, trivial scaling means that the real-world value is simply equal to the stored integer value:

$$\textit{real-world value} = \textit{integer}.$$

Scaling is always trivial for pure integers, such as `int8`, and also for the true floating-point types `single` and `double`.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages C

TLC Functions `FixPt_DataTypeIsScalingTrivial`

See Also `ssGetDataTypeIsScalingPow2`

ssGetDataTypeNumberOfChunks

Purpose

Return number of chunks in multiword storage container of registered data type

Syntax

```
extern int ssGetDataTypeNumberOfChunks(SimStruct *S,
                                       DTypeId dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know the number of chunks in its multiword storage container.

Description

This function returns the number of chunks in the multiword storage container of the fixed-point data type designated by `dataTypeId`. This function is valid only for a registered data type whose storage container uses a multiword representation. You can use the `ssGetDataTypeStorageContainCat` function to identify the storage container category; for multiword storage containers, the function returns the category `FXP_STORAGE_MULTIWORD`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

See Also

`ssGetDataTypeStorageContainCat`

Purpose

Return storage container category of registered data type

Syntax

```
extern fxpStorageContainerCategory
ssGetDataTypeStorageContainCat(SimStruct *S, DTypeId dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know the container category.

Description

This function returns the storage container category of the data type designated by `dataTypeId`. The container category returned by this function is used to store input and output signals, run-time parameters, and DWorks during Simulink simulations.

During simulation, fixed-point signals are held in one of the types of containers shown in the following table. Therefore in many cases, signals are represented in containers with more bits than their actual word length.

Fixed-Point Storage Containers

Container Category	Signal Word Length	Container Word Length	Container Size
`FXP_STORAGE_INT8` (signed) `FXP_STORAGE_UINT8` (unsigned)	1 to 8 bits	8 bits	1 byte
`FXP_STORAGE_INT16` (signed) `FXP_STORAGE_UINT16` (unsigned)	9 to 16 bits	16 bits	2 bytes

Fixed-Point Storage Containers (Continued)

Container Category	Signal Word Length	Container Word Length	Container Size
`FXP_STORAGE_INT32` (signed) `FXP_STORAGE_UINT32` (unsigned)	17 to 32 bits	32 bits	4 bytes
`FXP_STORAGE_OTHER_SINGLE_WORD`	33 to word length of `long` data type	Length of `long` data type	Length of `long` data type
`FXP_STORAGE_MULTIWORD`	Greater than the word length of `long` data type to 128 bits	Multiples of length of `long` data type to 128 bits	Multiples of length of `long` data type to 128 bits

When the number of bits in the signal word length is less than the size of the container, the word length bits are always stored in the least significant bits of the container. The remaining container bits must be sign extended to fit the bits of the container:

- If the data type is unsigned, then the sign-extended bits must be cleared to zero.
- If the data type is signed, then the sign-extended bits must be set to one for strictly negative numbers, and cleared to zero otherwise.

The `ssGetDataTypeStorageContainCat` function can also return the following values.

Other Storage Containers

Container Category	Description
FXP_STORAGE_UNKNOWN	Returned if the storage container category is unknown
FXP_STORAGE_SINGLE	Container type for a Simulink `single`
FXP_STORAGE_DOUBLE	Container type for a Simulink `double`
FXP_STORAGE_SCALEDDOUBLE	Container type for a data type that has been overridden with `Scaled double`

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

Because the mapping of storage containers in simulation to storage containers in code generation is not one-to-one, the TLC functions for storage containers in TLC are different from those in simulation. Refer to "Storage Container TLC Functions" on page A-11 for more information:

- `FixPt_DataTypeNativeType`
- `FixPt_DataTypeStorageDouble`
- `FixPt_DataTypeStorageSingle`
- `FixPt_DataTypeStorageScaledDouble`
- `FixPt_DataTypeStorageSInt`
- `FixPt_DataTypeStorageUInt`
- `FixPt_DataTypeStorageSLong`

- `FixPt_DataTypeStorageULong`
- `FixPt_DataTypeStorageSShort`
- `FixPt_DataTypeStorageUShort`

See Also `ssGetDataTypeStorageContainerSize`

Purpose

Return storage container size of registered data type

Syntax

```
extern size_t ssGetDataTypeStorageContainerSize
                                        (SimStruct *S, DTypeId
                                          dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know the container size.

Description

This function returns the storage container size of the data type designated by `dataTypeId`. This function returns the same value as would the `sizeof( )` function; it does not return the word length of either the storage container or the data type. To get the word length of the storage container, use `ssGetDataTypeFxpContainWordLen`. To get the word length of the data type, use `ssGetDataTypeFxpWordLength`.

The container of the size returned by this function stores input and output signals, run-time parameters, and DWorks during Simulink simulations. It is also the appropriate size measurement to pass to functions like `memcpy( )`.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

Examples

An `sfix24_En10` data type has a word length of 24, but is actually stored in 32 bits during simulation. For this signal,

- `ssGetDataTypeStorageContainerSize` or `sizeof( )` returns `4`, which is the storage container size in bytes.
- `ssGetDataTypeFxpContainWordLen` returns `32`, which is the storage container word length in bits.
- `ssGetDataTypeFxpWordLength` returns `24`, which is the data type word length in bits.

TLC Functions

`FixPt_GetDataTypeStorageContainerSize`

See Also

`ssGetDataTypeFxpContainWordLen`, `ssGetDataTypeFxpWordLength`, `ssGetDataTypeStorageContainCat`

Purpose

Return total slope of scaling of registered data type

Syntax

```
extern double ssGetDataTypeTotalSlope (SimStruct *S, DTypeId
                                        dataTypeId)
```

Arguments

`S`
: SimStruct representing an S-function block.

`dataTypeId`
: Data type ID of the registered data type for which you want to know the total slope.

Description

Fixed-point numbers can be represented as

$$\textit{real-world value} = (\textit{slope} \times \textit{integer}) + \text{bias},$$

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}.$$

This function returns the total slope, rather than the fractional slope, of the data type designated by `dataTypeId`. To get the fractional slope, use `ssGetDataTypeFracSlope`:

- If the registered data type has trivial scaling, including `double` and `single` data types, the function returns a total slope of `1`.
- If the registered data type is `ScaledDouble`, the function returns the total slope of the nonoverridden data type. Refer to the examples below.

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

ssGetDataTypeTotalSlope

Languages

C

Examples

The data type `sfix32_En4` becomes `flts32_En4` with data type override. The total slope returned by this function in either case is `0.0625` (2^{-4}).

The data type `ufix16_s7p98` becomes `fltu16_s7p98` with data type override. The total slope returned by this function in either case is `7.98`.

TLC Functions

`FixPt_DataTypeTotalSlope`

See Also

`ssGetDataTypeBias`, `ssGetDataTypeFixedExponent`, `ssGetDataTypeFracSlope`

Purpose

Record information collected during simulation

Syntax

```
extern void ssLogFixptInstrumentation
                                (SimStruct *S,
                                 double minValue,
                                 double maxValue,
                                 int countOverflows,
                                 int countSaturations,
                                 int countDivisionsByZero,
                                 char *pStrName)
```

Arguments

`S`
SimStruct representing an S-function block.

`minValue`
Minimum output value that occurred during simulation.

`maxValue`
Maximum output value that occurred during simulation.

`countOverflows`
Number of overflows that occurred during simulation.

`countSaturations`
Number of saturations that occurred during simulation.

`countDivisionsByZero`
Number of divisions by zero that occurred during simulation.

`*pStrName`
The string argument is currently unused.

Description

ssLogFixptInstrumentation records information collected during a simulation, such as output maximum and minimum, any overflows, saturations, and divisions by zero that occurred. The Fixed-Point Tool displays this information after a simulation.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages C

Purpose

Register fixed-point data type with binary-point-only scaling and return its data type ID

Syntax

```
extern DTypeId ssRegisterDataTypeFxpBinaryPoint
                                    (SimStruct *S,
                                     int isSigned,
                                     int wordLength,
                                     int fractionLength,
                                     int obeyDataTypeOverride)
```

Arguments

`S`
: SimStruct representing an S-function block.

`isSigned`
: `TRUE` if the data type is signed.

  `FALSE` if the data type is unsigned.

`wordLength`
: Total number of bits in the data type, including any sign bit.

`fractionLength`
: Number of bits in the data type to the right of the binary point.

`obeyDataTypeOverride`
: `TRUE` indicates that the **Data Type Override** setting for the subsystem is to be obeyed. Depending on the value of **Data Type Override**, the resulting data type could be `Double`, `Single`, `Scaled double`, or the fixed-point data type specified by the other arguments of the function.

  `FALSE` indicates that the **Data Type Override** setting is to be ignored.

Description

This function fully registers a fixed-point data type with the Simulink software and returns a data type ID. Note that unlike the standard Simulink function `ssRegisterDataType`, you do not need to take any additional registration steps. The data type ID can be used to specify

the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods in `simstruc.h`, such as `ssGetDataTypeSize`.

Use this function if you want to register a fixed-point data type with binary-point-only scaling. Alternatively, you can use one of the other fixed-point registration functions:

- Use `ssRegisterDataTypeFxpFSlopeFixExpBias` to register a data type with [Slope Bias] scaling by specifying the word length, fractional slope, fixed exponent, and bias.
- Use `ssRegisterDataTypeFxpScaledDouble` to register a scaled `double`.
- Use `ssRegisterDataTypeFxpSlopeBias` to register a data type with [Slope Bias] scaling.

If the registered data type is not one of the Simulink built-in data types, a Simulink Fixed Point software license is checked out. To prevent a Simulink Fixed Point software license from being checked out when you simply open or view a model, protect registration calls with

```
if (ssGetSimMode(S) != SS_SIMMODE_SIZES_CALL_ONLY )
 ssRegisterDataType...
```

Note Because of the nature of the assignment of data type IDs, you should always use API functions to extract information from a data type ID about a data type in your S-function. For more information, refer to "Data Type IDs" on page A-13.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

None. Data types should be registered in the Simulink software. Registration of data types is not supported in TLC.

See Also

`ssRegisterDataTypeFxpFSlopeFixExpBias`, `ssRegisterDataTypeFxpScaledDouble`, `ssRegisterDataTypeFxpSlopeBias`

ssRegisterDataTypeFxpFSlopeFixExpBias

Purpose

Register fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias, and return its data type ID

Syntax

```
extern DTypeId ssRegisterDataTypeFxpFSlopeFixExpBias
                                        (SimStruct *S,
                                         int isSigned,
                                         int wordLength,
                                         double fractionalSlope,
                                         int fixedExponent,
                                         double bias,
                                         int obeyDataTypeOverride)
```

Arguments

`S`
: SimStruct representing an S-function block.

`isSigned`
: `TRUE` if the data type is signed.

  `FALSE` if the data type is unsigned.

`wordLength`
: Total number of bits in the data type, including any sign bit.

`fractionalSlope`
: Fractional slope of the data type.

`fixedExponent`
: Exponent of the slope of the data type.

`bias`
: Bias of the scaling of the data type.

`obeyDataTypeOverride`
: `TRUE` indicates that the **Data Type Override** setting for the subsystem is to be obeyed. Depending on the value of **Data Type Override**, the resulting data type could be `Double`, `Single`, `Scaled double`, or the fixed-point data type specified by the other arguments of the function.

`FALSE` indicates that the **Data Type Override** setting is to be ignored.

Description

This function fully registers a fixed-point data type with the Simulink software and returns a data type ID. Note that unlike the standard Simulink function `ssRegisterDataType`, you do not need to take any additional registration steps. The data type ID can be used to specify the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods in `simstruc.h`, such as `ssGetDataTypeSize`.

Use this function if you want to register a fixed-point data type by specifying the word length, fractional slope, fixed exponent, and bias. Alternatively, you can use one of the other fixed-point registration functions:

- Use `ssRegisterDataTypeFxpBinaryPoint` to register a data type with binary-point-only scaling.
- Use `ssRegisterDataTypeFxpScaledDouble` to register a scaled double.
- Use `ssRegisterDataTypeFxpSlopeBias` to register a data type with [Slope Bias] scaling.

If the registered data type is not one of the Simulink built-in data types, a Simulink Fixed Point software license is checked out. To prevent a Simulink Fixed Point software license from being checked out when you simply open or view a model, protect registration calls with

```
if (ssGetSimMode(S) != SS_SIMMODE_SIZES_CALL_ONLY )
 ssRegisterDataType...
```

Note Because of the nature of the assignment of data type IDs, you should always use API functions to extract information from a data type ID about a data type in your S-function. For more information, refer to "Data Type IDs" on page A-13.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

None. Data types should be registered in the Simulink software. Registration of data types is not supported in TLC.

See Also

`ssRegisterDataTypeFxpBinaryPoint`, `ssRegisterDataTypeFxpScaledDouble`, `ssRegisterDataTypeFxpSlopeBias`

Purpose

Register scaled double data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias, and return its data type ID

Syntax

```
extern DTypeId ssRegisterDataTypeFxpScaledDouble
                                        (SimStruct *S,
                                         int isSigned,
                                         int wordLength,
                                         double fractionalSlope,
                                         int fixedExponent,
                                         double bias,
                                         int obeyDataTypeOverride)
```

Arguments

`S`
: SimStruct representing an S-function block.

`isSigned`
: `TRUE` if the data type is signed.

  `FALSE` if the data type is unsigned.

`wordLength`
: Total number of bits in the data type, including any sign bit.

`fractionalSlope`
: Fractional slope of the data type.

`fixedExponent`
: Exponent of the slope of the data type.

`bias`
: Bias of the scaling of the data type.

`obeyDataTypeOverride`
: `TRUE` indicates that the **Data Type Override** setting for the subsystem is to be obeyed. Depending on the value of **Data Type Override**, the resulting data type could be `Double`, `Single`,

`Scaled double`, or the fixed-point data type specified by the other arguments of the function.

`FALSE` indicates that the **Data Type Override** setting is to be ignored.

Description

This function fully registers a fixed-point data type with the Simulink software and returns a data type ID. Note that unlike the standard Simulink function `ssRegisterDataType`, you do not need to take any additional registration steps. The data type ID can be used to specify the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods in `simstruc.h`, such as `ssGetDataTypeSize`.

Use this function if you want to register a scaled double data type. Alternatively, you can use one of the other fixed-point registration functions:

- Use `ssRegisterDataTypeFxpBinaryPoint` to register a data type with binary-point-only scaling.
- Use `ssRegisterDataTypeFxpFSlopeFixExpBias` to register a data type with [Slope Bias] scaling by specifying the word length, fractional slope, fixed exponent, and bias.
- Use `ssRegisterDataTypeFxpSlopeBias` to register a data type with [Slope Bias] scaling.

If the registered data type is not one of the Simulink built-in data types, a Simulink Fixed Point software license is checked out. To prevent a Simulink Fixed Point software license from being checked out when you simply open or view a model, protect registration calls with

```
if (ssGetSimMode(S) != SS_SIMMODE_SIZES_CALL_ONLY )
 ssRegisterDataType...
```

Note Because of the nature of the assignment of data type IDs, you should always use API functions to extract information from a data type ID about a data type in your S-function. For more information, refer to "Data Type IDs" on page A-13.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages

C

TLC Functions

None. Data types should be registered in the Simulink software. Registration of data types is not supported in TLC.

See Also

`ssRegisterDataTypeFxpBinaryPoint`, `ssRegisterDataTypeFxpFSlopeFixExpBias`, `ssRegisterDataTypeFxpSlopeBias`

ssRegisterDataTypeFxpSlopeBias

Purpose

Register data type with [Slope Bias] scaling and return its data type ID

Syntax

```
extern DTypeId ssRegisterDataTypeFxpSlopeBias
                                            (SimStruct *S,
                                             int isSigned,
                                             int wordLength,
                                             double totalSlope,
                                             double bias,
                                             int obeyDataTypeOverride)
```

Arguments

`S`
: SimStruct representing an S-function block.

`isSigned`
: `TRUE` if the data type is signed.

  `FALSE` if the data type is unsigned.

`wordLength`
: Total number of bits in the data type, including any sign bit.

`totalSlope`
: Total slope of the scaling of the data type.

`bias`
: Bias of the scaling of the data type.

`obeyDataTypeOverride`
: `TRUE` indicates that the **Data Type Override** setting for the subsystem is to be obeyed. Depending on the value of **Data Type Override**, the resulting data type could be `Double`, `Single`, `Scaled double`, or the fixed-point data type specified by the other arguments of the function.

  `FALSE` indicates that the **Data Type Override** setting is to be ignored.

Description

This function fully registers a fixed-point data type with the Simulink software and returns a data type ID. Note that unlike the standard

Simulink function `ssRegisterDataType`, you do not need to take any additional registration steps. The data type ID can be used to specify the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods in `simstruc.h`, such as `ssGetDataTypeSize`.

Use this function if you want to register a fixed-point data type with [Slope Bias] scaling. Alternately, you can use one of the other fixed-point registration functions:

- Use `ssRegisterDataTypeFxpBinaryPoint` to register a data type with binary-point-only scaling.
- Use `ssRegisterDataTypeFxpFSlopeFixExpBias` to register a data type with [Slope Bias] scaling by specifying the word length, fractional slope, fixed exponent, and bias.
- Use `ssRegisterDataTypeFxpScaledDouble` to register a scaled double.

If the registered data type is not one of the Simulink built-in data types, a Simulink Fixed Point software license is checked out. To prevent a Simulink Fixed Point software license from being checked out when you simply open or view a model, protect registration calls with

```
if (ssGetSimMode(S) != SS_SIMMODE_SIZES_CALL_ONLY )
 ssRegisterDataType...
```

Note Because of the nature of the assignment of data type IDs, you should always use API functions to extract information from a data type ID about a data type in your S-function. For more information, refer to "Data Type IDs" on page A-13.

Requirement

To use this function, you must include `fixedpoint.h` and `fixedpoint.c`. For more information, see "Structure of the S-Function" on page A-5.

Languages C

TLC Functions None.

See Also `ssRegisterDataTypeFxpBinaryPoint`, `ssRegisterDataTypeFxpFSlopeFixExpBias`, `ssRegisterDataTypeFxpScaledDouble`

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

Z