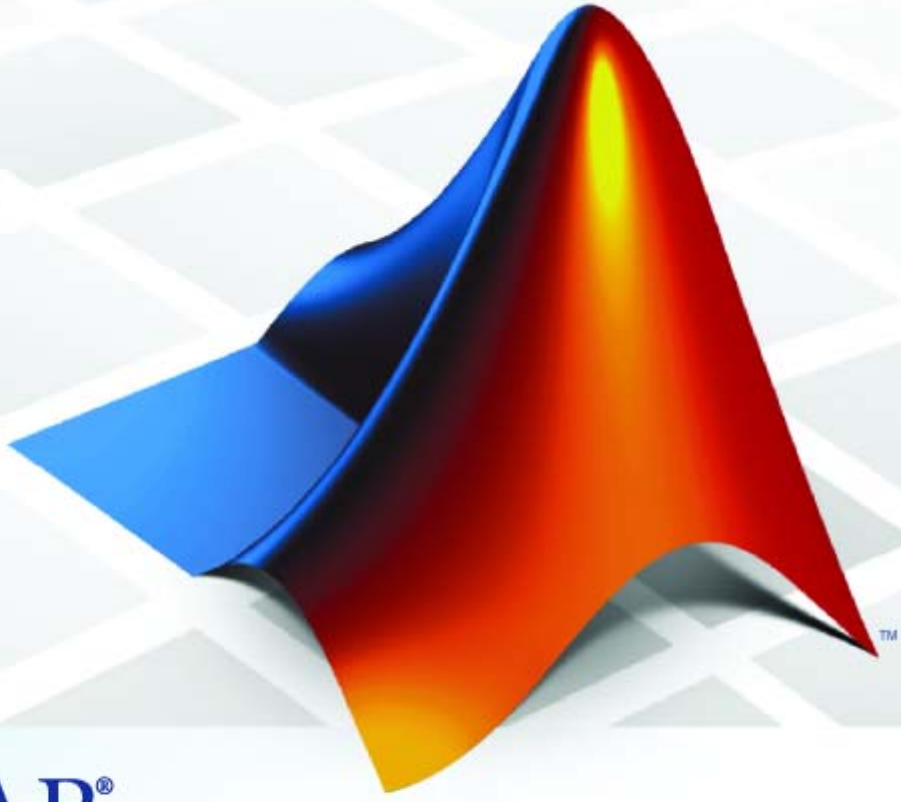
Simulink® Fixed Point™ 5
User's Guide



MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The 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

© COPYRIGHT 1995–2008 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

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)

Contents

1 Getting Started

Product Overview 1-2

What You Need to Get Started 1-4
Installation 1-4
Sharing Fixed-Point Models 1-4
Demos 1-5

Physical Quantities and Measurement Scales 1-8
Introduction 1-8
Selecting a Measurement Scale 1-9
Example: Selecting a Measurement Scale 1-11

Why Use Fixed-Point Hardware? 1-16

Why Use the Simulink® Fixed Point™ Software? 1-18

The Development Cycle 1-19

Simulink® Fixed Point™ Software Features 1-21
Configuring Blocks with Fixed-Point Output 1-21
Configuring Blocks with Fixed-Point Parameters 1-29
Passing Fixed-Point Data Between Simulink® Models and the MATLAB® Software 1-32
Additional Features and Capabilities 1-35

Example: Converting from Doubles to Fixed Point 1-37
About This Example 1-37
Block Descriptions 1-38
Simulation Results 1-38

2 Data Types and Scaling

Overview **2-2**

Fixed-Point Numbers **2-3**
About Fixed-Point Numbers **2-3**
Signed Fixed-Point Numbers **2-4**
Binary Point Interpretation **2-4**
Scaling **2-5**
Quantization **2-7**
Range and Precision **2-9**
Constant Scaling for Best Precision **2-12**
Fixed-Point Data Type and Scaling Notation **2-15**

Floating-Point Numbers **2-17**
About Floating-Point Numbers **2-17**
Scientific Notation **2-17**
The IEEE® Format **2-19**
Range and Precision **2-22**
Exceptional Arithmetic **2-24**

3 Arithmetic Operations

Overview **3-3**

Limitations on Precision **3-4**
Introduction **3-4**
Rounding **3-4**
Padding with Trailing Zeros **3-14**
Example: Limitations on Precision and Errors **3-15**
Example: Maximizing Precision **3-15**

Limitations on Range **3-17**
Introduction **3-17**
Saturation and Wrapping **3-18**
Guard Bits **3-21**

Example: Limitations on Range 3-21

Recommendations for Arithmetic and Scaling 3-23
Introduction 3-23
Addition 3-24
Accumulation 3-27
Multiplication 3-27
Gain 3-29
Division 3-31
Summary 3-33

Parameter and Signal Conversions 3-34
Introduction 3-34
Parameter Conversions 3-35
Signal Conversions 3-36

Rules for Arithmetic Operations 3-39
Introduction 3-39
Computational Units 3-39
Addition and Subtraction 3-40
Multiplication 3-45
Division 3-52
Shifts 3-54

Example: Conversions and Arithmetic Operations 3-56

4 Realization Structures

Overview 4-2
Introduction 4-2
Realizations and Data Types 4-2

Targeting an Embedded Processor 4-4
Introduction 4-4
Size Assumptions 4-4
Operation Assumptions 4-4
Design Rules 4-5

Canonical Forms **4-7**
Introduction **4-7**
Direct Form II **4-8**
Series Cascade Form **4-11**
Parallel Form **4-14**

5 Fixed-Point Advisor

Working With the Fixed-Point Advisor **5-2**
Introduction to the Fixed-Point Advisor **5-2**
Running the Fixed-Point Advisor **5-2**
Fixing a Task Failure **5-3**
Automatically Fixing Failures **5-3**
Batch Fixing Failures **5-4**

Tutorial: Converting a Model from Floating- to Fixed-Point **5-5**
About This Tutorial **5-5**
Starting the Fixed-Point Advisor **5-5**
Prepare Model for Conversion **5-6**
Prepare for Data Typing and Scaling **5-15**
Perform Data Typing and Scaling **5-18**
Prepare for Code Generation **5-21**

6 Fixed-Point Tool

Overview of the Fixed-Point Tool **6-2**
Introduction to the Fixed-Point Tool **6-2**
Opening the Fixed-Point Tool **6-2**
Understanding the Interface **6-3**

Working with the Fixed-Point Tool **6-5**
Fixed-Point Tool Workflow **6-5**
Proposing Scaling **6-5**
Reviewing Scaling Proposals **6-6**

Applying Scaling 6-11

Introduction to the Tutorial 6-12
Opening the Demo Model 6-12
About the Demo Model 6-13
Simulation Setup 6-14
Idealized Feedback Design 6-14
Digital Controller Realization 6-15

Tutorial: Feedback Controller 6-18
Before You Begin 6-18
Initial Guess at Scaling 6-19
Data Type Override 6-20
Automatic Scaling 6-24

7 Tutorial: Producing Lookup Table Data

Overview 7-2

Worst-Case Error for a Lookup Table 7-3
What Is Worst-Case Error for a Lookup Table? 7-3
Example: Square Root Function 7-3

Creating Lookup Tables for a Sine Function 7-6
Introduction 7-6
Parameters for fixpt_look1_func_approx 7-6
Setting Function Parameters for the Lookup Table 7-8
Example: Using errmax with Unrestricted Spacing 7-8
Example: Using nptsmax with Unrestricted Spacing 7-12
Example: Using errmax with Even Spacing 7-14
Example: Using nptsmax with Even Spacing 7-15
Example: Using errmax with Power of Two Spacing 7-16
Example: Using nptsmax with Power of Two Spacing 7-18
Specifying Both errmax and nptsmax 7-19
Comparing the Examples 7-20

Summary: Using the Lookup Table Functions 7-22

Effect of Spacing on Speed, Error, and Memory Usage **7-23**
Introduction **7-23**
Data ROM Required **7-24**
Determining Out-of-Range Inputs **7-25**
Determining Input Location **7-25**
Interpolation **7-27**
Conclusion **7-29**

8 Code Generation

Overview **8-2**

Code Generation Support **8-3**
Introduction **8-3**
Languages **8-3**
Storage Class of Variables **8-3**
Storage Class of Parameters **8-4**
Rounding Modes **8-4**
Overflow Handling **8-4**
Blocks **8-4**
Scaling **8-5**

Accelerating Fixed-Point Models **8-6**

Using External Mode or Rapid Simulation Target **8-8**
Introduction **8-8**
External Mode **8-8**
Rapid Simulation Target **8-9**

Optimizing Your Generated Code **8-10**
Introduction **8-10**
Restrict Data Type Word Lengths **8-11**
Avoid Fixed-Point Scalings with Bias **8-11**
Wrap and Round to Floor or Simplest **8-11**
Limit the Use of Custom Storage Classes **8-13**
Limit the Use of Unevenly Spaced Lookup Tables **8-13**

Minimize the Variety of Similar Fixed-Point Utility Functions 8-14

Optimizing Your Generated Code with the Model Advisor 8-15
Introduction 8-15
Optimize Lookup Table Data 8-16
Reduce Cumbersome Multiplications 8-16
Optimize the Number of Multiply and Divide Operations . . 8-17
Reduce Multiplies and Divides with Nonzero Bias 8-18
Eliminate Mismatched Scaling 8-18
Minimize Internal Conversion Issues 8-20
Use the Most Efficient Rounding 8-22

9 Fixed-Point Advisor Reference

Fixed-Point Advisor 9-2
Fixed-Point Advisor Overview 9-3

Prepare Model for Conversion 9-6
Prepare Model for Conversion Overview 9-7
Verify model simulation settings 9-8
Address unsupported blocks 9-9
Verify update diagram status 9-10
Set up signal logging 9-11
Create simulation reference data 9-12
Verify hardware selection 9-13
Verify Fixed-Point Conversion Guidelines Overview 9-14
Check model configuration data validity diagnostic parameters settings 9-15
Implement logic signals as Boolean data 9-16
Check for proper bus usage 9-17

Prepare for Data Typing and Scaling 9-18
Prepare for Data Typing and Scaling Overview 9-19
Remove output data type inheritance 9-20
Relax input data type settings 9-21
Verify Stateflow charts have strong data typing with Simulink 9-22

Specify block minimum and maximum values 9-23

Perform Data Typing and Scaling 9-24
Perform Data Typing and Scaling Overview 9-25
Propose scaling for Inport blocks 9-26
Propose scaling for Constant blocks 9-30
Propose scaling for blocks 9-34
Resolve scaling conflicts 9-39
Summarize data types 9-41
Propose scaling for parameters 9-42
Check for numeric errors 9-45
Analyze logged signals 9-46

Prepare for Code Generation 9-47
Prepare for Code Generation Overview 9-48
Disable signal logging 9-49
Identify blocks that generate expensive saturation and rounding code 9-50
Identify questionable fixed-point operations 9-51

10 Function Reference

Global Changes 10-1

Tools 10-1

Functions — Alphabetical List

11

Writing Fixed-Point S-Functions

A

Data Type Support **A-3**
Supported Data Types **A-3**
The Treatment of Integers **A-4**
Data Type Override **A-4**

Structure of the S-Function **A-6**

Storage Containers **A-8**
Introduction **A-8**
Storage Containers in Simulation **A-8**
Storage Containers in Code Generation **A-12**

Data Type IDs **A-15**
The Assignment of Data Type IDs **A-15**
Registering Data Types **A-16**
Setting and Getting Data Types **A-18**
Getting Information About Data Types **A-19**
Converting Data Types **A-21**

Overflow Handling and Rounding Methods **A-22**
Tokens for Overflow Handling and Rounding Methods . . . **A-22**
Overflow Logging Structure **A-23**

Creating MEX-Files **A-24**
Introduction **A-24**
MEX-Files on UNIX® **A-24**
MEX-Files on Windows® **A-24**

Fixed-Point S-Function Examples **A-26**
List of Fixed-Point S-Function Examples **A-26**
Getting the Input Port Data Type **A-27**

Setting the Output Port Data Type **A-29**
Interpreting an Input Value **A-30**
Writing an Output Value **A-32**
Using the Input Data Type to Determine the Output Data Type **A-34**

API Functions — Alphabetical List **A-35**

B Selected Bibliography

Glossary

Index

1

Getting Started

Product Overview (p. 1-2)	Describes the Simulink® Fixed Point™ software
What You Need to Get Started (p. 1-4)	Describes how to ensure that the Simulink Fixed Point software is installed, and introduces its demos
Physical Quantities and Measurement Scales (p. 1-8)	Provides an overview of measurement scales and representing numbers
Why Use Fixed-Point Hardware? (p. 1-16)	Discusses the limitations and benefits of fixed-point hardware
Why Use the Simulink® Fixed Point™ Software? (p. 1-18)	Describes the advantages of using the Simulink Fixed Point software
The Development Cycle (p. 1-19)	Provides an overview of the development cycle for simulating dynamic systems
Simulink® Fixed Point™ Software Features (p. 1-21)	Introduces key features of the Simulink Fixed Point software
Example: Converting from Doubles to Fixed Point (p. 1-37)	Provides an example based on the `fxpdemo_dbl2fix` demo, which highlights many of the key features of the Simulink Fixed Point software

Product Overview

The Simulink® Fixed Point™ software enables the intrinsic fixed-point capabilities of the following products across the Simulink® product family:

- Simulink
- Stateflow®
- Signal Processing Blockset™
- Communications Blockset™
- Target Support Package™ TC2
- Target Support Package TC6
- Target Support Package IC1
- Target Support Package FM5
- Video and Image Processing Blockset™

The following products can be used to generate fixed-point code when used with the Simulink Fixed Point software:

- Real-Time Workshop®
- Real-Time Workshop® Embedded Coder™
- Stateflow® Coder™
- xPC Target™

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 for execution on a fixed-point embedded processor with the Real-Time Workshop 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 Signal Processing Blockset software, refer to “Working with Fixed-Point Data” in the Signal Processing Blockset 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” on page 1-4
“Sharing Fixed-Point Models” on page 1-4
“Demos” on page 1-5

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, see your platform-specific MATLAB Installation Guide.

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 by doing the following:

1. Access the Fixed-Point Tool from the model by selecting **Tools > Fixed-Point > Fixed-Point Tool**.
2. Set the **Logging mode** parameter to `Force off` model wide.
3. Set the **Data type override** parameter to `True doubles` or `True singles` model wide.

Note If a parameter in your model specifies a `fi` object, you can prevent the checkout of a Fixed-Point Toolbox™ license by setting the `fipref DataTypeOverride` property to `TrueDoubles`. See "Licensing" in the *Fixed-Point Toolbox User's Guide* for more information.

This procedure allows you to share fixed-point Simulink models among people in your company who may or may not have the Simulink Fixed Point software.

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 two groups: basic demos that illustrate the basic functionality of the Simulink Fixed Point software, and advanced demos that illustrate the functionality of systems built with fixed-point blocks. All demos are located in the `fxpdemos` directory.

You can access Simulink Fixed Point demos from the Help browser **Demos** pane (see "Demos in the Help Browser" in the MATLAB documentation), or by typing at the MATLAB command line:

```
demo simulink 'Simulink Fixed Point'
```

Basic Simulink® Fixed Point™ Demos

The basic demos are listed in the following table.

Demo Name	Description
Double to Fixed-Point Conversion	Convert a double-precision value to a fixed-point value.
Fixed-Point to Fixed-Point Conversion	Convert a fixed-point value to another fixed-point value.
Fixed-Point to Fixed-Point Inherited Conversion	Convert a fixed-point value to an inherited fixed-point value.
Fixed-Point Sine	Add and multiply two fixed-point sine wave signals.
Fixed-Point Filters	Simulate implementations of a fixed-point filter.
Scaling a Fixed-Point Control Design	Simulate a fixed-point feedback design.

"Example: Converting from Doubles to Fixed Point" on page 1-37 discusses the Double to Fixed-Point Conversion demo, while Chapter 6, "Fixed-Point Tool" discusses the Scaling a Fixed-Point Control Design demo.

Advanced Simulink® Fixed Point™ Demos

The advanced demos are intended to show you how to build and test systems suited to your particular needs. The output of these demos is compared to the output of analogous built-in Simulink blocks with identical input.

The advanced demos are listed in the following table.

Demo Name	Description
Fixed-Point Integrators	Compare output from the Integrator Trapezoidal, Integrator Backward, and Integrator Forward blocks to output from the Simulink Discrete Integrator block.

Demo Name	Description
Fixed-Point Derivatives	Compare output from the Derivative and Derivative: Filtered realizations to output from the derivatives built using the Simulink Discrete Filter and Transfer Fcn blocks.
Fixed-Point Lead and Lag Filters	Compare output from the Lead and Lag Filter block to output from analogous filters built using the Simulink Discrete Filter block.
Fixed-Point State Space	Compare output from the State-Space Realization to output from the analogous built-in Simulink State-Space and Discrete State-Space blocks.
Fixed-Point Function Approximation	Compare the fixed-point lookup approximation of a function with the ideal function.
Fixed-Point Direct Form Filter	Display the implementation of a fixed-point direct form filter constructed with Simulink blocks.
Fixed-Point Parallel Form Filter	Display the implementation of a fixed-point parallel form filter constructed with Simulink blocks.
Fixed-Point Series Cascade Form Filter	Display the implementation of a fixed-point series cascade form filter constructed with Simulink blocks.
Fixed-Point S-Function Example: Querying Properties	Demonstrate S-functions that probe signal properties.
Fixed-Point S-Function Example: Arithmetic Shift	Demonstrate an S-function that performs an arithmetic shift.
Fixed-Point S-Function Example: Fixed-Point Source	Demonstrate an S-function that generates a fixed-point constant value.
Fixed-Point S-Function Example: Data Type Propagation	Demonstrate an S-function that enforces data types across multiple signals.
Fixed-Point S-Function Example: Product and Sum	Demonstrate an S-function that computes a fixed-point product and sum operation.

Physical Quantities and Measurement Scales

In this section...
"Introduction" on page 1-8
"Selecting a Measurement Scale" on page 1-9
"Example: Selecting a Measurement Scale" on page 1-11

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:

$$V = \begin{cases} \frac{100°C}{100\ \text{bits}} \cdot Q_1 & \text{Conversion 1} \\ \frac{100°C}{255\ \text{bits}} \cdot Q_2 & \text{Conversion 2} \\ \frac{100°C}{200\ \text{bits}} \cdot Q_3 & \text{Conversion 3} \end{cases}$$

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

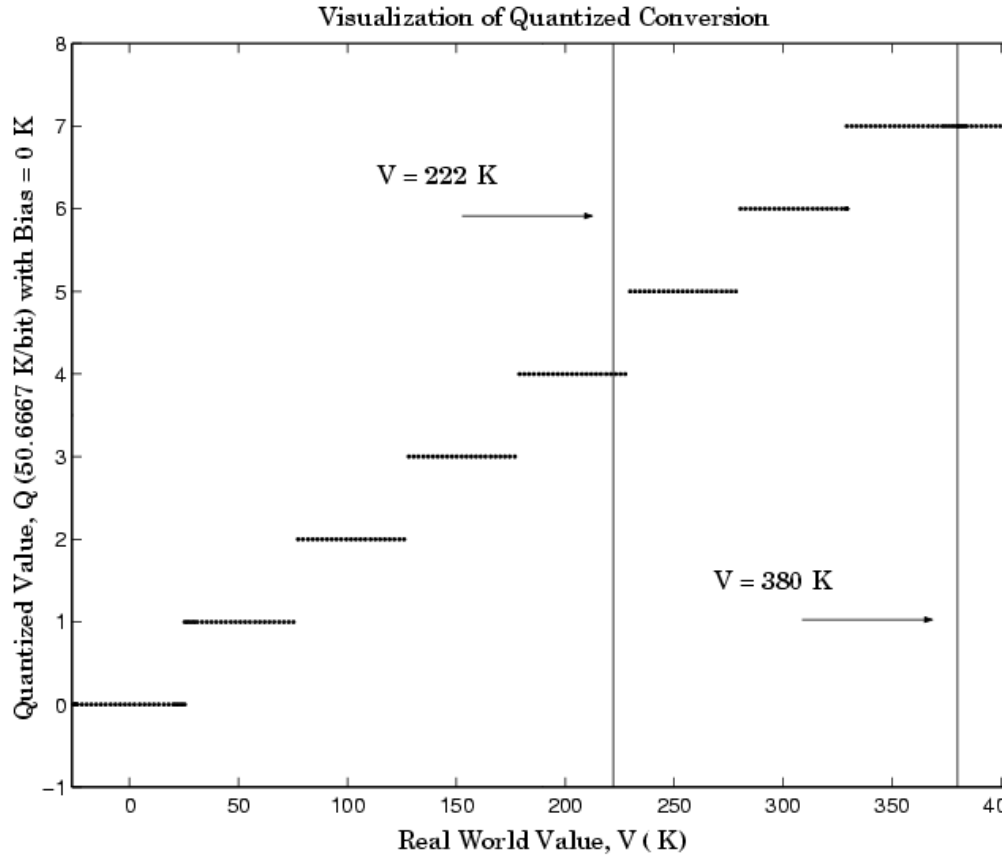
On typical electronically controlled internal combustion engines, the flow of fuel is regulated to obtain the desired ratio of air to fuel in the cylinders just prior to combustion. Therefore, knowledge of the current air flow rate is required. Some manufacturers use sensors that directly measure air 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

$$V = \frac{380\ \text{K}}{7.5\ \text{bit}} \cdot Q$$

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

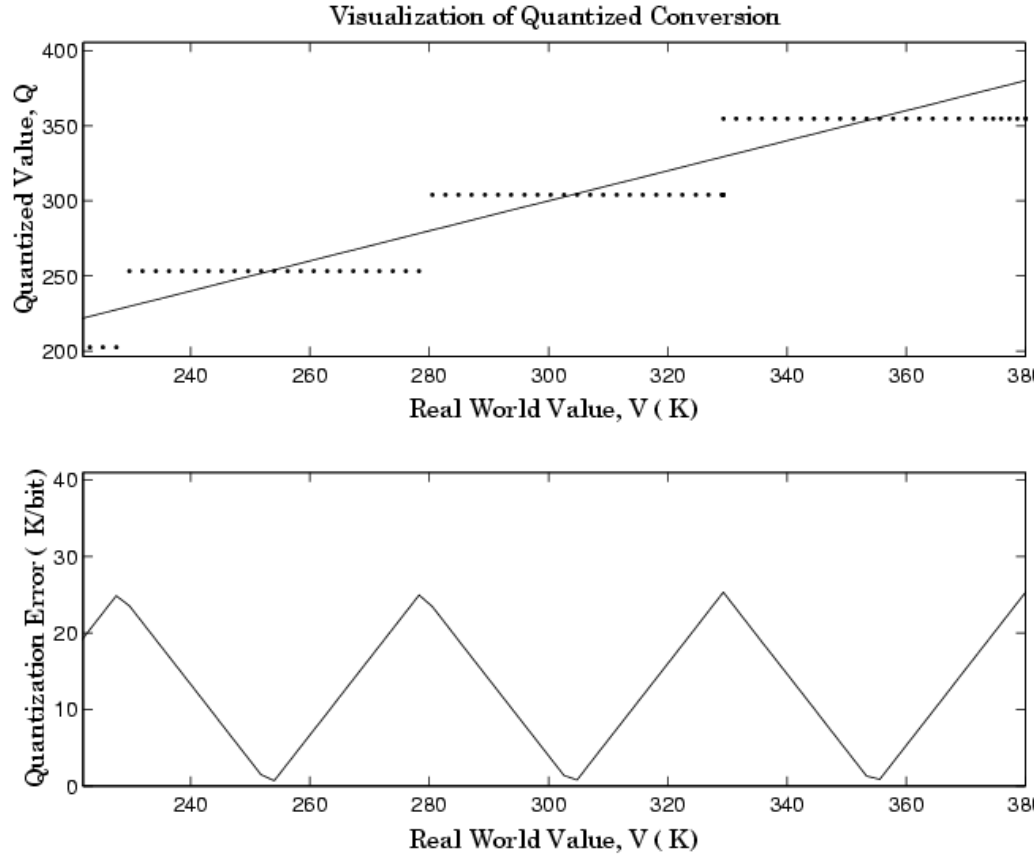


Notice that there are 7.5 possible quantization values. This is because only half of the first bit corresponds to temperatures (real-world values) greater than zero.

The quantization error is

$$-25.33 \text{ K/bit} \le \textit{Error} \le 25.33 \text{ K/bit}$$

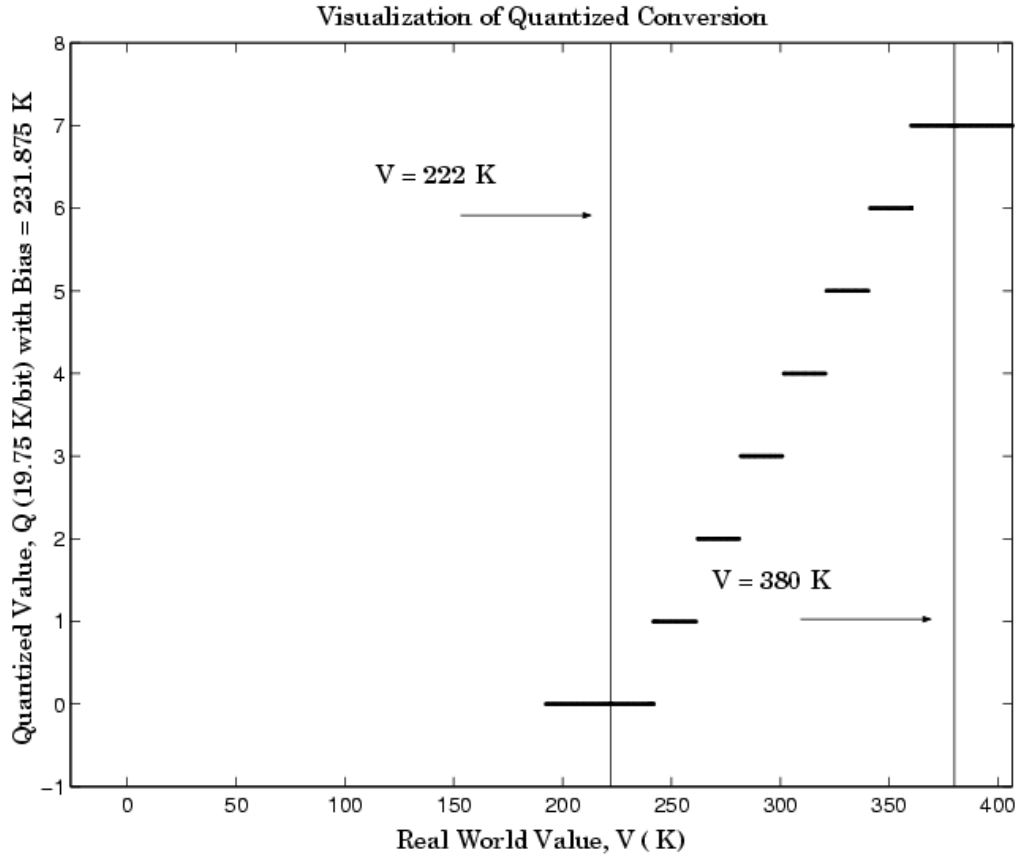
The range of interest of the quantized conversion and the absolute value of the quantized error are shown in the following figure.



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

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

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

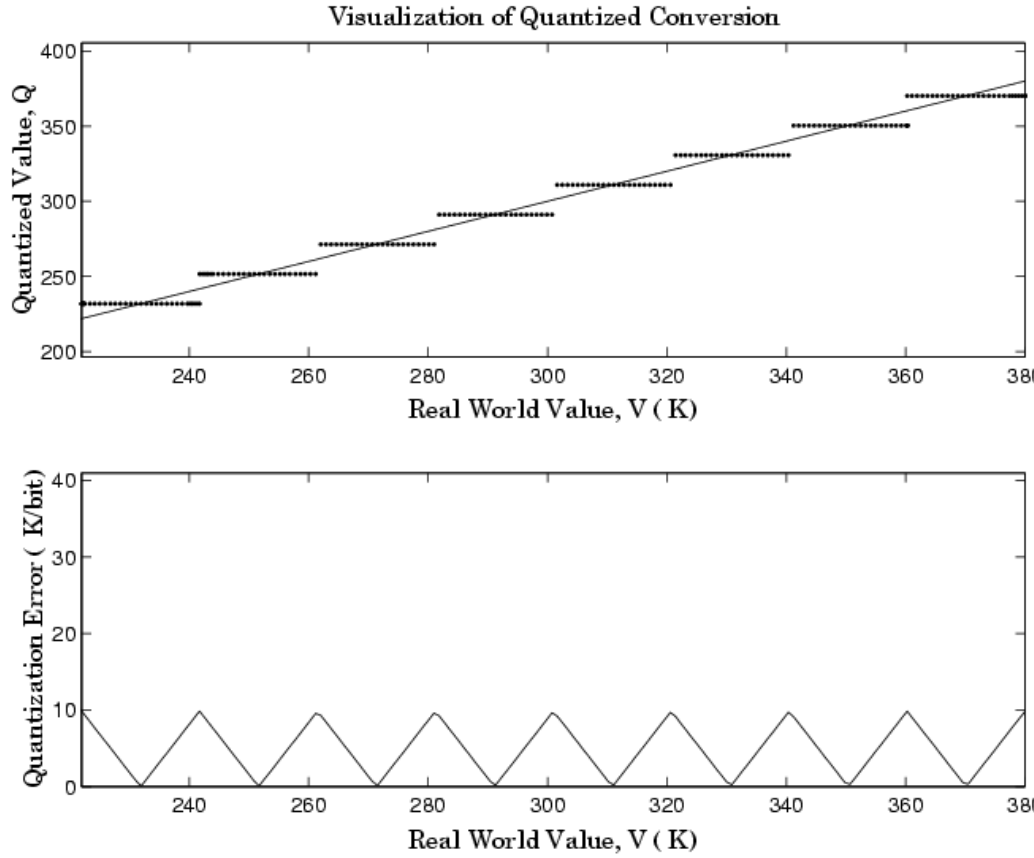


The quantization error is

$$-9.875 \text{ K/bit} \leq Error \leq 9.875 \text{ K/bit}$$

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?

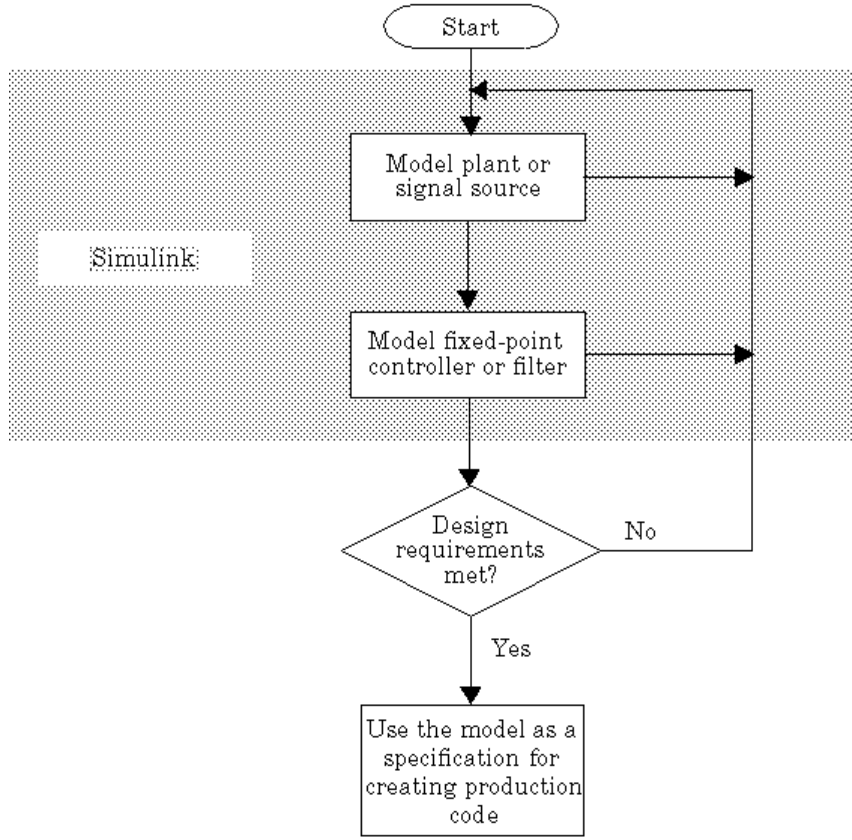
The Simulink® Fixed Point™ software allows you to efficiently design control systems and digital filters that you will implement using fixed-point arithmetic. With the Simulink Fixed Point software, you can construct Simulink® and Stateflow® models that contain detailed fixed-point information about 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 Real-Time Workshop® 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 Real-Time Workshop® 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 Real-Time Workshop® 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.

Simulink® Fixed Point™ Software Features

In this section...

"Configuring Blocks with Fixed-Point Output" on page 1-21

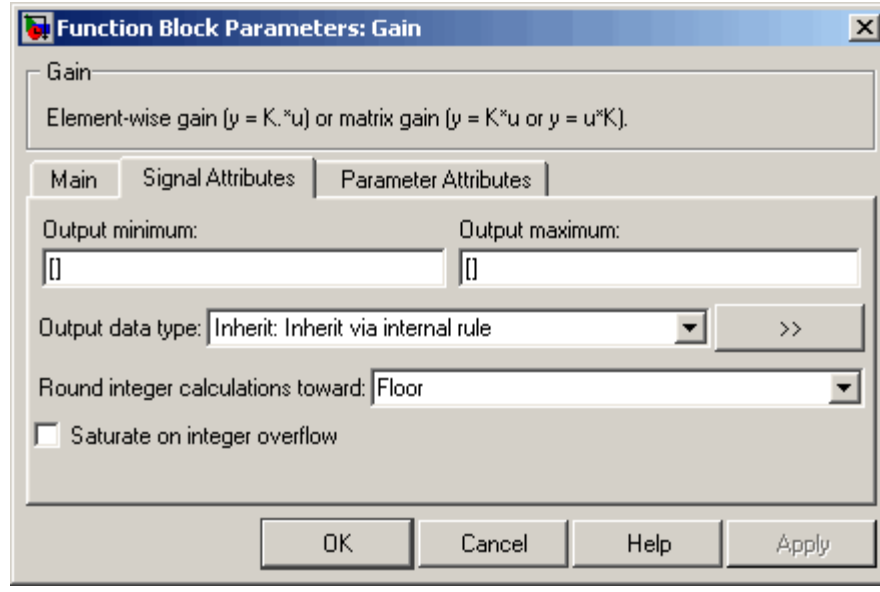
"Configuring Blocks with Fixed-Point Parameters" on page 1-29

"Passing Fixed-Point Data Between Simulink® Models and the MATLAB® Software" on page 1-32

"Additional Features and Capabilities" on page 1-35

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 1-27
- "Overflow Handling" on page 1-28
- "Locking the Output Scaling" on page 1-28
- "Real-World Values Versus Stored Integer Values" on page 1-28

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 *Using Simulink* 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 2-17 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')`.

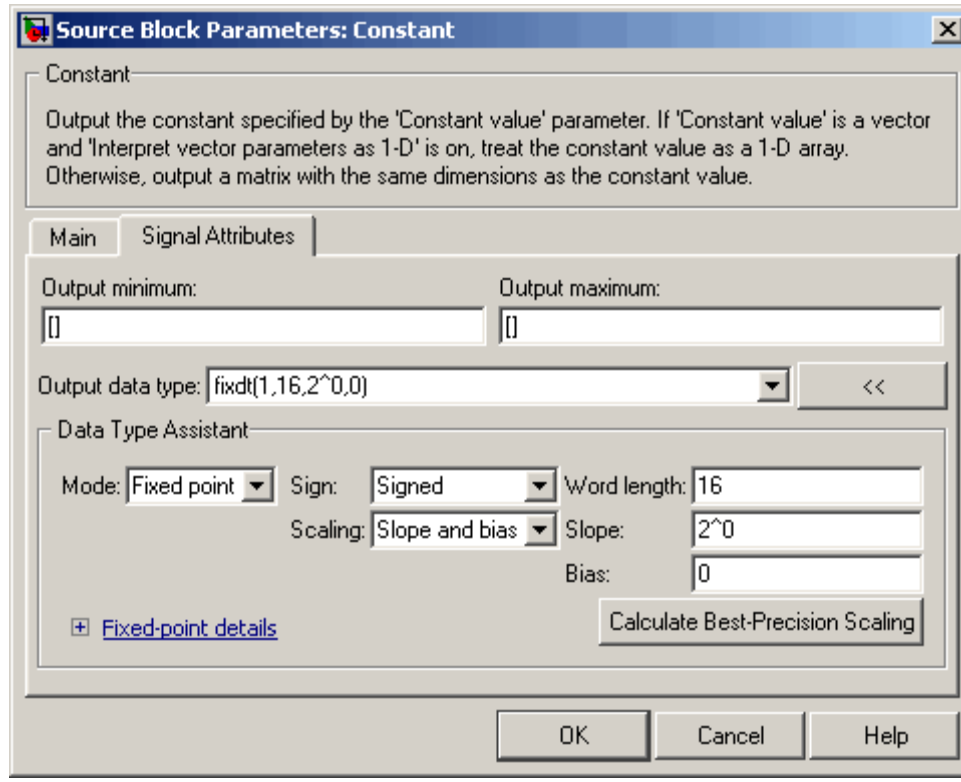
You can also specify a nonstandard floating-point number that mimics the IEEE style. For this data type, the fraction (mantissa) can range from 1 to 52 bits and the exponent can range from 1 to 11 bits. For example, to configure a nonstandard floating-point number having 32 total bits and 9 exponent bits, specify **Output data type** as `float(32,9)`.

Note These numbers are normalized with a hidden leading 1 for all exponents except the smallest possible exponent. However, the largest possible exponent might not be treated as a flag for Infs or NaNs.

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 *Using Simulink*.

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:

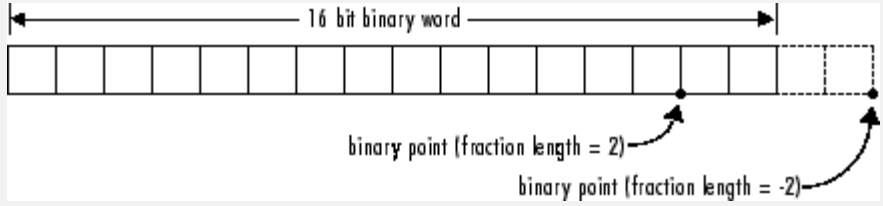


You can set the following fixed-point attributes:

Sign. 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.

Scaling Mode	Description
`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 Bias] Scaling" on page 2-7 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 **Round integer calculations toward** parameter. The following rounding modes are supported:

- `Zero` — This mode rounds toward zero and is equivalent to the MATLAB® `fix` 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.
- `Ceiling` — This mode rounds toward positive infinity and is equivalent to the MATLAB `ceil` function.

- `Floor` — This mode rounds toward negative infinity and is equivalent to the MATLAB `floor` function.
- `Simplest` — This mode automatically chooses between round toward floor and round toward zero to produce generated code that is as efficient as possible.

For more information about each of these rounding modes, see "Rounding" on page 3-4.

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.

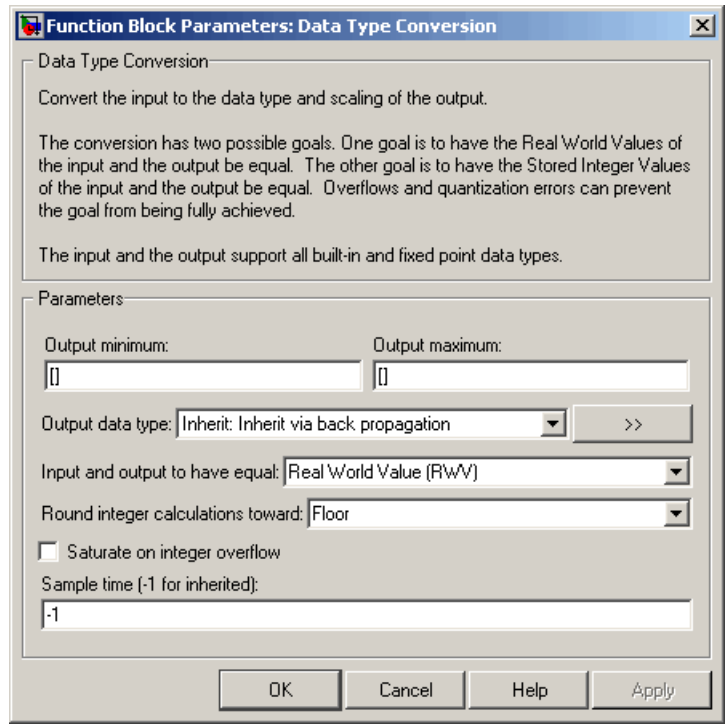
Locking the Output Scaling

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

When locked, the Fixed-Point Tool and automatic scaling script `autofixexp` (see "Automatic Scaling" on page 1-36) do not change the output scaling. Otherwise, the tool and `autofixexp` script are free to adjust the scaling.

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 1-30
- "Specifying Fixed-Point Values Via Parameter Objects" on page 1-31

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.

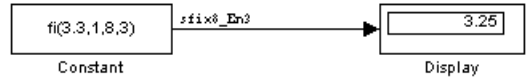
Specifying Fixed-Point Values Directly

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,true,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
   ```

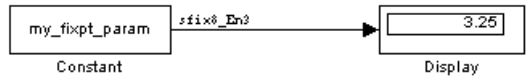
2. Specify either the name of a `fi` object or an expression that includes the `fi` constructor function as the parameter object's `Value` property:

   ```
   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.

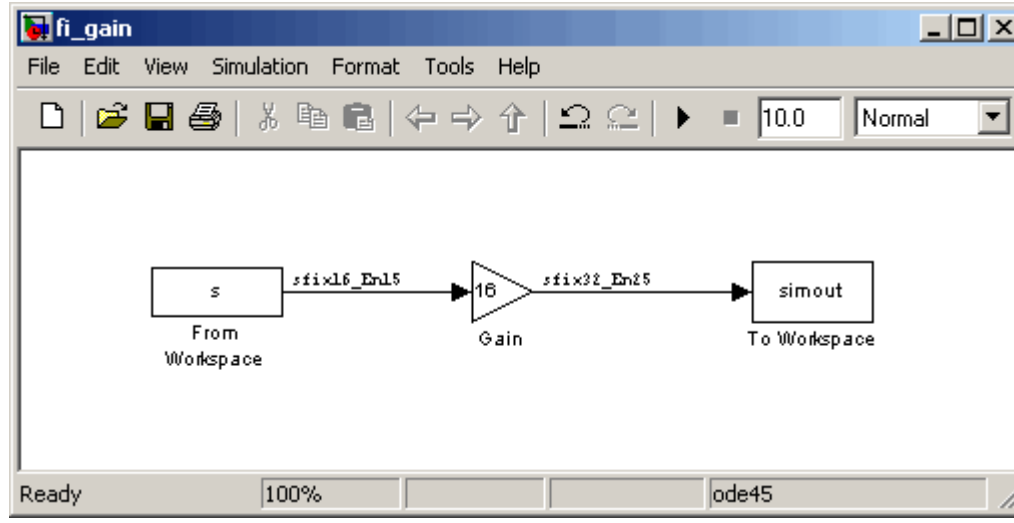
```
a = fi([sin(0:10)' sin(10:-1:0)'])

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 "Logging Signals" in *Using Simulink*.

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

The Simulink software 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 *Using Simulink*.

Additional Features and Capabilities

In addition to the features described in the previous sections, the Simulink Fixed Point software provides you with:

- An automatic scaling tool
- Code generation capabilities

Automatic Scaling

You can 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.

As an alternative to, and extension of, the automatic scaling script, you can use the Fixed-Point Tool. This interface allows you to easily control the parameters associated with automatic scaling and display the simulation results for a given model. To learn how to use the Fixed-Point Tool, refer to Chapter 6, “Fixed-Point Tool”.

Code Generation

With the Real-Time Workshop® 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 8, “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" on page 1-37

"Block Descriptions" on page 1-38

"Simulation Results" on page 1-38

About This Example

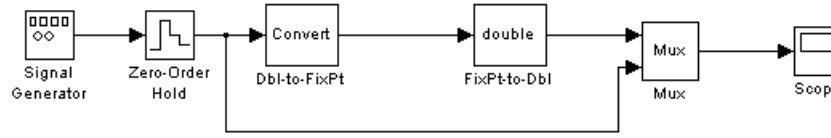
The purpose of this example is to show you how to simulate a continuous real-world doubles signal using a generalized fixed-point data type. Although simple in design, the model gives you an opportunity to explore many of the important features of the Simulink® Fixed Point™ software, including

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

The model used in this example is given by the `fxpdemo_dbl2fix` demo. You can launch this demo by typing its name at the MATLAB® command line:

```
fxpdemo_dbl2fix
```

The model is shown in the following figure.



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

Block Descriptions

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

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

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

Simulation Results

The results of two simulation trials are given below. The first trial uses binary-point-only scaling while the second trial uses [Slope Bias] scaling.

Trial 1: 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 set to 1 and no bias is required.

The Data Type Conversion (Dbl-to-FixPt) block is configured in this way:

- **Output data type and scaling**

  The output data type is given by `fixdt(1,5,2)`. This 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. This gives a maximum value of 011.11 = 3.75, a minimum value of 100.00 = -4.00, and a precision of $(1/2)^2$ = 0.25.

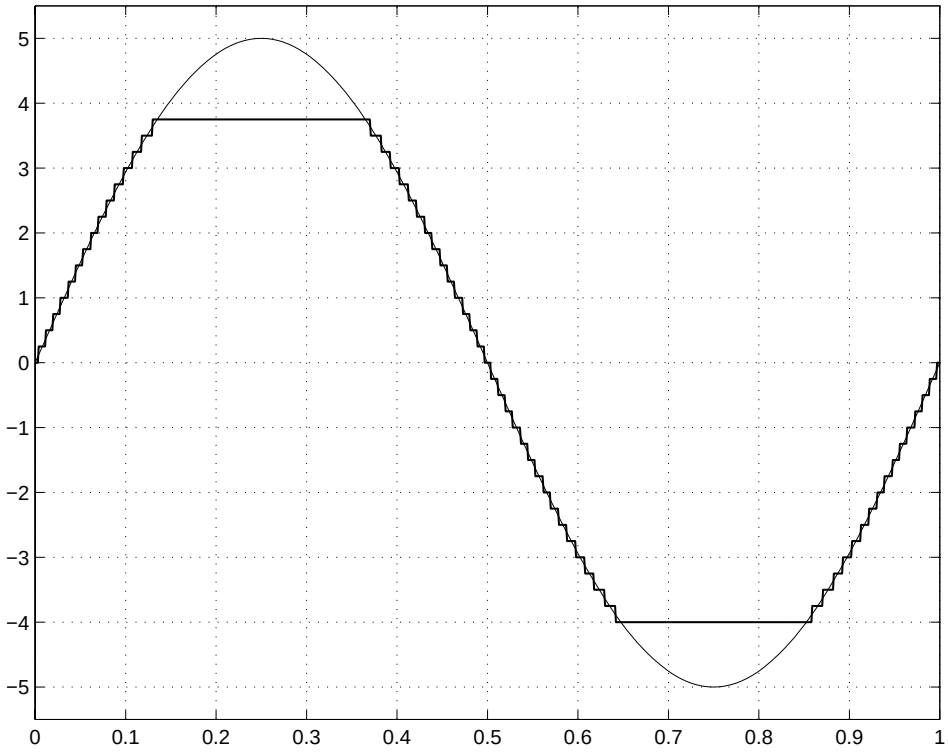
- **Rounding**

  The rounding mode is given by `Floor`. This rounds the fixed-point result toward negative infinity.

- **Overflows**

  Fixed-point values that overflow wrap to the appropriate value that is representable by the data type.

The resulting real-world and fixed-point simulation results are shown in the following figure.



The simulation clearly demonstrates the quantization effects of fixed-point arithmetic. The combination of 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 would be given by `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 $(1/2)^1 = 0.5$.

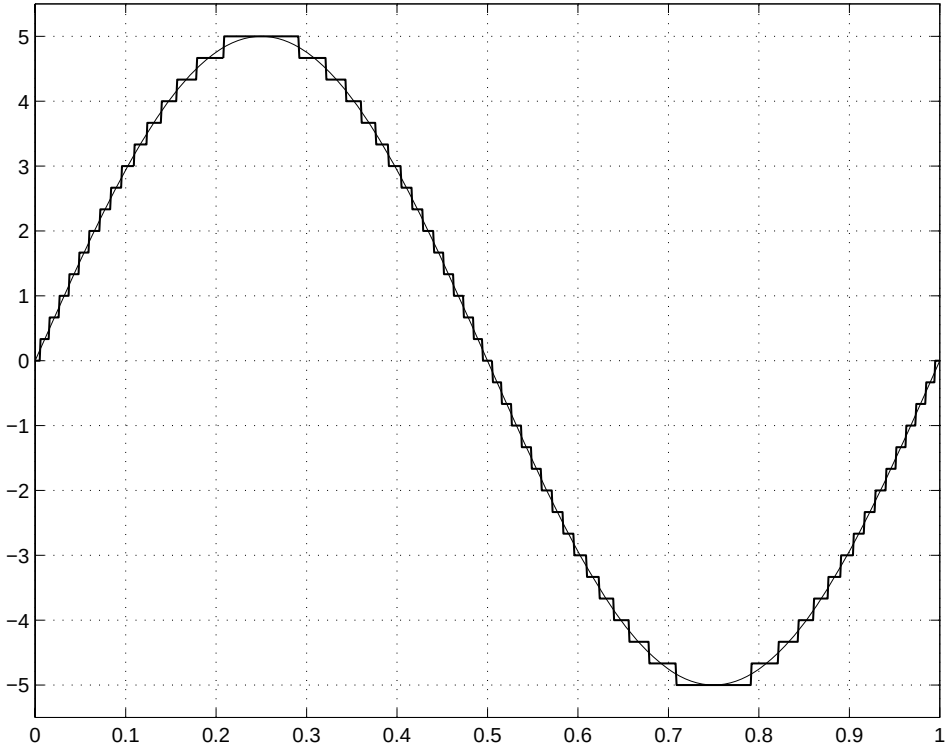
Trial 2: [Slope Bias] Scaling

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. No bias is required for this example because the sine wave is defined on the interval `[-5 5]`. The Data Type Conversion (Dbl-to-FixPt) block configuration is the same as that of the previous trial except for the scaling.

To arrive at a value for the slope, you can begin by assuming a fixed power-of-two exponent of -2. In the previous trial, this value defined the binary-point-only scaling and resulted in a precision of 0.25. 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.

Of course, you could have specified a fixed power-of-two exponent of -1 and a corresponding fractional slope of 0.6667. Naturally, the resulting slope is the same since *E* was reduced by one bit but *F* was increased by one bit. In this case, the blockset would automatically store *F* as 1.3332 and *E* as -2 because of the normalization condition of $1 \le F < 2$.

The resulting real-world and fixed-point simulation results are shown in the following figure.



This somewhat cumbersome process used to find the slope is not really necessary. All that is required is the range of the data you are simulating and the size of the fixed-point word used in the simulation. In general, you can achieve reasonable simulation results by selecting your scaling based on the formula

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

where

- *max* is the maximum value to be simulated.
- *min* 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 (p. 2-2)	Provides an overview of data types and scaling in digital hardware
Fixed-Point Numbers (p. 2-3)	Discusses the representation and manipulation of fixed-point numbers, both in general and in the Simulink® Fixed Point™ software
Floating-Point Numbers (p. 2-17)	Discusses the representation and manipulation of floating-point numbers

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 Real-Time Workshop® product, you can generate production code based on that model.

Fixed-Point Numbers

In this section...

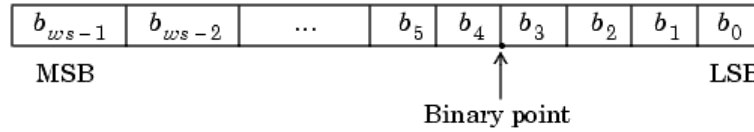
"About Fixed-Point Numbers" on page 2-3
"Signed Fixed-Point Numbers" on page 2-4
"Binary Point Interpretation" on page 2-4
"Scaling" on page 2-5
"Quantization" on page 2-7
"Range and Precision" on page 2-9
"Constant Scaling for Best Precision" on page 2-12
"Fixed-Point Data Type and Scaling Notation" on page 2-15

About Fixed-Point Numbers

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

Note Fixed-point word sizes up to 128 bits are supported.

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).
- The size of the word in bits is given by ws.
- 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 is 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.

Within the Simulink Fixed Point software, the main difference between fixed-point data types is the default binary point. For integers and fractionals, the binary point is fixed at the default value. For generalized fixed-point data types, you must either explicitly specify the scaling by configuring dialog box parameters, or inherit the scaling from another block. The sections that follow describe the supported fixed-point data types.

Integers

The default binary point for signed and unsigned integer data types is assumed to be just to the right of the LSB. You specify unsigned and signed integers with the `uint` and `sint` functions, respectively.

Fractionals

The default binary point for unsigned fractional data types is just to the left of the MSB, while for signed fractionals the binary point is just to the right of the MSB. If you specify guard bits, then they lie to the left of the binary point. You specify unsigned and signed fractional numbers with the `ufrac` and `sfrac` functions, respectively.

Generalized Fixed-Point Numbers

For signed and unsigned generalized fixed-point numbers, there is no default binary point. You specify unsigned and signed generalized fixed-point numbers with the `ufix` and `sfix` functions, respectively.

Note You can also use the `fixdt` function to create integer, fractional, and generalized fixed-point objects.

Scaling

The dynamic range of fixed-point numbers is much less than that of 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 default binary point, or you can select a generalized fixed-point data type and choose an arbitrary linear scaling that suits your needs. This section presents the scaling choices available for generalized fixed-point data types.

A fixed-point number can be represented by a general [Slope 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 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 fractional slope. 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 3-23.

Binary-Point-Only Scaling

As the name implies, binary-point-only (or power-of-two) scaling involves moving only the binary point within the generalized fixed-point word. The advantage of this scaling mode is that the number of processor arithmetic operations is minimized.

With binary-point-only scaling, the components of the general [Slope Bias] formula have these values:

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

That is, the scaling of the quantized real-world number is defined only by the slope S, which is restricted to a power of two.

In the Simulink Fixed Point software, you specify binary-point-only scaling with the syntax `2^-E` where `E` is unrestricted. This creates a MATLAB® structure with a bias $B = 0$ and a fractional slope $F = 1.0$. For example, the syntax `2^-10` defines a scaling such that the binary point is at a location 10 places to the left of the least significant bit.

[Slope 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. You specify scaling by slope and bias with the syntax `[slope bias]`, which creates a MATLAB structure with the given slope and bias. For example, a [Slope Bias] scaling specified by `[5/9 10]` defines a slope of 5/9 and a bias of 10. The slope must be a positive number.

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 Bias] encoding scheme, the value of an unsigned fixed-point quantity is given by

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

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

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

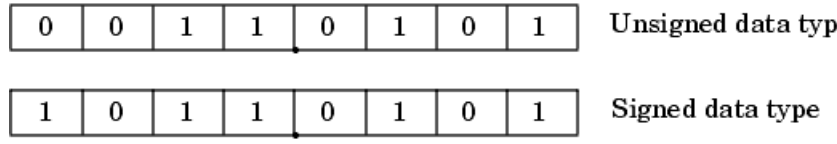
where

- b_i are binary digits, with b_i = 1, 0.
- The word size in bits is given by *ws*, with *ws* = 1,2,3,...,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

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



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

$$\tilde{V} = (F2^E) \cdot Q = 2^E \cdot \left[\sum_{i=0}^{ws-1} b_i 2^i \right]$$

$$= 2^{-4} \cdot (0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)$$

$$= 3.3125$$

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

$$\tilde{V} = (F2^E) \cdot Q = 2^E \cdot \left[\sum_{i=0}^{ws-1} b_i 2^i \right]$$

$$= 2^{-4} \cdot (1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)$$

$$= 11.3125$$

while the two's complement value is

$$\tilde{V} = (F2^E) \cdot Q = 2^E \cdot \left[-b_{ws-1} 2^{ws-1} + \sum_{i=0}^{ws-2} b_i 2^i \right]$$

$$= 2^{-4} \cdot (-1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)$$

$$= -4.6875$$

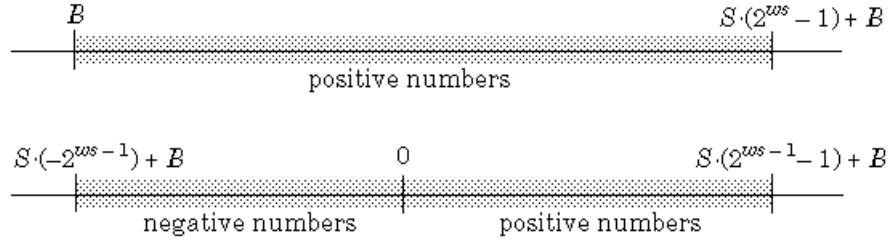
Range and Precision

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 depend 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 (scaling) of integer and fractional data types is specified by the default binary point. For generalized fixed-point data types, the scaling must be explicitly defined as either [Slope Bias] or binary-point-only. In either case, the precision is given by the slope.

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 3-4 and “Limitations on Range” on page 3-17 for more information.

Fixed-Point Data Type Range and Default Scaling

Name	Data Type	Low Limit	High Limit	Default Scaling (~Precision)
Integer	`uint`	0	$2^{ws} - 1$	1
	`sint`	$-2^{ws - 1}$	$2^{ws - 1} - 1$	1
Fractional	`ufrac`	0	$1 - 2^{-ws}$	2^{-ws}
	`sfrac`	-1	$1 - 2^{-(ws - 1)}$	$2^{-(ws - 1)}$
Generalized Fixed-Point	`ufix`	N/A	N/A	N/A
	`sfix`	N/A	N/A	N/A

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

The precision, range of signed values, and range of unsigned values for an 8-bit generalized fixed-point data type with binary-point-only scaling follow. 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

Scaling	Precision	Range of Signed Values (Low, High)	Range of Unsigned Values (Low, High)
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 Bias] Scaling

The precision and range of signed and unsigned values for an 8-bit fixed-point data type using [Slope Bias] scaling follow. The slope starts at a value of 1.25 and the bias is 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 generalized fixed-point data types. All other fixed-point data types use their default scaling. You can use the **Data Type Assistant** (see "Using the Data Type Assistant" in *Using Simulink*) on a block's dialog box to enable the constant scaling mode:

1. Click the **Show data type assistant** button on a block dialog box.

   The **Data Type Assistant** appears.

2. In the **Data Type Assistant**, set the **Mode** field to `Fixed point`.

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

3. Set the **Scaling** option to `Best precision`.

To understand how you might use this scaling mode, consider a 5-by-4 matrix of doubles, M, defined as

```
3.3333e-005  3.3333e-006  3.3333e-007  3.3333e-008
3.3333e-004  3.3333e-005  3.3333e-006  3.3333e-007
3.3333e-003  3.3333e-004  3.3333e-005  3.3333e-006
3.3333e-002  3.3333e-003  3.3333e-004  3.3333e-005
3.3333e-001  3.3333e-002  3.3333e-003  3.3333e-004
```

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 rightmost bit). With this data format, M becomes

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

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

The disadvantage of constant scaling is reduced precision resulting from the use of a common binary point. However, the potential advantages of using a common binary point include reduced code size and increased processor speed.

Fixed-Point Data Type and Scaling Notation

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

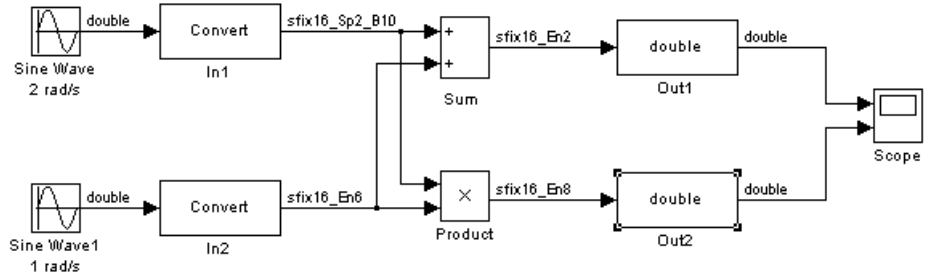
Symbol	Description
`uint`	Unsigned integer fixed-point data type
`sint`	Signed integer fixed-point data type
`ufrac`	Unsigned fraction fixed-point data type
`sfrac`	Signed fraction fixed-point data type
`ufix`	Unsigned generalized fixed-point data type
`sfix`	Signed generalized fixed-point data type
`fltu`	Doubles override of an unsigned fixed-point data type
`flts`	Doubles override of a signed fixed-point data type
`B`	Bias
`E`	2^
`e`	10^
`F`	Fractional slope
`n`	Negative
`p`	Decimal point
`S`	Slope

Example: Port Data Type Display

For example, to display the data types of ports in your model, select **Port/Signal Displays** > **Port Data Types** from the Simulink **Format** menu.

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's **Output data type** parameter value or the data type and scaling that is inherited from the driving block or through backpropagation.

The following model displays its port data types:



The data type display associated with the `In1` block in the model indicates that the output data type is `sfix(16)` (a signed, 16-bit, generalized fixed-point number) with [Slope Bias] scaling of `[0.2 10]`. Note that this scaling is not the block's default scaling. The data type display associated with the `In2` block indicates that the output data type is `sfix(16)` with binary-point-only scaling of `2^-6`.

Floating-Point Numbers

In this section...
"About Floating-Point Numbers" on page 2-17
"Scientific Notation" on page 2-17
"The IEEE® Format" on page 2-19
"Range and Precision" on page 2-22
"Exceptional Arithmetic" on page 2-24

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 $\pm f \times 2^{\pm e}$ 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 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 q is an integer of unrestricted range. The previous equation is valid for both fixed- and floating-point numbers. For both these data types, the fraction can be changed at any time by the processor. However, for fixed- point numbers the exponent never changes, while for floating-point numbers the exponent can be changed any time by the processor.

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

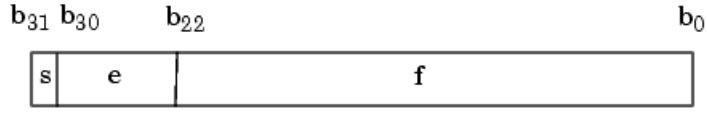
denormalized numbers, which have a fraction of the form 0.*f*. Normalized and denormalized formats are discussed in more detail in the next section.

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 2-24 and "NaN" on page 2-25.

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*. A representation of this format is given below.



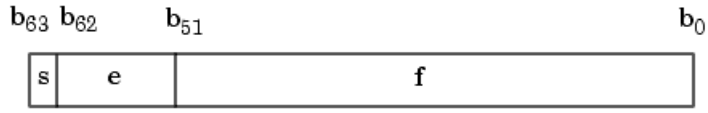
The relationship between this format and the representation of real numbers is given by

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

"Exceptional Arithmetic" on page 2-24 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*. 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

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

"Exceptional Arithmetic" on page 2-24 discusses denormalized values.

Nonstandard IEEE® Format

The Simulink Fixed Point software supports a nonstandard IEEE-style floating-point data type. This data type adheres to the definitions and formulas previously given for IEEE singles and doubles. You create nonstandard floating-point numbers with the `float` function:

```
float(TotalBits,ExpBits)
```

`TotalBits` is the total word size and `ExpBits` is the size of the exponent field. The size of the fraction field and the bias are calculated from these input arguments. You can specify any number of exponent bits up to 11, and any number of total bits such that the fraction field is no more than 53 bits.

When specifying a nonstandard format, you should remember that the number of exponent bits largely determines the range of the result and the number of fraction bits largely determines the precision of the result.

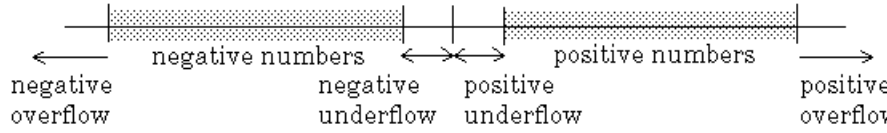
Note These numbers are normalized with a hidden leading one for all exponents except the smallest possible exponent. However, the largest possible exponent might not be treated as a flag for `Inf` or `NaN`.

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 $(2 - 2^{-f}) \cdot 2^{bias}$.
- Normalized negative numbers are defined within the range $-2^{(1 - bias)}$ to $-(2 - 2^{-f}) \cdot 2^{bias}$.
- Positive numbers greater than $(2 - 2^{-f}) \cdot 2^{bias}$ and negative numbers greater than $-(2 - 2^{-f}) \cdot 2^{bias}$ 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,

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

and

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

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 $0.f = 0.0$.

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 $-\infty \le (\text{representable numbers}) < \infty$. 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 (p. 3-3)	Provides an overview of issues that need to be considered when performing fixed-point arithmetic operations—overflow, quantization, computational noise, and limit cycles
Limitations on Precision (p. 3-4)	Discusses the limits placed on the precision of fixed-point calculations, and how they are handled in the Simulink® software
Limitations on Range (p. 3-17)	Discusses the limits placed on the range of fixed-point calculations, and how they are handled in the Simulink software
Recommendations for Arithmetic and Scaling (p. 3-23)	Recommends scaling in your fixed-point design based on the limitations of fixed-point arithmetic
Parameter and Signal Conversions (p. 3-34)	Discusses the way the data types of parameters and signals are converted in Simulink simulations

Rules for Arithmetic Operations (p. 3-39)	Describes the way arithmetic operations are performed on inputs and parameters in the Simulink software
Example: Conversions and Arithmetic Operations (p. 3-56)	Provides an example highlighting the way the data types are converted and arithmetic operations are performed on inputs and parameters in the Simulink software

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" on page 3-4

"Rounding" on page 3-4

"Padding with Trailing Zeros" on page 3-14

"Example: Limitations on Precision and Errors" on page 3-15

"Example: Maximizing Precision" on page 3-15

Introduction

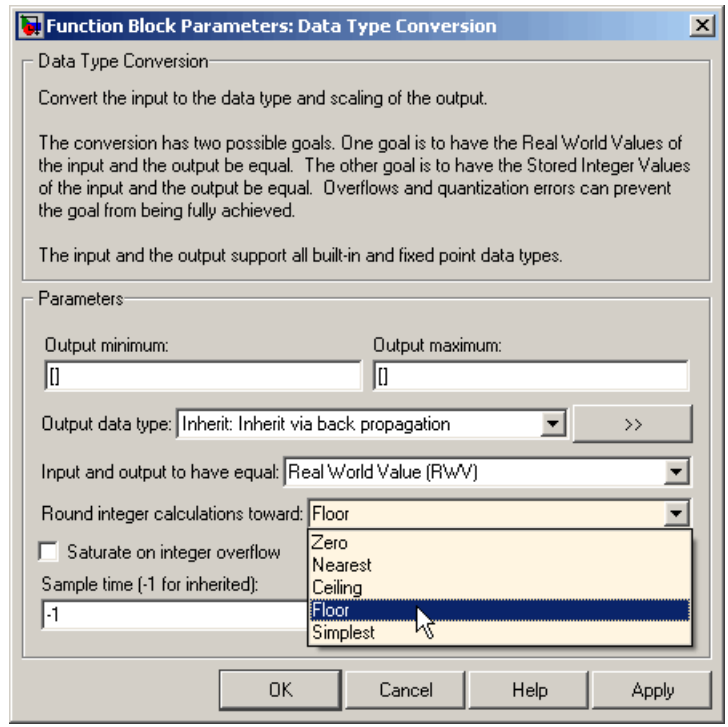
Computer words consist of a finite numbers of bits. This means that the binary encoding of variables is only an approximation of an arbitrarily precise real-world value. Therefore, the limitations of the binary representation automatically introduce limitations on the precision of the value. For a general 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.

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

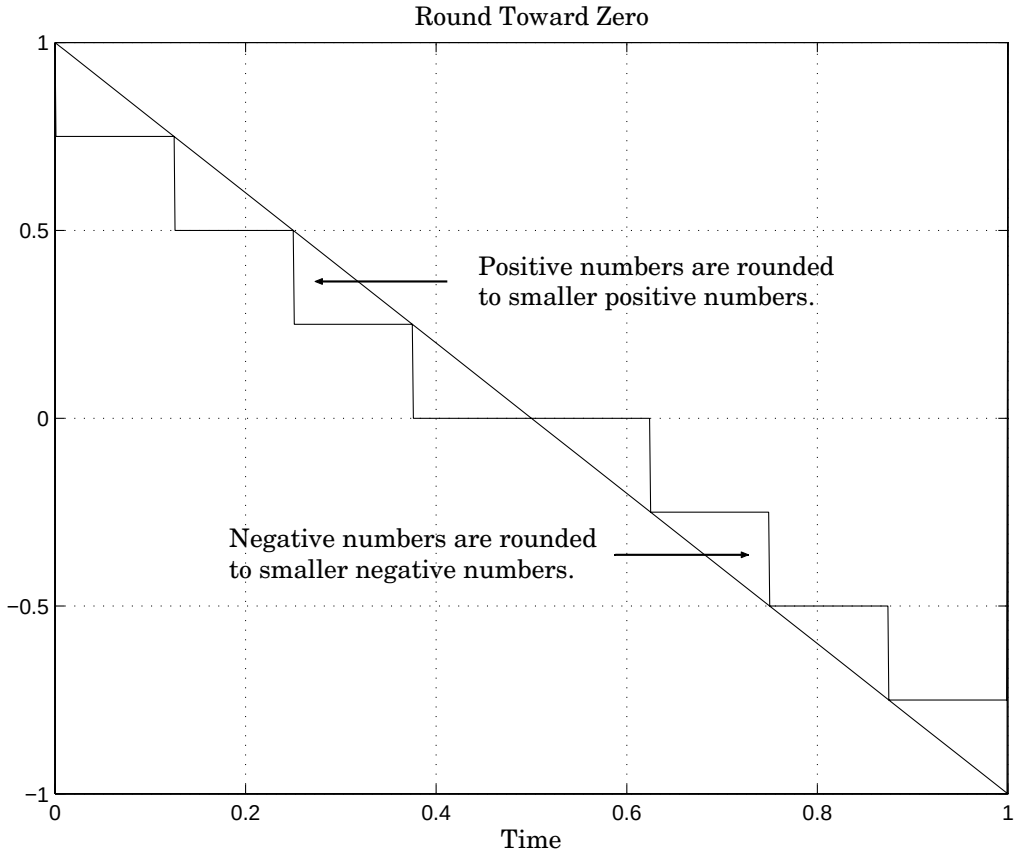


These rounding modes are discussed in the sections that follow.

Round Toward Zero

The simplest rounding mode computationally is when all digits beyond the number required are dropped. This mode is referred to as rounding toward zero, and it 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.



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.

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.

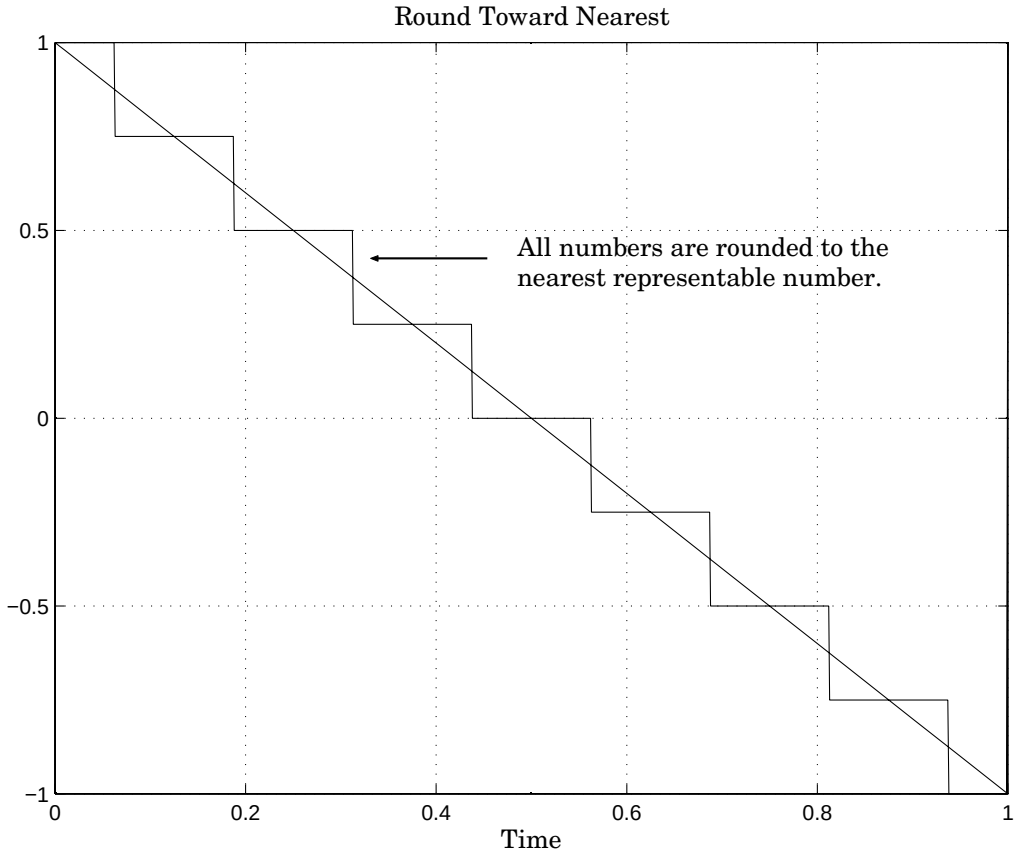
Now consider rounding a 5-bit two's complement number by dropping the two least significant bits. At first glance, you may think truncating a two's complement number is the same as rounding to zero. For example, dropping the last two digits of -3.75 yields -3.00. However, digital hardware performing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

As you can see, rounding to zero for a two's complement number is not the same as truncation when the original value is negative. For this reason, the ambiguous term "truncation" is not used in this guide, and four explicit rounding modes are used instead.

Round Toward Nearest

When you round toward nearest, the number is rounded to the nearest representable value. This mode has the smallest errors associated with it and these errors are symmetric. As a result, rounding toward nearest is the most useful approach for most applications.

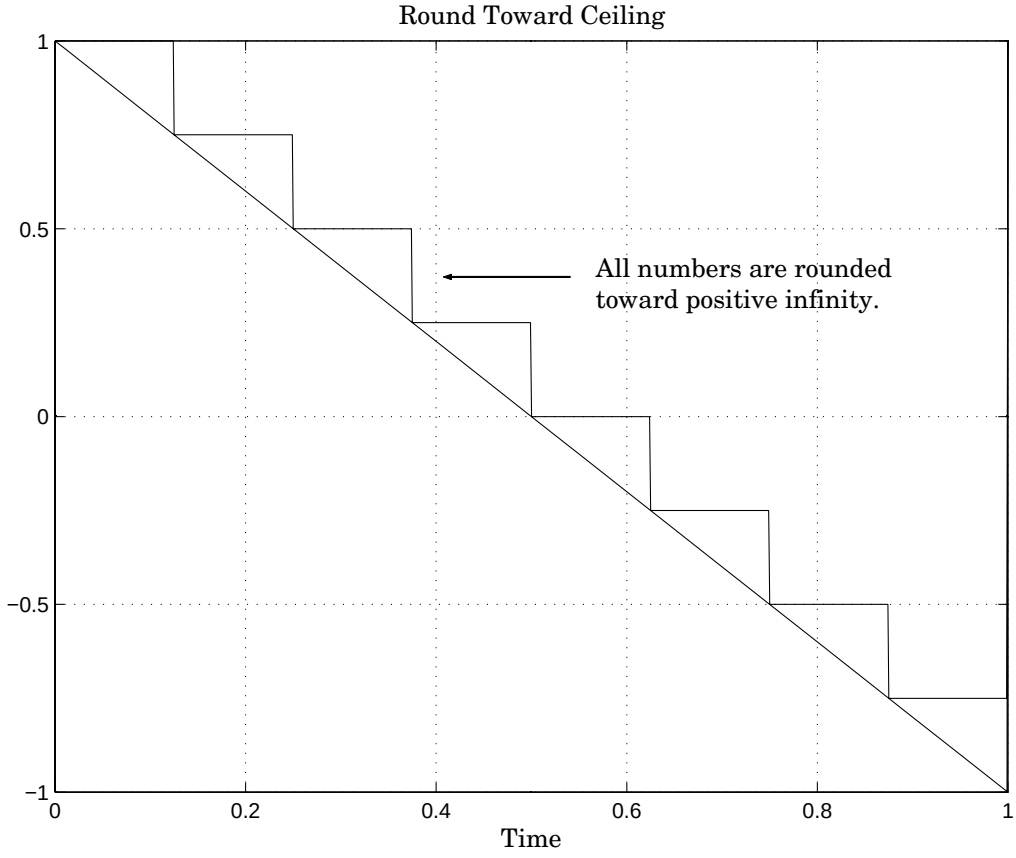
In the Fixed-Point Toolbox™ software, you can round to nearest using the `nearest` function. Rounding toward nearest is shown in the following figure.


Round Toward Nearest
All numbers are rounded to the
nearest representable number.
1
0.5
0
−0.5
−1
0
0.2
0.4
0.6
0.8
1
Time

Round Toward 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.

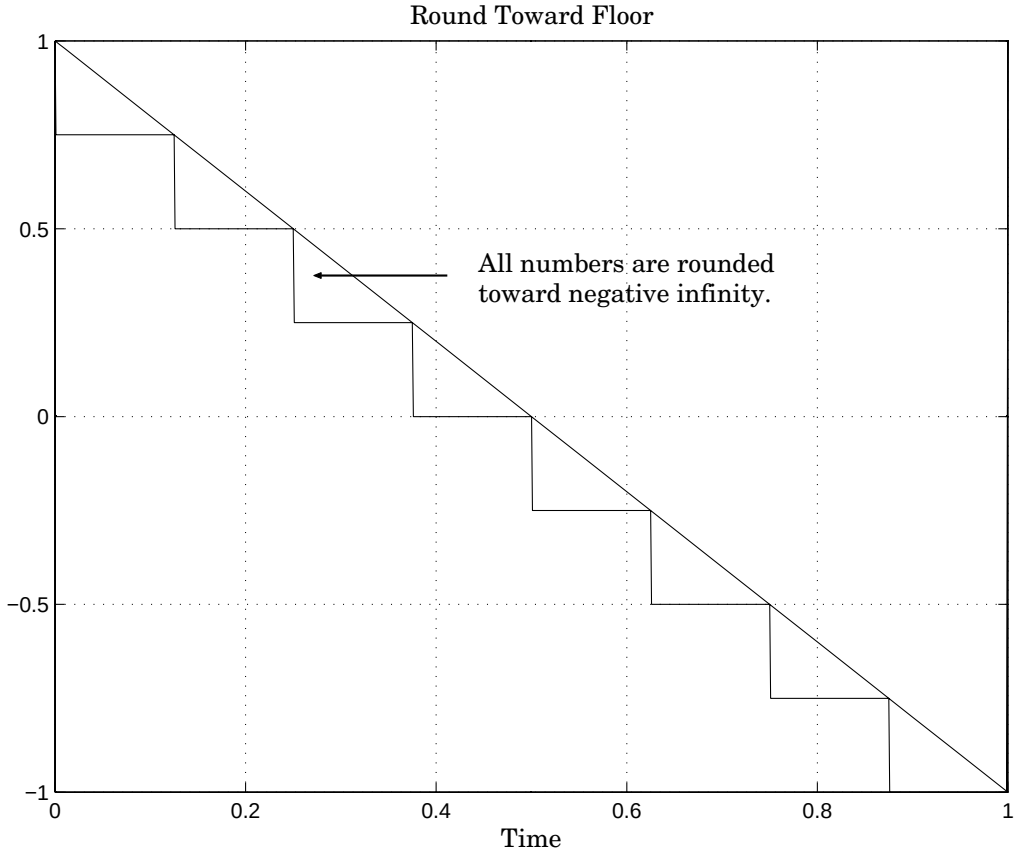
In the MATLAB software, you can round to ceiling using the `ceil` function. Rounding toward ceiling is shown in the following figure.



Round Toward 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 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 required.

Simplest Rounding

The `Simplest` rounding mode is currently available for the following blocks:

- Data Type Conversion
- Divide, Product, Product of Elements
- Interpolation Using Prelookup
- Lookup Table
- Lookup Table (2-D)
- Lookup Table Dynamic
- Prelookup

This mode attempts to reduce or eliminate the need for extra rounding code in your generated code for these blocks. It does this in one or more of three ways for each block, discussed in the following sections:

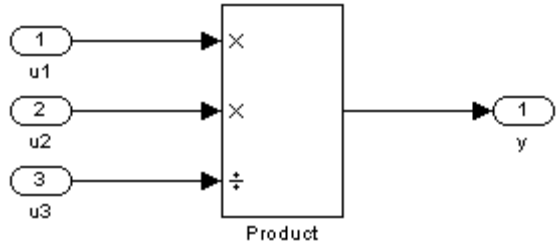
- "Optimize Rounding for Casts" on page 3-12
- "Optimize Rounding for High-Level Arithmetic Operations" on page 3-12
- "Optimize Rounding for Intermediate Arithmetic Operations" on page 3-13

Note In many cases, in order for the `Simplest` rounding mode to produce the most efficient generated code, you must specify the **Signed integer division rounds to** parameter on the **Hardware Implementation** pane of the Configuration Parameters dialog box with the correct information for your target hardware.

Optimize Rounding for Casts. The Data Type Conversion block casts a signal with one data type to another data type. When the signal is cast 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 the cast is from one integer or fixed-point data type to another, the `Simplest` mode rounds toward floor.
- When the cast is 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 in every case is 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 hardware.

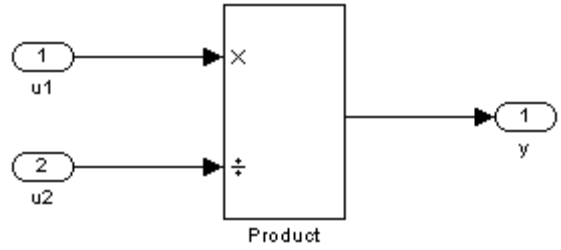
The `Simplest` rounding mode

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

To get the most efficient code, you must specify whether your hardware rounds to zero or to floor for integer divide in the **Signed integer division rounds to** parameter. It is very common for a hardware target to round to zero for integer division operations. Note that the `Simplest` mode enables "mixed-mode" rounding for such cases, as it will round to floor for multiplies and to zero for divides.

If the **Signed integer division rounds to** parameter is set to `Undefined`, the simplest rounding mode may not be able to produce the most efficient code. The `Simplest` mode will round to zero for division for this case, however it cannot rely on your target hardware to perform the rounding since the parameter is `Undefined`. Therefore, extra code has to be added to ensure rounding to zero behavior.

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 there are also intermediate additions and multiplies that are performed:

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

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 3-12. 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 you must specify the **Signed integer division rounds to** parameter on the **Hardware Implementation** pane of the Configuration Parameters dialog box with the correct information for your hardware to generate the most efficient code using the `Simplest` rounding mode.

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:

$$1.0000000 \cdot 2^q - 1.1111111 \cdot 2^{q-1}$$

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

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

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

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

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 6-15.

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

$$\tilde{V} = 2^{-2}Q + 32$$

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:

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

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

$$max(V) = F2^{E}(max(Q)) + B$$
$$min(V) = F2^{E}(min(Q)) + B$$

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.

Limitations on Range

In this section...

"Introduction" on page 3-17

"Saturation and Wrapping" on page 3-18

"Guard Bits" on page 3-21

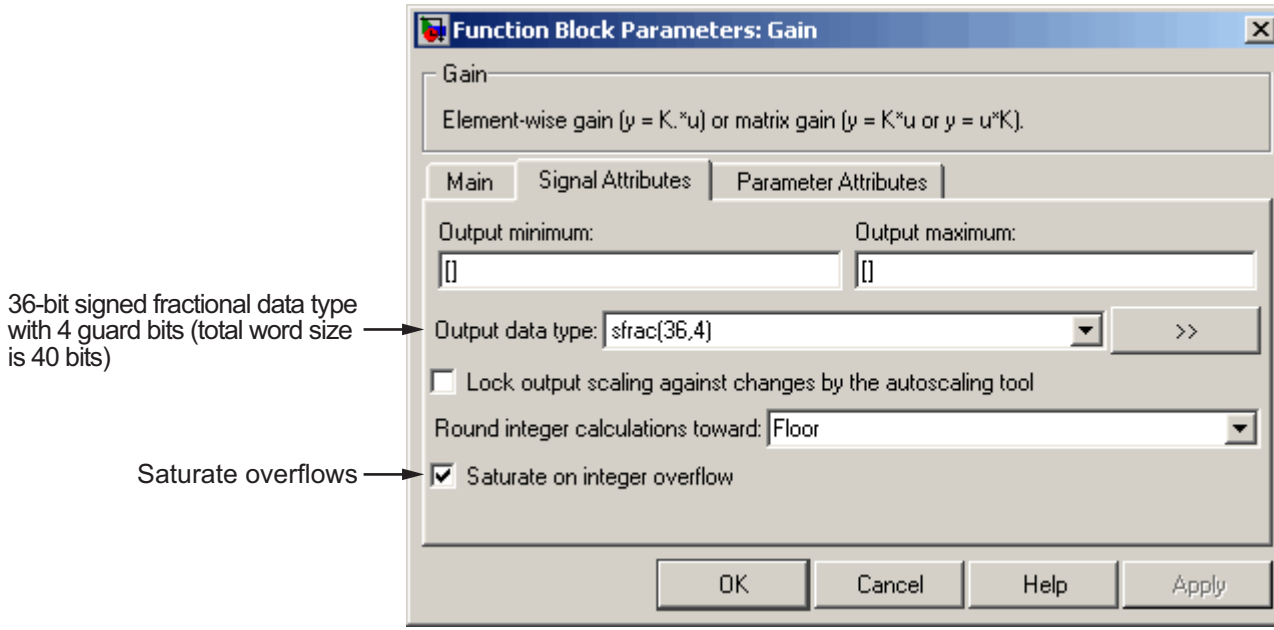
"Example: Limitations on Range" on page 3-21

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 $\mathbf{m \neq n}$. 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.

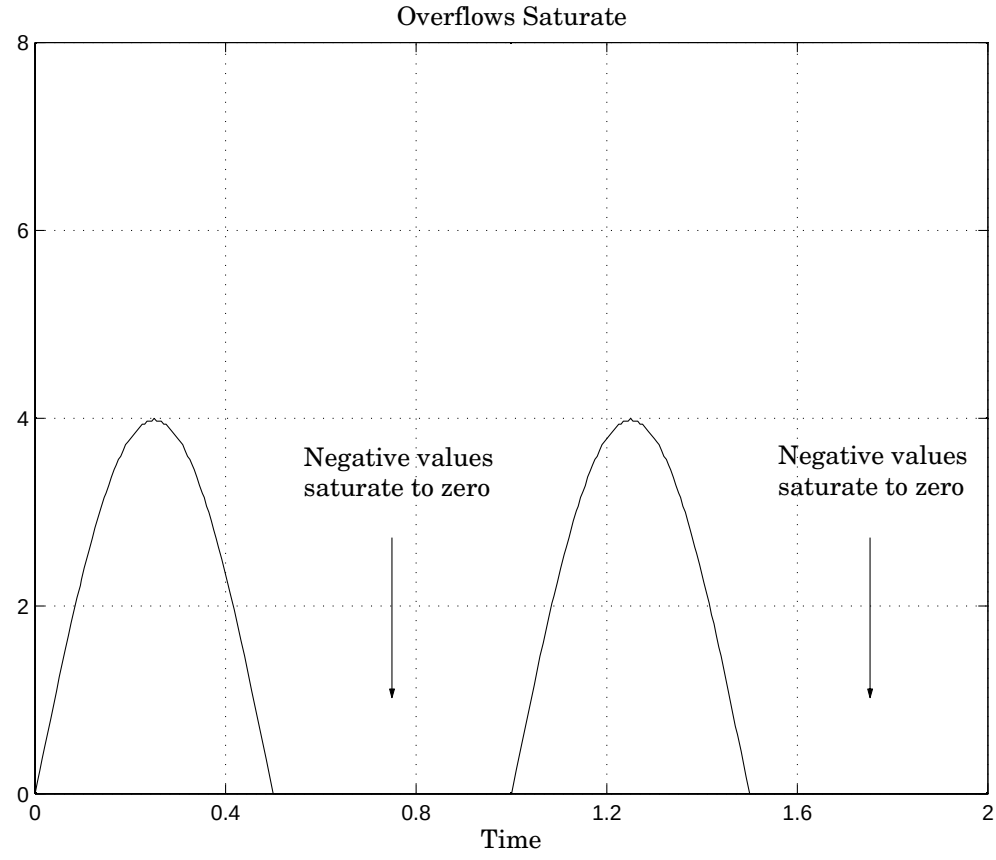


Saturation and Wrapping

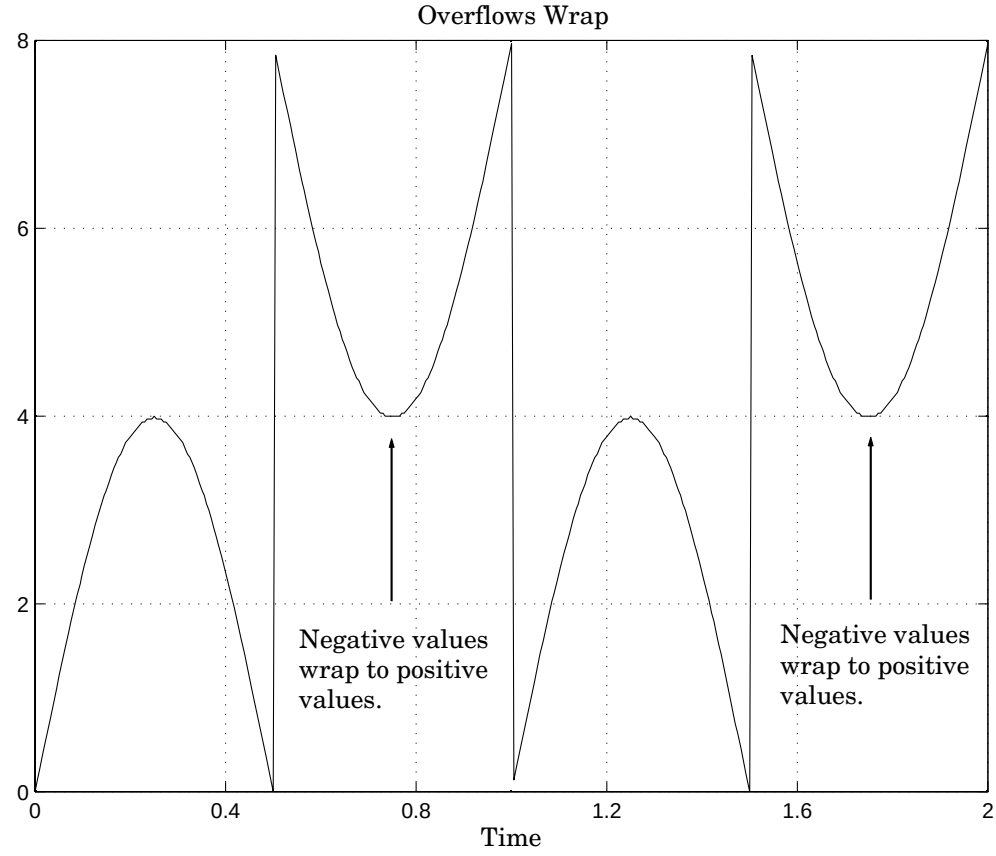
Saturation and wrapping describe a particular way that some processors deal with overflow conditions. For example, the ADSP-2100 family of processors from Analog Devices™ supports either of these modes. If a register has a saturation mode of operation, then an overflow condition is set to the maximum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

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

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

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" on page 3-23
"Addition" on page 3-24
"Accumulation" on page 3-27
"Multiplication" on page 3-27
"Gain" on page 3-29
"Division" on page 3-31
"Summary" on page 3-33

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:

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

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

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

or

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

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 3-15 shows that you can determine the range of a fixed-point operation from $max(V_a)$ and $min(\tilde{V}_a)$. For a summation, you can determine the range from

$$min(\tilde{V}_a) = min(\tilde{V}_b) + min(\tilde{V}_c)$$
$$max(\tilde{V}_a) = max(\tilde{V}_b) + max(\tilde{V}_c)$$

You can now derive the maximum-precision slope:

$$F_a 2^{E_a} = \frac{max(\tilde{V}_a) - min(\tilde{V}_a)}{2^{ws_a} - 1}$$
$$= \frac{F_b 2^{E_b}(2^{ws_b} - 1) + F_c 2^{E_c}(2^{ws_c} - 1)}{2^{ws_a} - 1}$$

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

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

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

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

$$B_a \approx B_b + B_c$$

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:

$$Q_{a_new} = Q_{a_old} + \frac{F_b}{F_a} \cdot 2^{E_b - E_a} Q_b + \frac{B_b}{F_a} \cdot 2^{-E_a}$$

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:

$$V_a = V_b \times V_c$$

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:

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

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

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

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 3-15 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

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

where

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

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

$$V_a = K \cdot V_b$$

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:

$$Q_a = \left(\frac{KF_b 2^{E_b}}{F_a 2^{E_a}}\right) \cdot Q_b + \left(\frac{KB_b - B_a}{F_a 2^{E_a}}\right)$$

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:

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

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

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

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 3-15 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:

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

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" on page 3-34

"Parameter Conversions" on page 3-35

"Signal Conversions" on page 3-36

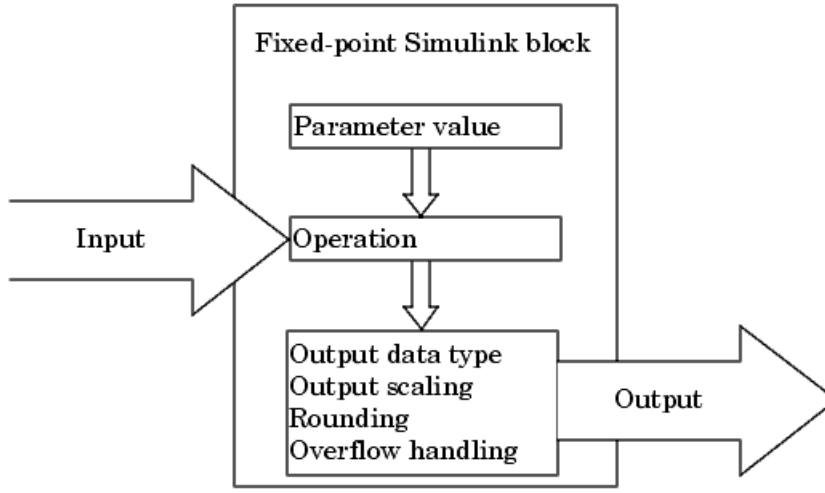
Introduction

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 3-39 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

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

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:

$$Q_a = \frac{F_b}{F_a} 2^{E_b - E_a} Q_b + \frac{B_b - B_a}{F_a} 2^{-E_a}$$

$$= F_s 2^{E_b - E_a} Q_b + B_{net}$$

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" on page 3-39

"Computational Units" on page 3-39

"Addition and Subtraction" on page 3-40

"Multiplication" on page 3-45

"Division" on page 3-52

"Shifts" on page 3-54

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_a 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 3-24 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:

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

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

4. The summation operation is performed:

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

   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.

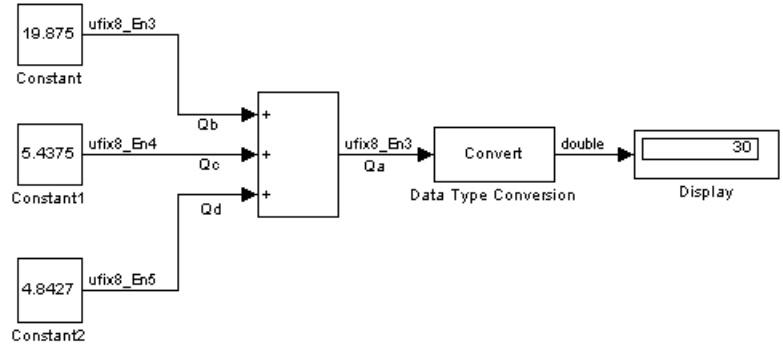
Streamlining Simulations and Generated Code

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:

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

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

$$V_a = V_b \times V_c$$

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:

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

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 efficiency and the possibility of 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:

$$V_{1Temp} = V_1$$

$$V_{2Temp} = V_2$$

$$V_{3\,Temp} = V_{1Temp} \times V_{2Temp}$$

$$V_3 = V_{3\,Temp}$$

where

$$V_{1Temp} = 2^{E_{1Temp}} \times Q_{1Temp}$$

$$V_{2Temp} = 2^{E_{2Temp}} \times Q_{2Temp}$$

$$V_{3Temp} = 2^{E_{3Temp}} \times Q_{3Temp}$$

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:

$$V_{Max} = F \times 2^E \times Q_{MaxInteger} \times B$$

$$V_{Min} = F \times 2^E \times Q_{MinInteger} \times B$$

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,

$$Q_{MaxInteger} = 255$$

$$Q_{MinInteger} = 0$$

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

uint8_T $Q_{Original}$

uint16_T Q_{Temp}

$$Q_{Temp} = ((\text{int16_T})Q_{Original} \ll 2) - 173$$

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

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

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 $Q_{\text{RawProduct}}$ 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 3-36 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 3-36 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 $Q_{\text{RawProduct}}$ 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:

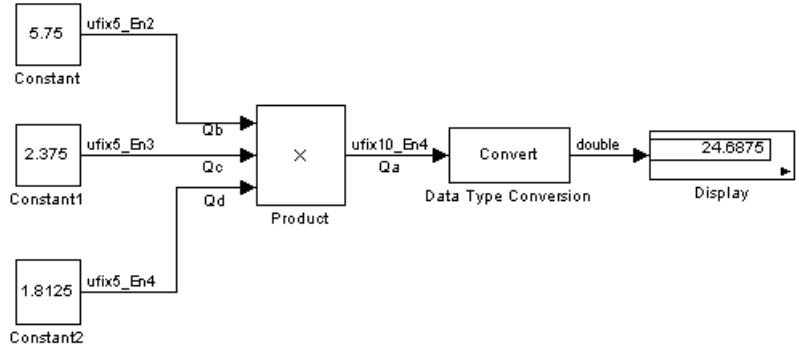
   $$Q_a = convert(Q_{RawProduct})$$

   This conversion includes any necessary bit shifting, rounding, or overflow handling. “Signal Conversions” on page 3-36 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:

$$
\begin{aligned}
Q_{RawProduct} = \quad & 101.11 \\
& \times 10.011 \\
\hline
& 101.11 \cdot 2^{-3} \\
& 101.11 \cdot 2^{-2} \\
& + 101.11 \cdot 2^{1} \\
\hline
& 01101.10101 = 13.65625
\end{aligned}
$$

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:

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

"Signal Conversions" on page 3-36 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:

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

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:

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

$$= 011000.1011 = 24.6875$$

"Signal Conversions" on page 3-36 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 3-45.

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 slopes 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:

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

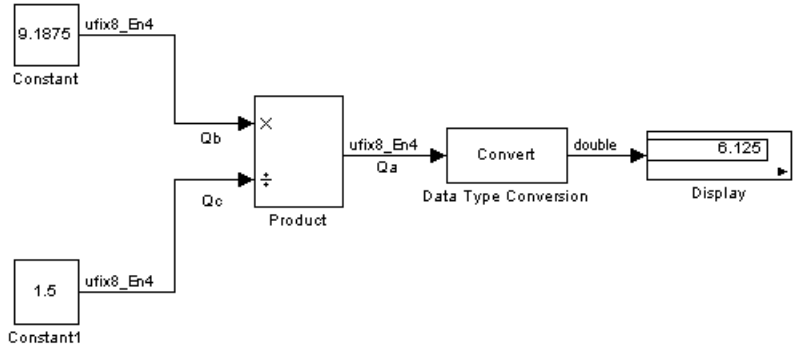
This equation involves an integer division and some bit shifts. If $E_a \geq E_b - E_c$, then any bit shifts are to the right and the implementation is simple. However, if $E_a < E_b - E_c$, then the bit shifts are to the left and the implementation can be more complicated. The essential issue is that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

$$Q_a = 2^{-4-(-4)-(-4)}(Q_b/Q_c)$$
$$= 2^4(Q_b/Q_c)$$

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

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

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

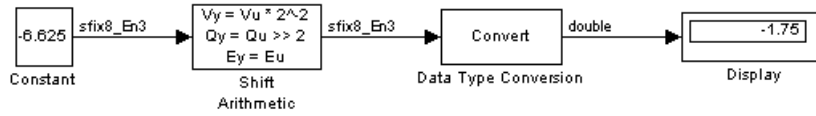
Shift Operation	Binary Value	Decimal Value
Shift left by 2 bits	11010100	212
Shift right by 2 bits	00001101	13

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.

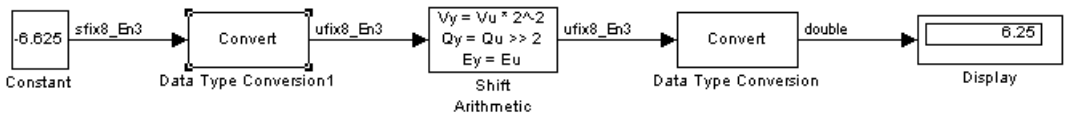
Shifting Bits to the Right

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).



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 3-34 and "Rules for Arithmetic Operations" on page 3-39.

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

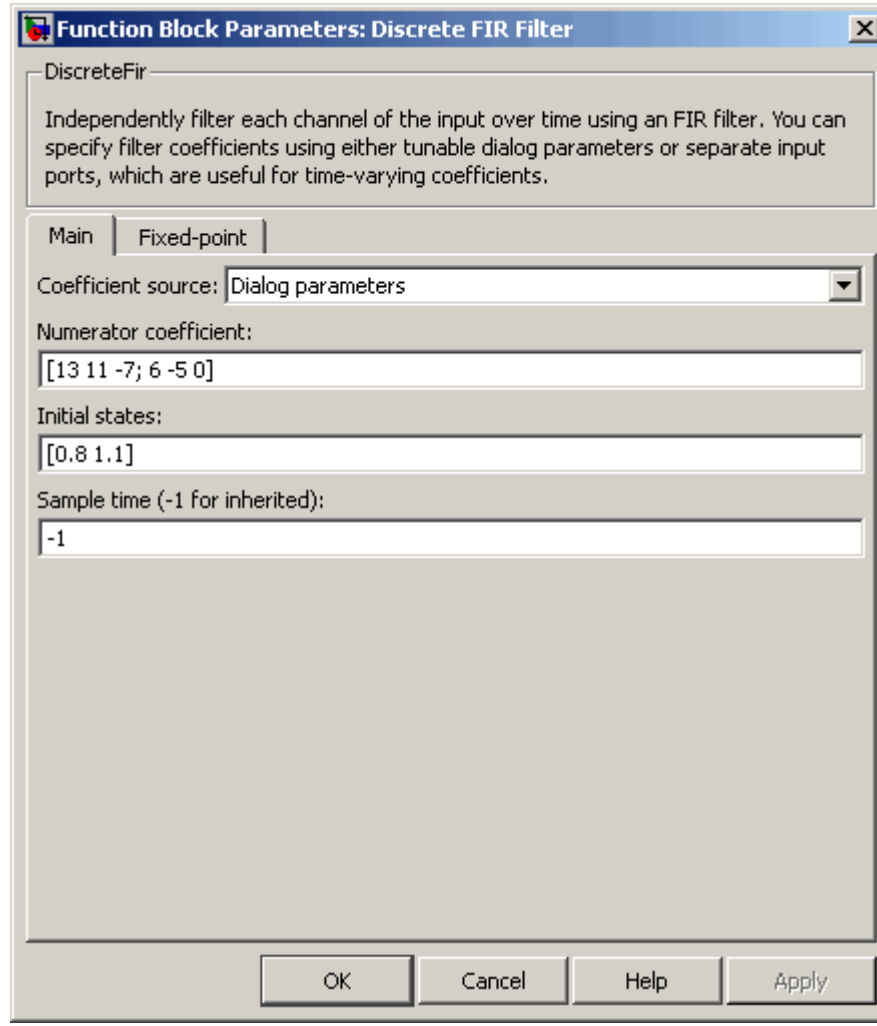
$$y_1(k) = 13 \cdot u(k) + 11 \cdot u(k-1) - 7 \cdot u(k-2)$$

and the second output is given by

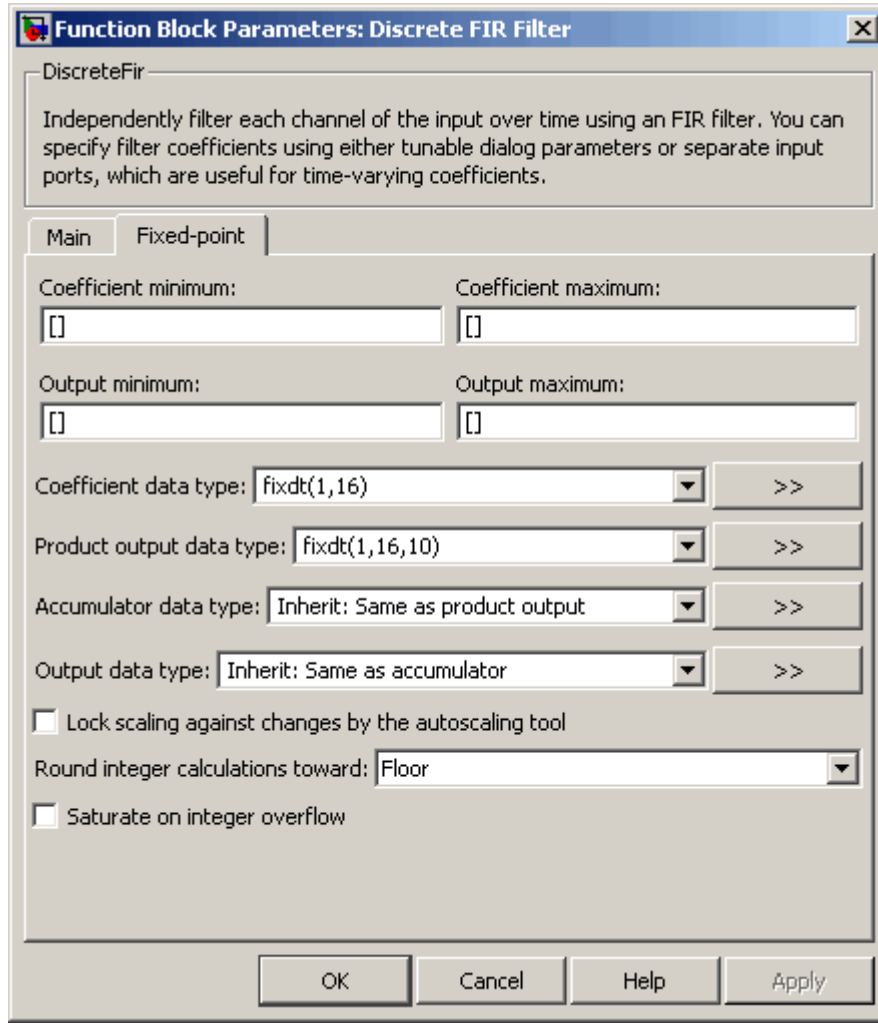
$$y_2(k) = 6 \cdot u(k) - 5 \cdot u(k-1)$$

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 **Numerator coefficient** 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 **Fixed-point** pane of the block's dialog box to appear as follows:



Parameter conversions and block operations are given below in the order in which they are carried out by the Discrete FIR Filter block:

1 The **Numerator coefficient** parameter is converted offline from doubles to the **Coefficient data type** value 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 then converted to the **Output data type** value using the specified rounding and overflow modes.

3 The sum is carried out for $y_1(0)$ and $y_2(0)$. Note that the rules for addition and subtraction are satisfied, because the coefficients and inputs are already converted to the **Output data type** value.

4 Steps 2 and 3 are repeated for subsequent time steps.

4

Realization Structures

Overview (p. 4-2)	Provides a brief overview of creating filters using fixed-point Simulink® blocks
Targeting an Embedded Processor (p. 4-4)	Describes issues that arise when targeting a fixed-point design for use on an embedded processor
Canonical Forms (p. 4-7)	Discusses some canonical forms that optimize filter implementation with respect to certain factors

Overview

In this section...
"Introduction" on page 4-2
"Realizations and Data Types" on page 4-2

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 Signal Processing Blockset™ 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" on page 4-4

"Size Assumptions" on page 4-4

"Operation Assumptions" on page 4-4

"Design Rules" on page 4-5

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" on page 4-7
"Direct Form II" on page 4-8
"Series Cascade Form" on page 4-11
"Parallel Form" on page 4-14

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 1-19, 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 Real-Time Workshop® 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:

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

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

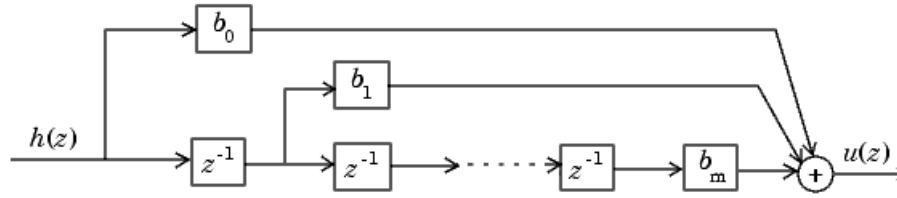
Canonical Form" in *Digital Control of Dynamic Systems* by Franklin, Powell, and Workman.

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

$$\frac{u(z)}{e(z)} = \frac{u(z)}{h(z)} \cdot \frac{h(z)}{e(z)}$$

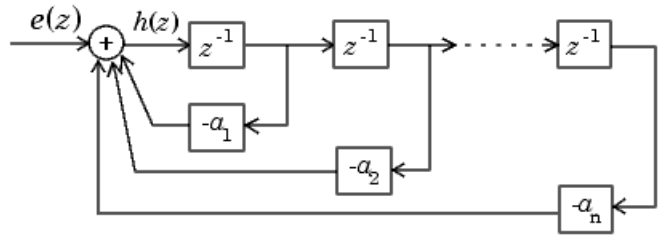
$$= \underbrace{(b_0 + b_1 z^{-1} + \ldots + b_m z^{-m})}_{\frac{u(z)}{h(z)}} \underbrace{\frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_n z^{-n}}}_{\frac{h(z)}{e(z)}}$$

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



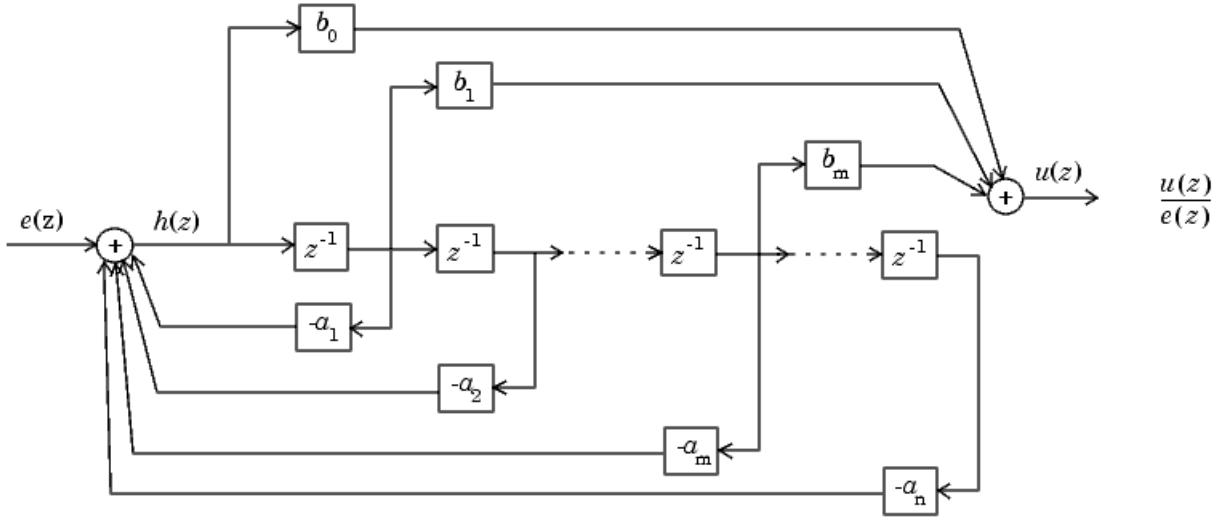
$$\frac{u(z)}{h(z)} = b_0 + b_1 z^{-1} + \ldots + b_m z^{-m}$$

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.



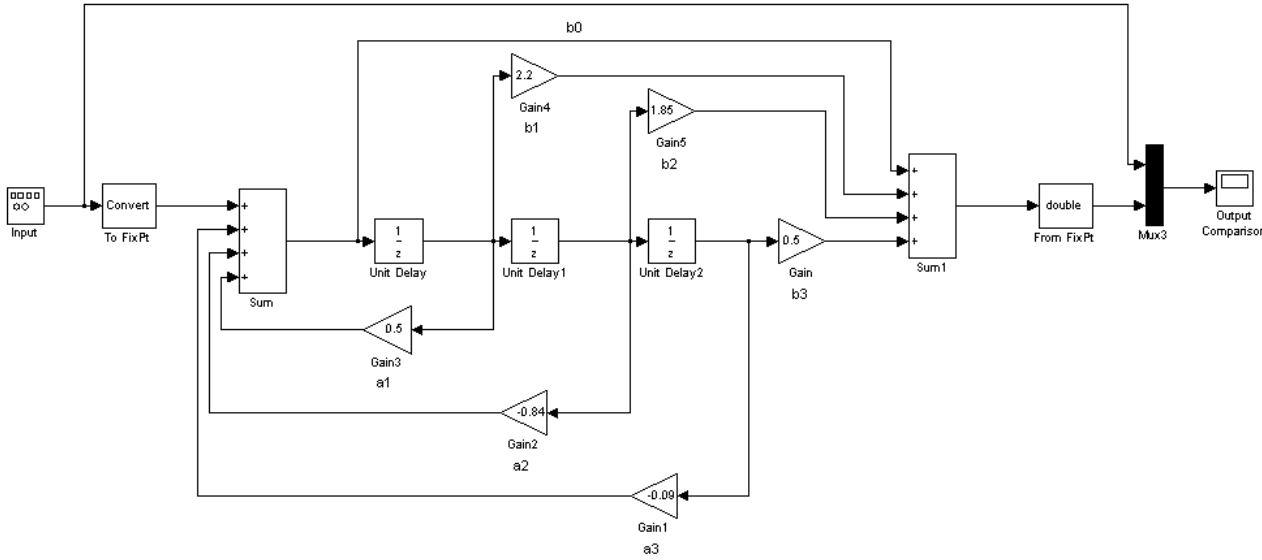
The direct form II example transfer function is given by

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

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.

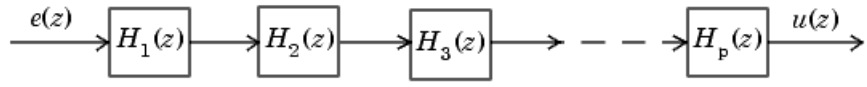


Series Cascade Form

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.



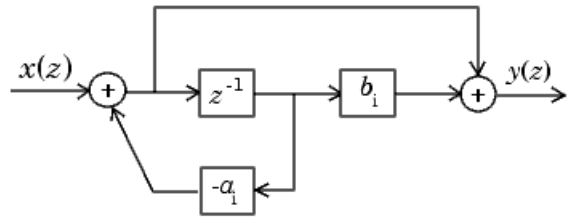
Factoring *H(z)* into $H_i(z)$ where i = 1,2,3,...,p can be done in a number of ways. Using the poles and zeros of *H(z)*, you can obtain $H_i(z)$ by grouping pairs of conjugate complex poles and pairs of conjugate complex zeros to produce second-order transfer functions, or by grouping real poles and 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

$$H(z) = H_1(z)H_2(z)\ldots H_p(z)$$
$$= \prod_{i=1}^{j} \frac{1 + b_i z^{-1}}{1 + a_i z^{-1}} \prod_{i=j+1}^{p} \frac{1 + e_i z^{-1} + f_i z^{-2}}{1 + c_i z^{-1} + d_i z^{-2}}$$

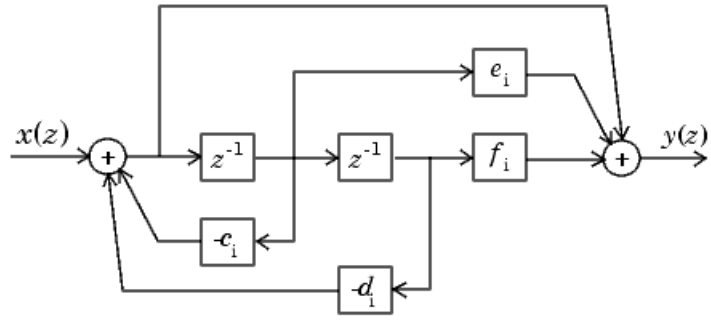
You must also take into consideration that the ordering of the individual $H_i(z)$'s will lead to systems with different numerical characteristics. You might want to try various orderings for a given set of $H_i(z)$'s to determine which gives the best numerical characteristics.

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.



$$\frac{y(z)}{x(z)} = \frac{1 + e_i z^{-1} + f_i z^{-2}}{1 + c_i z^{-1} + d_i z^{-2}}$$

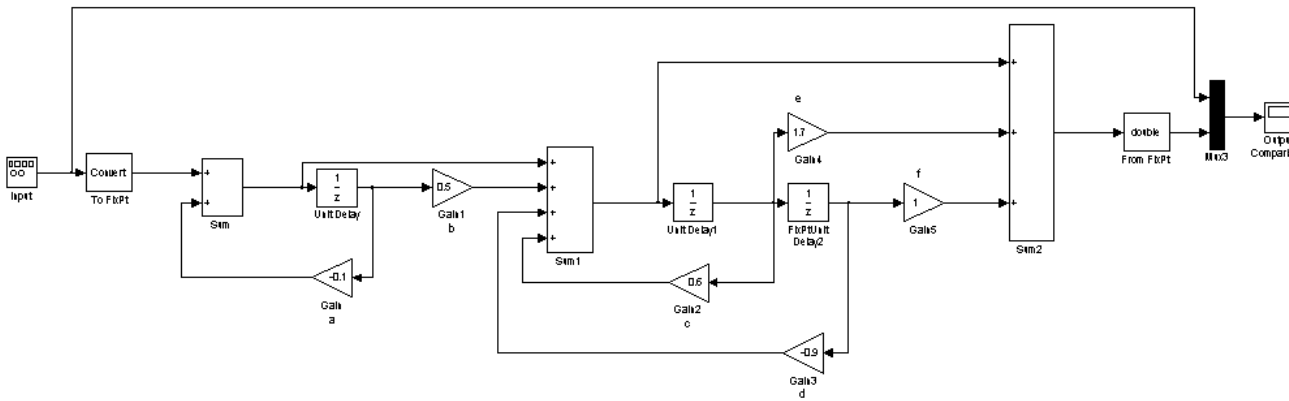
The series cascade form example transfer function is given by

$$H_{ex}(z) = \frac{(1+0.5z^{-1})(1+1.7z^{-1}+z^{-2})}{(1+0.1z^{-1})(1-0.6z^{-1}+0.9z^{-2})}$$

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.

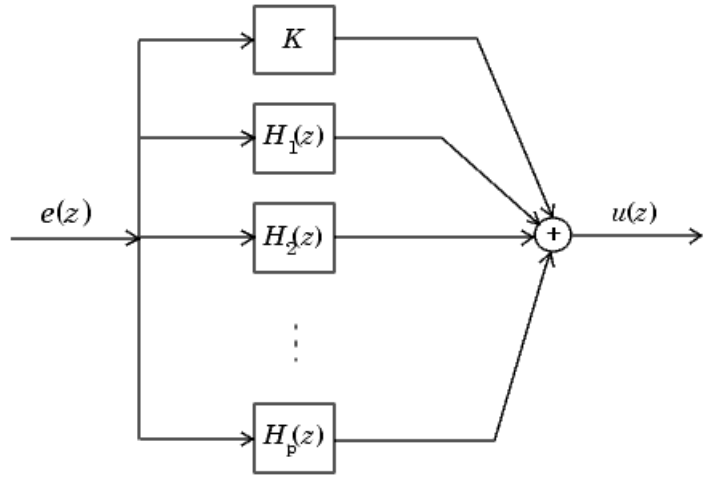


Parallel Form

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.

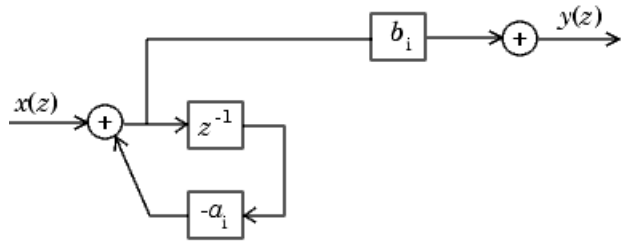


As in the series canonical form, there is no unique description for the first-order and second-order transfer function. Because of the nature of the Sum block, the ordering of the individual filters doesn't matter. However, because of the constant *K*, you can choose the first-order and second-order 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

$$H(z) = K + \sum_{i=1}^{j} H_i(z) + \sum_{i=j+1}^{p} H_i(z)$$

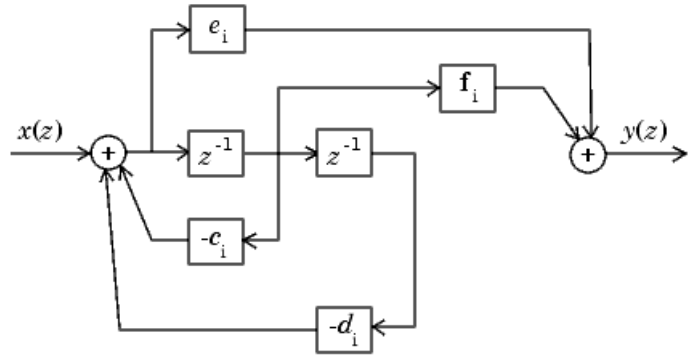
$$= K + \sum_{i=1}^{j} \frac{b_i}{1 + a_i z^{-1}} + \sum_{i=j+1}^{p} \frac{e_i + f_i z^{-1}}{1 + c_i z^{-1} + d_i z^{-2}}$$

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.



$$\frac{y(z)}{x(z)} = \frac{e_i + f_i z^{-1}}{1 + c_i z^{-1} + d_i z^{-2}}$$

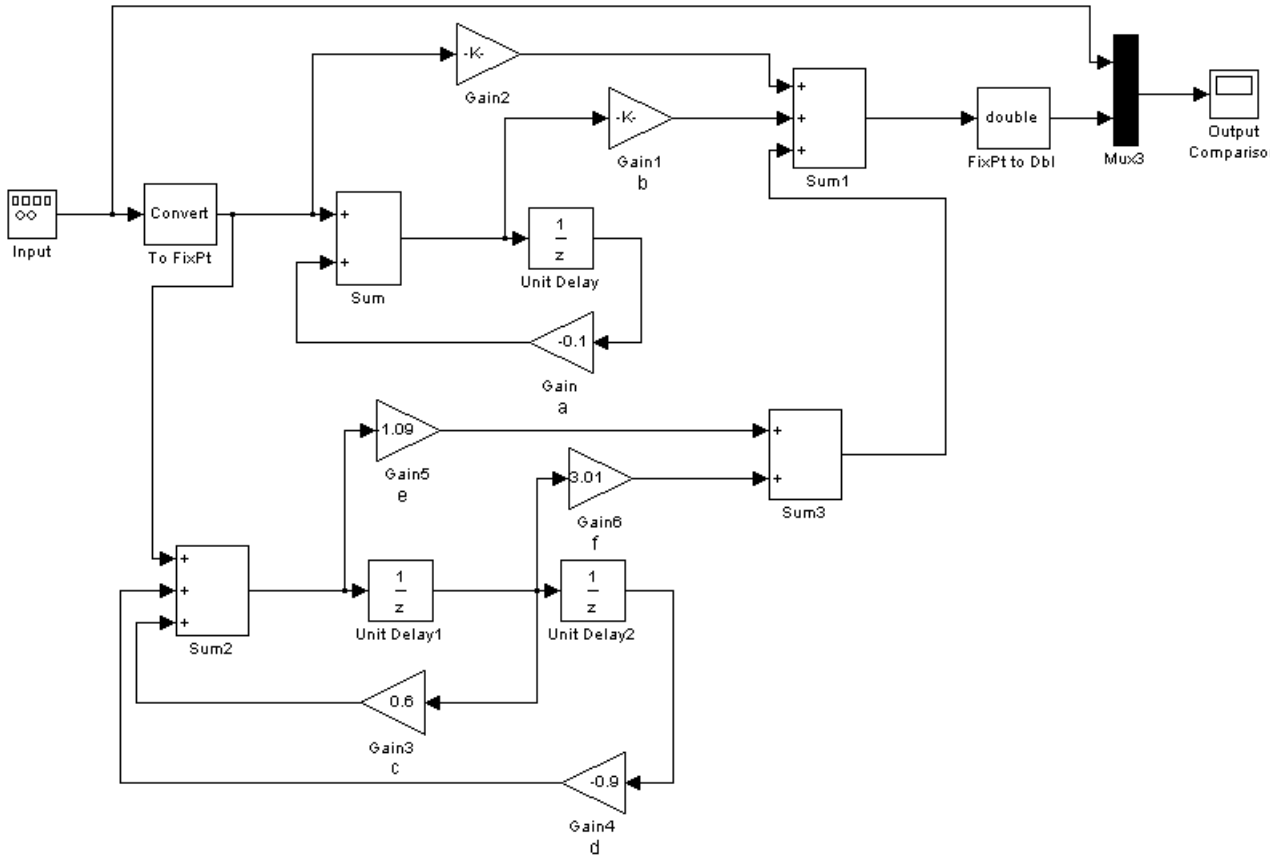
The parallel form example transfer function is given by

$$H_{ex}(z) = 5.5556 - \frac{3.4639}{1 + 0.1z^{-1}} + \frac{-1.0916 + 3.0086z^{-1}}{1 - 0.6z^{-1} + 0.9z^{-2}}$$

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.



5

Fixed-Point Advisor

Working With the Fixed-Point Advisor (p. 5-2)	Introduces the Fixed-Point Advisor
Tutorial: Converting a Model from Floating- to Fixed-Point (p. 5-5)	Provides a step-by-step tutorial on how to convert a model from floating-point to fixed-point data types

Working With the Fixed-Point Advisor

In this section...

"Introduction to the Fixed-Point Advisor" on page 5-2
"Running the Fixed-Point Advisor" on page 5-2
"Fixing a Task Failure" on page 5-3
"Automatically Fixing Failures" on page 5-3
"Batch Fixing Failures" on page 5-4

Introduction to the 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. You can use the Fixed-Point Advisor to prepare a model for conversion and obtain an initial scaling to use as the starting point for refinement and exploration inside the Fixed-Point Tool.

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 then selecting **Fixed-Point > Fixed-Point Advisor** from the Tools menu.
   - Right-clicking a subsystem block and then selecting **Fixed-Point > Fixed-Point Advisor** from the subsystem context menu.

   The Fixed-Point Advisor dialog opens.
3. Select the Fixed-Point Advisor folder in the Task Hierarchy 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 5-3.

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 9, "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.
- Fix the failure using the Action box. Use this method to automatically fix all failures. See "Automatically Fixing Failures" on page 5-3.
- Fix the failure using the Model Advisor Results Explorer. Use this method to batch fix failures. See "Batch Fixing Failures" on page 5-4

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.

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. In the Action box, click **Modify All**.

   The Action Result box displays the a table of changes.

2. 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. In the Analysis box, click **Explore Result**.

   The Model Advisor Result Explorer dialog box opens.

2. Use the Model Advisor Result Explorer to modify block parameters.

3. When you have finished making changes, in the Fixed-Point Advisor dialog box, 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.

Tutorial: Converting a Model from Floating- to Fixed-Point

In this section...

"About This Tutorial" on page 5-5
"Starting the Fixed-Point Advisor" on page 5-5
"Prepare Model for Conversion" on page 5-6
"Prepare for Data Typing and Scaling" on page 5-15
"Perform Data Typing and Scaling" on page 5-18
"Prepare for Code Generation" on page 5-21

About This Tutorial

This tutorial steps you through using the Fixed-Point Advisor to convert `fxpdemo_fpa` from using floating-point data types to using fixed-point data types. This tutorial shows you how to use the Fixed-Point Advisor to:

- Set model-wide configuration options.
- Set block-specific parameters.
- Set the initial scaling for the model.
- Validate the initial scaling against the floating-point model.

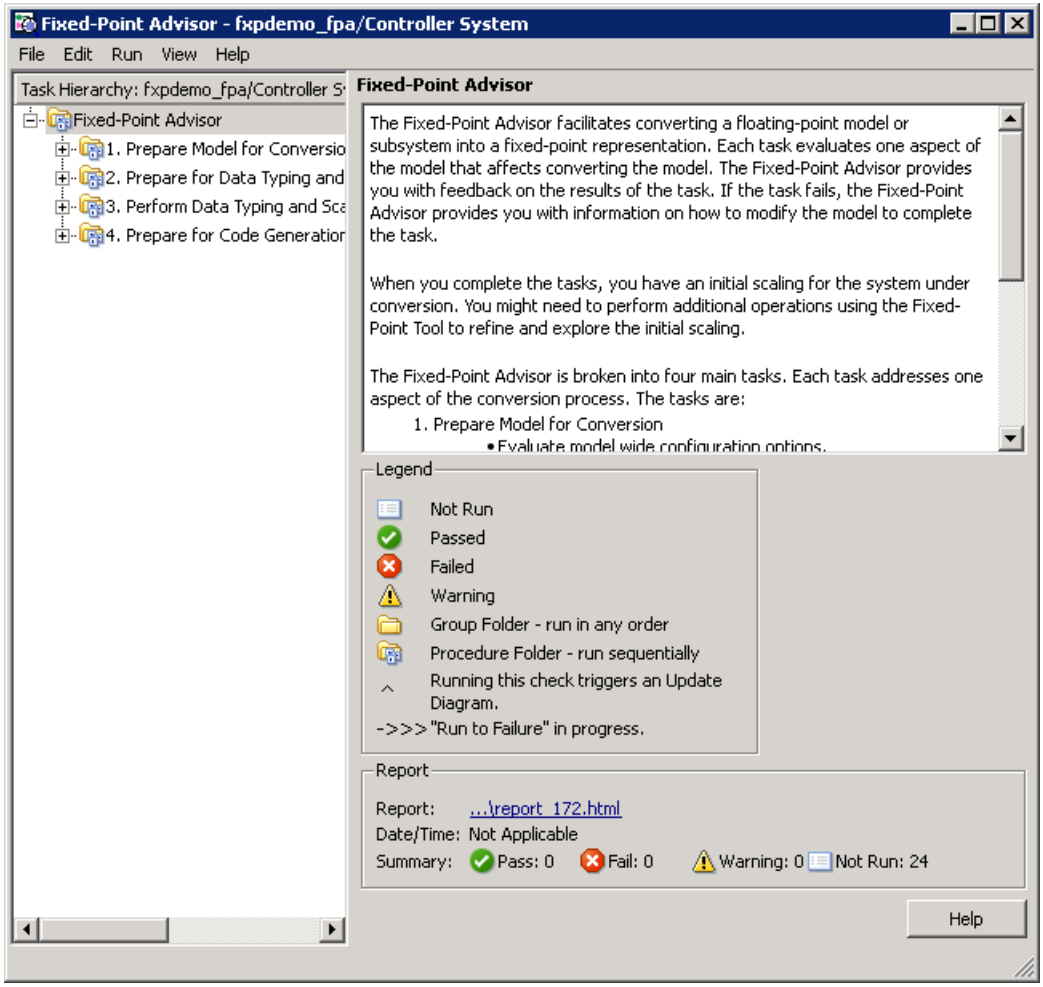
For more information about a task, see the Chapter 9, "Fixed-Point Advisor Reference".

Starting the Fixed-Point Advisor

1 Open the model: `fxpdemo_fpa.mdl`.

2 To start the Fixed-Point Advisor, right-click **Controller System** and select **Fixed-Point > Fixed-Point Advisor** from the subsystem context menu.

The Fixed-Point Advisor opens for the subsystem `Controller System`.



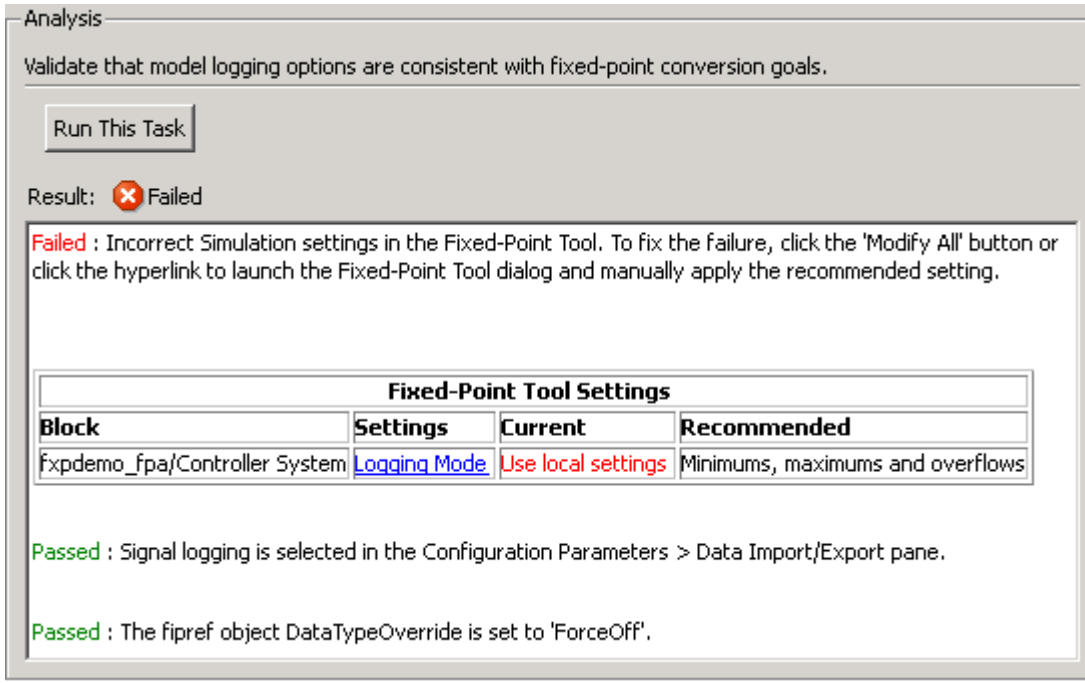
Prepare Model for Conversion

This folder contains tasks for validating model-wide settings and creating reference simulation data.

1 In the Task Hierarchy pane, right-click the **Prepare Model for Conversion** folder and select **Run to Failure** from the folder context menu.

2 The first task, **Verify model simulation settings** runs. This task validates that the model logging options are consistent with fixed-point conversion goals, ensuring that fixed-point data can be logged in downstream tasks.

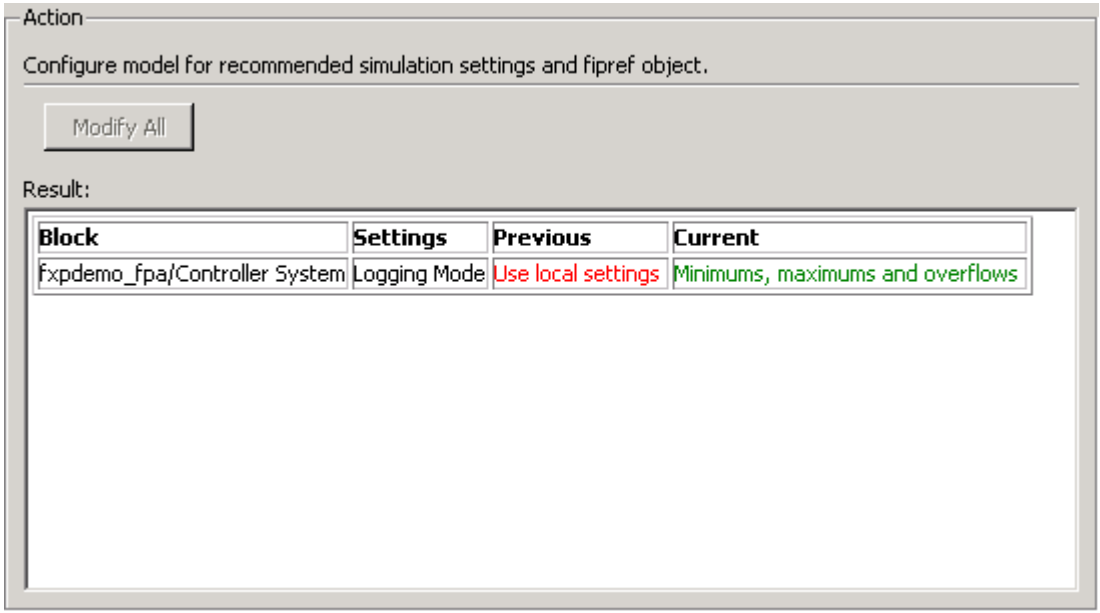
A waitbar appears while the task runs. When the run is complete, the result shows the task failed. Information about the failure is displayed in the Analysis Result box.



3 To fix the failure, in the Action box click **Modify All**.

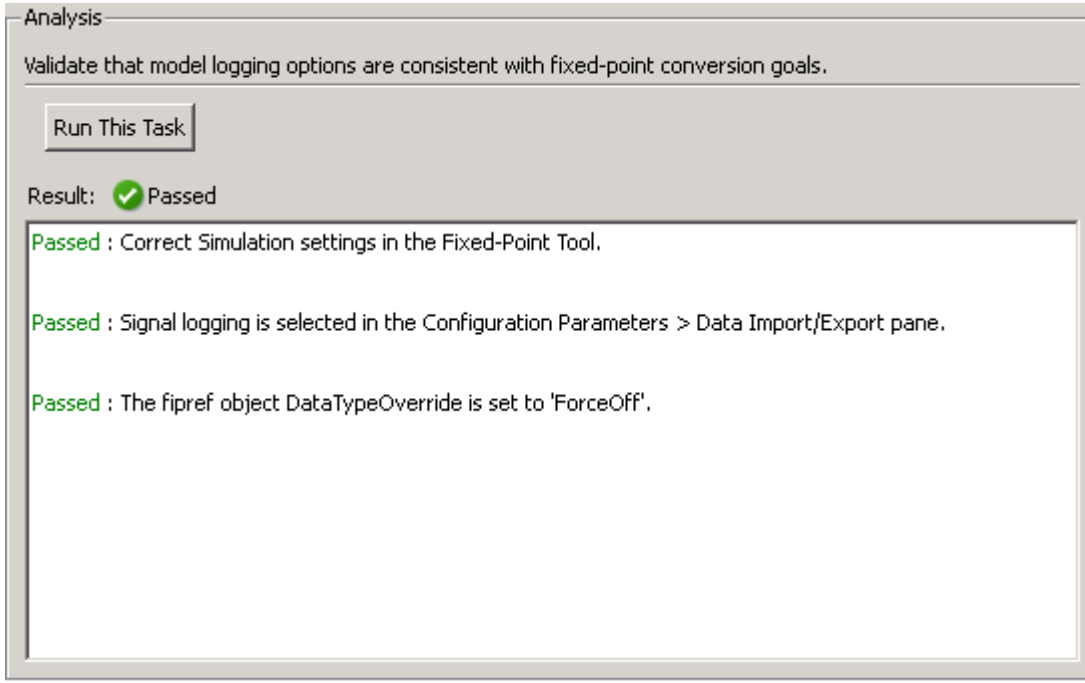
The **Modify All** action configures the model to the settings recommended in the Analysis Result. The Action Result box displays a table of changes.

Caution You should review the Analysis Result box prior to automatically fixing failures to ensure that you want to apply all of the recommended actions.



4 Click **Run This Task**.

Running the task after using the **Modify All** action verifies that you made the necessary changes. The Analysis Result box updates to display a passed result. When a task passes, the Analysis Result box displays information about why the task passes.



5 Right-click **Verify model simulation settings** and select **Continue** from the task context menu.

The Fixed-Point Advisor continues running the tasks in the folder from where you left off.

6 **Address unsupported blocks** runs. This task identifies blocks that do not support fixed-point data types and therefore cannot be converted. If at all possible, you must replace these blocks with Simulink® built-in blocks that do support fixed-point data types. If a replacement block is not available, you can isolate the unsupported block with Data Type Conversion blocks.

The task fails because the model has blocks that do not support fixed-point data types.

7 Fix the failure:

   a Insert Data Type Conversion blocks following the instructions in the Analysis Result box.

   b Rerun the task. A warning is displayed that the TrigFcn block is unsupported, however, you just isolated this block with the Data Type Conversion blocks, and therefore it is taken care of. Continue the run to failure.

8 **Verify update diagram status** runs. Your model must be able to successfully complete an update diagram action to run the checks in the Fixed-Point Advisor.

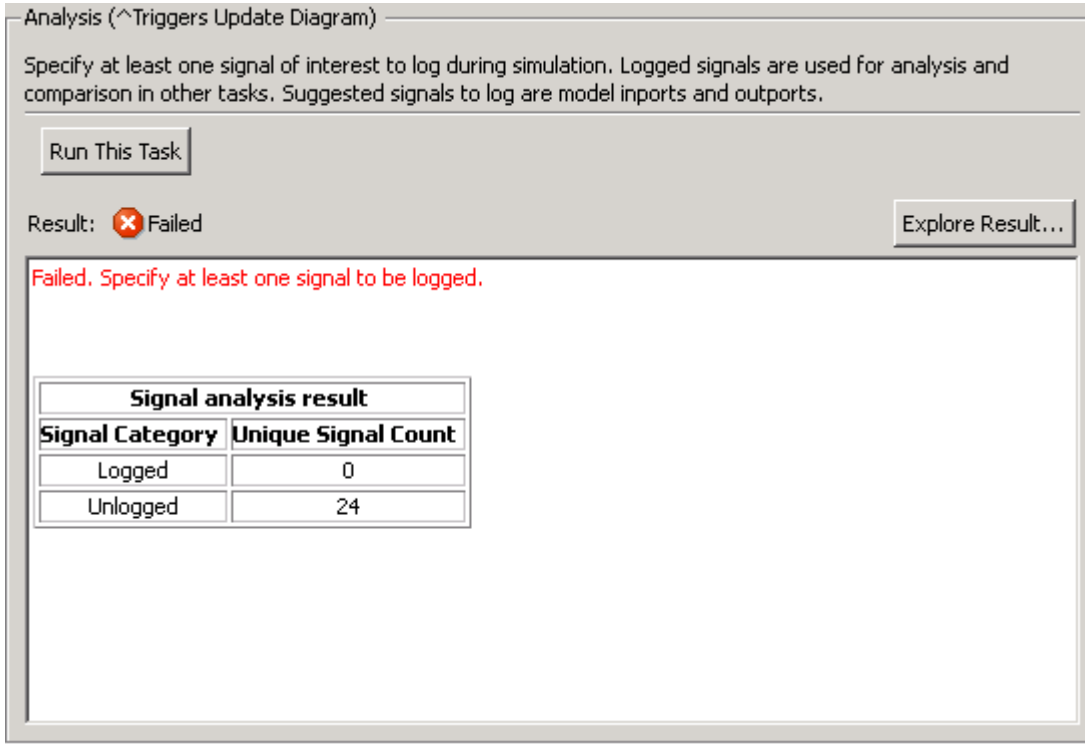
   The task passes.

9 **Set up signal logging** runs. 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.

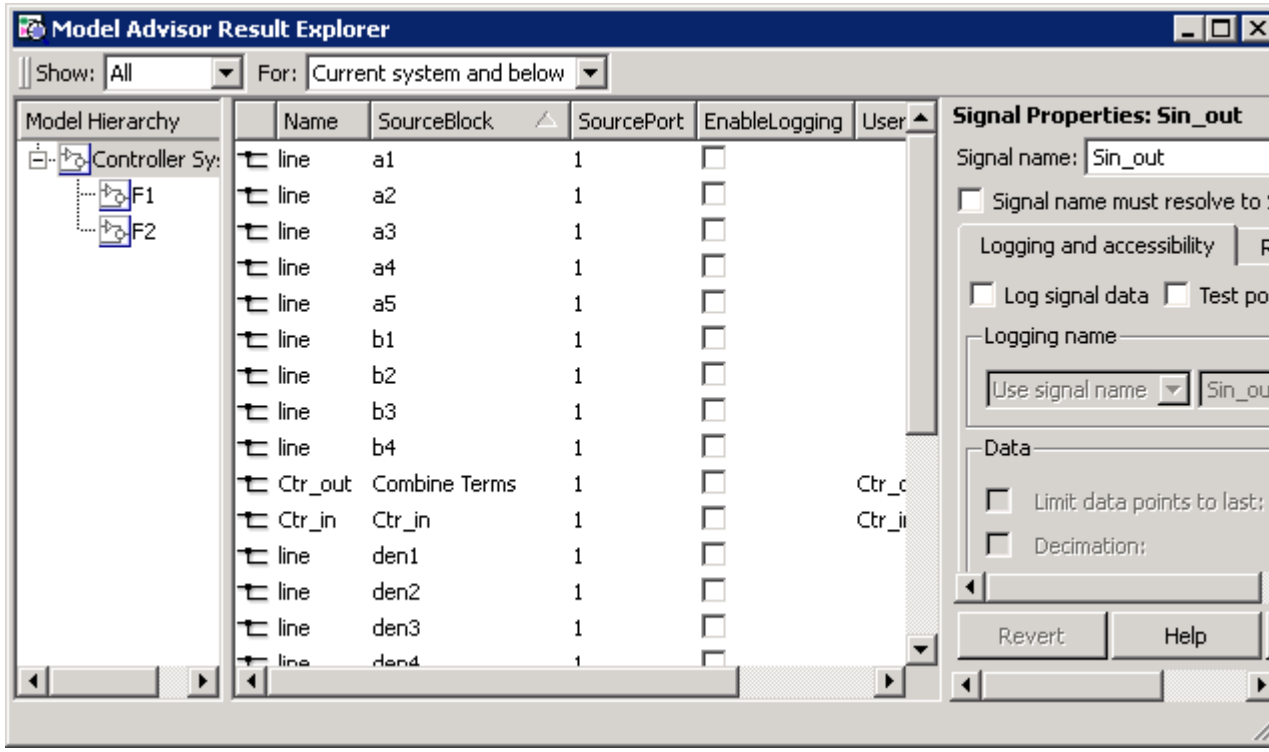
   The task fails because signal logging is not specified for any signals.

10 Fix the failure using the Model Advisor Result Explorer:

a Click the **Explore Result** button.



The Model Advisor Result Explorer opens.



b Select the signals you want to log and select the **EnableLogging** check box.

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 using any of the techniques described in "Changing Property Values" in the Simulink documentation. This includes selecting and changing multiple properties at one time.
- When you update a property value in the center pane, the Signal Properties (right) pane updates to contain the same information.

For this tutorial, log the signals connected to the Inport and Outport blocks:

- `Ctr_in`
- `Ctr_out`
- `Sin_out`

a In the Fixed-Point Advisor dialog box, click **Run This Task**.

The task passes because signal logging is enabled for at least one signal.

11 Continue running to failure. **Create simulation reference data** runs. 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 downstream checks.

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 passes.

12 **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 **Device vendor** and **Device type** parameters in the **Configuration Parameters** > **Hardware Implementation** dialog are not specified.

13 Fix the failure:

   a Click the *Hardware Implementation Device settings* link.

      The Configuration Parameters dialog box opens.

   b In the Configuration Parameters dialog box 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.

14 In the Fixed-Point Advisor dialog box, rerun the task.

   The task passes and displays the current device type and vendor.

15 Continue running to failure. **Check model configuration data validity diagnostic parameters settings** runs. This task verifies that the **Configuration Parameters** > **Diagnostics** > **Data Validity** parameters are set to `warning`. If these parameters are set to `error`, the model update diagram action fails in downstream checks.

   The task passes.

16 **Implement logic signals as Boolean data** runs. This task verifies that the **Configuration Parameters** > **Optimization** > **Implement logic signals as boolean data (vs. double)** check box is selected. If the check box is cleared, the code generated in downstream checks is not optimized.

   The task passes.

17 **Check for proper bus usage** runs. This task identifies:

- Mux blocks that are used as bus creators
- Bus signals treated as vectors in the top-level model

The task passes.

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 in “Perform Data Typing and Scaling” on page 5-18.

1 Right-click **Prepare for Data Typing and Scaling** and select **Run to Failure**.

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.

2 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.

3 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.

4 **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.

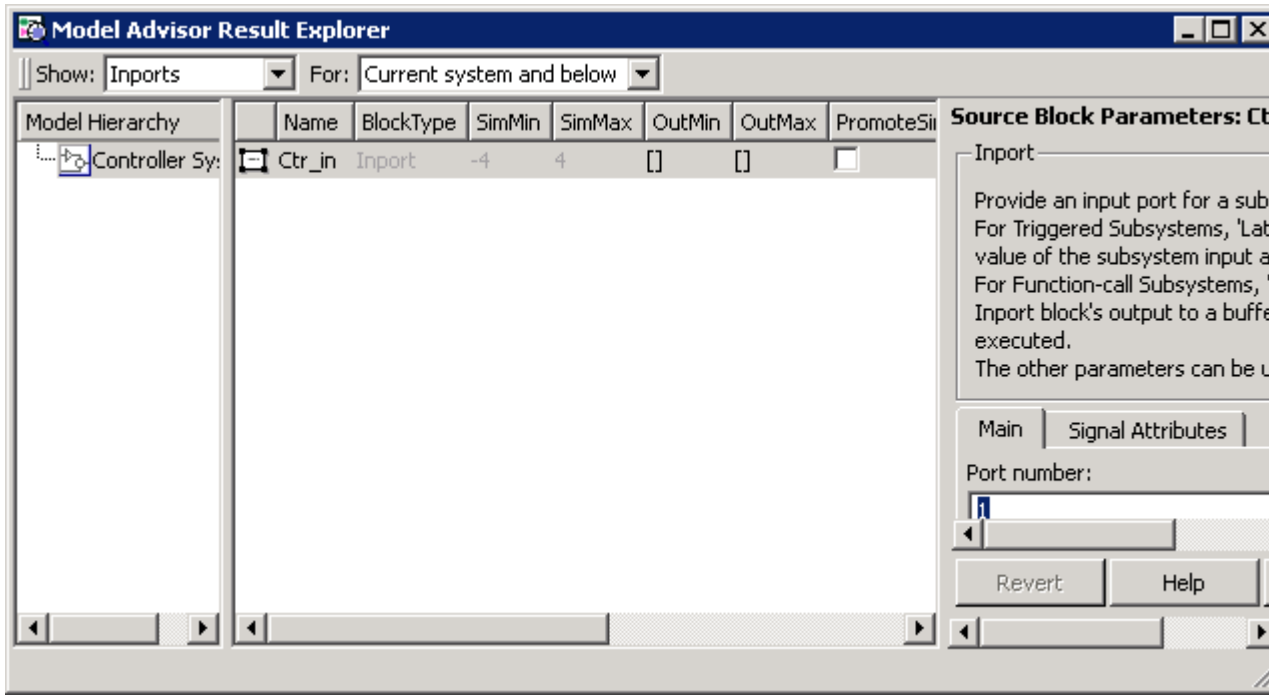
5 **Specify block minimum and maximum values** runs. You must specify block output and parameter minimum and maximum values for, at minimum, the Inport blocks in the system. You can specify the minimum and maximum values for any block in this step. Typically, they are determined during the design process based on the system you are creating.

   The task fails because you have not specified any minimum and maximum values.

6 Fix the failure 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 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 In the Fixed-Point Advisor, rerun the task.

The task passes because minimum and maximum values are specified for all Inport blocks.

d For the purpose of this tutorial, do not specify other minimum and maximum values for other blocks.

The run to failure action has completed for 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.

Perform Data Typing and Scaling

This folder contains tasks that assist you in specifying data type information and proposing initial scaling for all blocks.

The Fixed-Point Advisor computes recommended data types and scaling based on the following rules and rationales

- If the object has a fixed-point data type, and minimum and maximum values are available, the Fixed-Point Advisor keeps the current word length and covers the minimum and maximum range without overflow. Overflow is not possible, but there might be precision loss.
- If the object has a fixed-point data type, and minimum and maximum values are not available, the Fixed-Point Advisor keeps the current word length and precision with a potential for overflow.
- If the object has a floating-point data type and minimum and maximum values are available, the Fixed-Point Advisor uses the default word length and covers the minimum and maximum range without overflow.
- If the object has a floating-point data type and minimum and maximum values are not available, the Fixed-Point Advisor uses the default data type as specified by the input parameters. It is not possible to report better scaling, and the potential for overflow is unknown.

1 For the purpose of this tutorial, the tasks in the **Perform Data Typing and Scaling** folder are run one at a time.

   Select and run **Propose scaling for Inport blocks**. This task requires you to choose the fixed-point data type for Inport blocks that currently have floating-point data types. The Fixed-Point Advisor uses the type provided in the **Default Inport data type** menu with the output minimum and maximum values to provide recommended scaling for Inport blocks.

   The task fails because you have not specified a **Default Inport data type**.

2 Fix the failure by specifying `int16` in the **Default Inport data type** menu, and rerun the task.

   The result is a warning because you do not have fixed-point data type scaling for all Inport blocks. The Analysis Result box is updated with recommended fixed-point data type scaling for Inport blocks. You can

apply the recommendations by clicking the **Modify All** button, or use the Model Advisor Result Explorer dialog to manually enter fixed-point data type scaling.

3 Fix the failure by applying the recommendations using the **Modify All** button and rerun the task.

The task passes.

4 Select **Propose scaling for Constant blocks** and specify `int16` for the **Constant data type** menu.

This task requires you to choose the default data type for Constant blocks. The Fixed-Point Advisor uses the type provided in the **Constant data type** menu with either the output minimum and maximum or the Constant block values to provide recommended scaling for Constant blocks.

5 Run the task.

The task passes because you do not have Constant blocks in the model.

6 Select **Propose scaling for blocks** and specify `int16` in the **Output data type** menu. In the **Accumulator data type** and **Product output data type** menus, specify `int32`.

This task requires you to choose the fixed-point data type scaling for nonsource Simulink blocks that have floating-point data types. The Fixed-Point Advisor uses the fixed-point data types provided in the Input Parameters menus with either the output or simulation minimum and maximum values to provide recommended scaling.

7 Run the task.

The result is a warning because not all blocks have fixed-point data type scaling.

8 Fix the failure by applying all reported recommendations using the **Modify All** button and rerun the task.

The task passes.

9 Select and run **Resolve scaling conflicts**. This task resolves scaling conflicts introduced in previous tasks.

The task passes because no scaling conflicts were introduced in previous tasks.

10 Select and run **Summarize data types**. This task summarizes the data types used in the system. The Analysis Result box displays the number of:

- Floating-point data types
- Integer word lengths less than or equal to the native word size
- Integer word lengths greater than the native word size

The result is a warning because all signals are not fixed-point data types.

11 Select **Propose scaling for parameters** and specify `int32` in the **Parameter data type** menu.

This task requires you to choose the default fixed-point data type scaling for block parameters and Stateflow parameters and constants that have floating-point data types. The Fixed-Point Advisor uses the fixed-point data types provided in the **Parameter data type** menu with either the output or simulation minimum and maximum values to provide recommended scaling.

12 Run the task.

The task passes because the block parameters or Stateflow constants and parameters do not require a change in scaling.

13 Select and run **Check for numerical errors**. This task summarizes simulation numerical errors.

The task passes because there are no numerical errors.

14 Select and run **Analyze logged signals**. This task summarizes the differences between current simulation data and the reference data archived in “Create simulation reference data” on page 9-12.

The task passes. You can view the differences by clicking the *View Results* link in the Analysis Result box.

You have completed the tasks in the **Perform 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.

Prepare for Code Generation

This folder contains tasks that assist you in identifying settings that might lead to nonoptimal results in code generation

1. For the purpose of this tutorial, the tasks in the Prepare for Code Generation folder are run one at a time.

   Select and run **Disable signal logging**. This task disables unnecessary signal logging to avoid declaring extra signal memory in generated code.

   The result is a warning because there are signals that are logged.

2. Fix the warning using the **Modify All** button to disable signal logging on all signals and rerun the task.

   The task passes.

3. 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.

4. Fix the failure conditions by following the instructions in the Analysis Result box. Rerun the task.

   The task passes.

5. Select and run **Identify questionable fixed-point operations**. This task identifies fixed-point operations that can lead to nonoptimal results.

   The task results in a warning because some of the decisions made in earlier steps to make the initial scaling easier can result in less efficient code. For example, in **Prepare for Data Typing and Scaling > Relax input data type settings**, you use different data types for Product and Sum blocks. This results in less efficient code due to shifting. You need to

decide whether the scaling that you set is required, or if you can apply the recommended actions and make your code more efficient.

6 View the recommend actions in the Analysis Result box to see the types of conditions causing the warning.

You have completed the tasks in the **Prepare for Code Generation** folder. This completes the tutorial. At this point you can review the results report found at the folder level.

6

Fixed-Point Tool

Overview of the Fixed-Point Tool (p. 6-2)	Introduces the Fixed-Point Tool
Working with the Fixed-Point Tool (p. 6-5)	Describes a general workflow for using the Fixed-Point Tool
Introduction to the Tutorial (p. 6-12)	Introduces the `fxpdemo_feedback` model used in the tutorial
Tutorial: Feedback Controller (p. 6-18)	Provides a step-by-step tutorial on using the Fixed-Point Tool to scale fixed-point data types in a model

Overview of the Fixed-Point Tool

In this section...

"Introduction to the Fixed-Point Tool" on page 6-2

"Opening the Fixed-Point Tool" on page 6-2

"Understanding the Interface" on page 6-3

Introduction to the Fixed-Point Tool

The Fixed-Point Tool is a graphical user interface that automates the task of specifying scaling for 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, 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.

Opening the Fixed-Point Tool

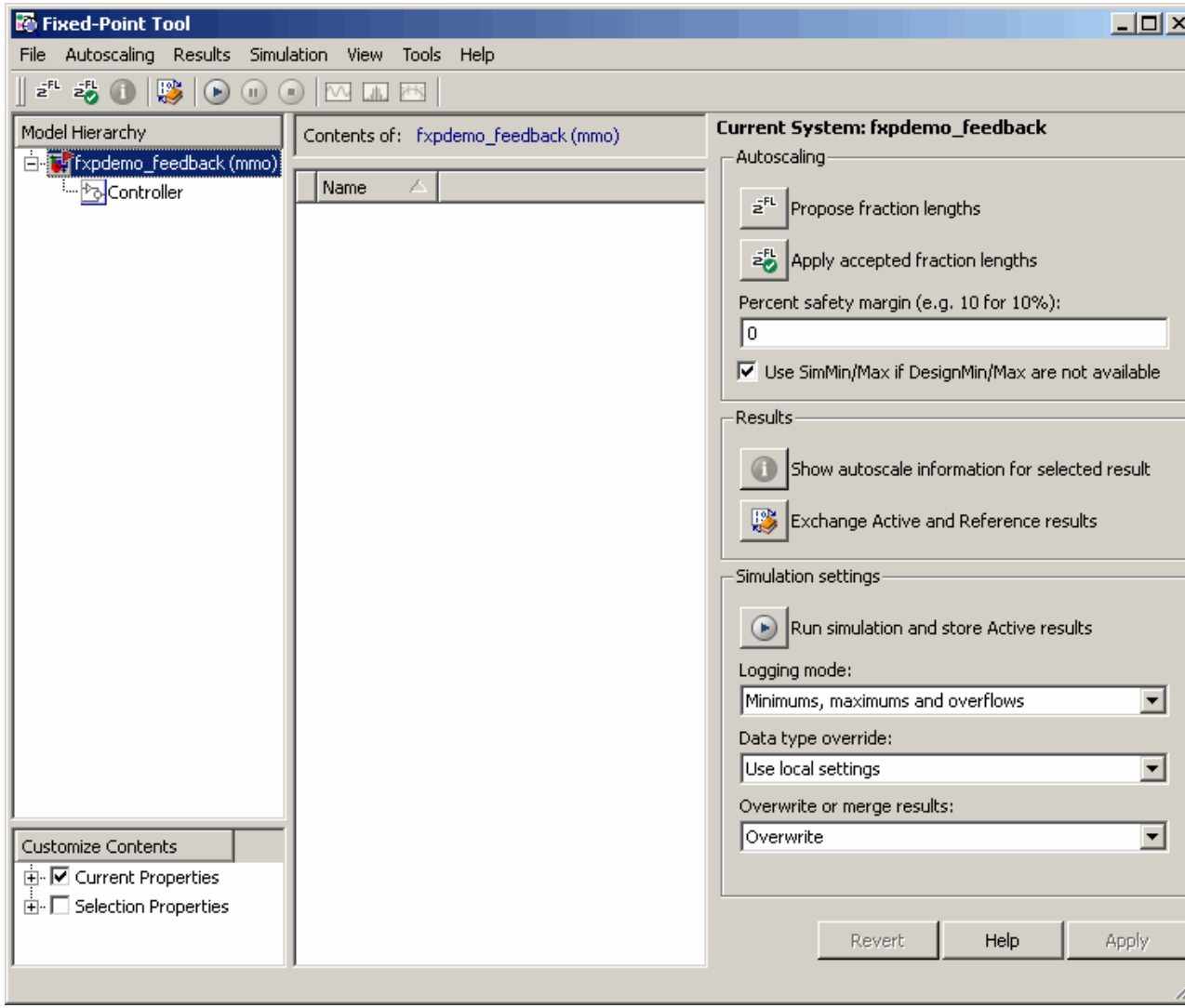
Use any of the following methods to open the Fixed-Point Tool:

- From the Simulink® **Tools** menu, select **Fixed-Point > 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 > 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 logging mode.
- **Main** toolbar — Provides buttons that execute commonly used Fixed-Point Tool commands.

For more information about each of these components, see the documentation for the `fxptdlg` function in the *Simulink Reference*.

Working with the Fixed-Point Tool

In this section...
"Fixed-Point Tool Workflow" on page 6-5
"Proposing Scaling" on page 6-5
"Reviewing Scaling Proposals" on page 6-6
"Applying Scaling" on page 6-11

Fixed-Point Tool Workflow

In general, you perform the following steps when using the Fixed-Point Tool to scale fixed-point data types in a model:

Step	Description	See...
1	Collect range data, i.e., minimum and maximum values that model objects either specify explicitly or generate during simulation, and propose fixed-point scaling.	"Proposing Scaling" on page 6-5
2	Use the Autoscale Information dialog to review the scaling proposals.	"Reviewing Scaling Proposals" on page 6-6
3	Apply the scaling proposals selectively to model objects.	"Applying Scaling" on page 6-11

Proposing Scaling

The Fixed-Point Tool generates scaling proposals for model objects that specify fixed-point data types. However, you can control whether a model object is subject to proposals by locking its scaling. If an object's **Lock output scaling against changes by the autoscaling tool** parameter is selected, the tool refrains from proposing scaling for that object.

When generating scaling proposals, the 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 *Using Simulink®* for a list of blocks that permit you to specify these values.
- Simulation minimum or maximum values — When simulating a system whose **Logging 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 **Logging mode** parameter, see the documentation for the `fxptdlg` function in the *Simulink Reference*.

The Fixed-Point Tool uses available range data to calculate scaling proposals according to the following rules:

- Design minimum and maximum values take precedence over the simulation range.
- The tool observes the simulation range only when the **Use SimMin/Max if DesignMin/Max are not available** option is selected. Otherwise, the tool ignores the simulation range.

Click the **Propose Fraction Lengths** button to generate scaling proposals for objects in your model. However, if you plan to scale fixed-point data types using simulation minimum and maximum values, you must first simulate the system to log its simulation range. See "Tutorial: Feedback Controller" on page 6-18 for an example that demonstrates how to scale fixed-point data types using only simulation range data.

Note The Fixed-Point Tool does not alter your model when it proposes scaling.

Reviewing Scaling Proposals

The Fixed-Point Tool lists its scaling proposals in the **Contents** pane under the **ProposedDT** column. The tool alerts you to potential scaling issues for each object in the list by displaying a green, yellow, or red icon, as shown here:

The proposed scaling poses no issues for this object.

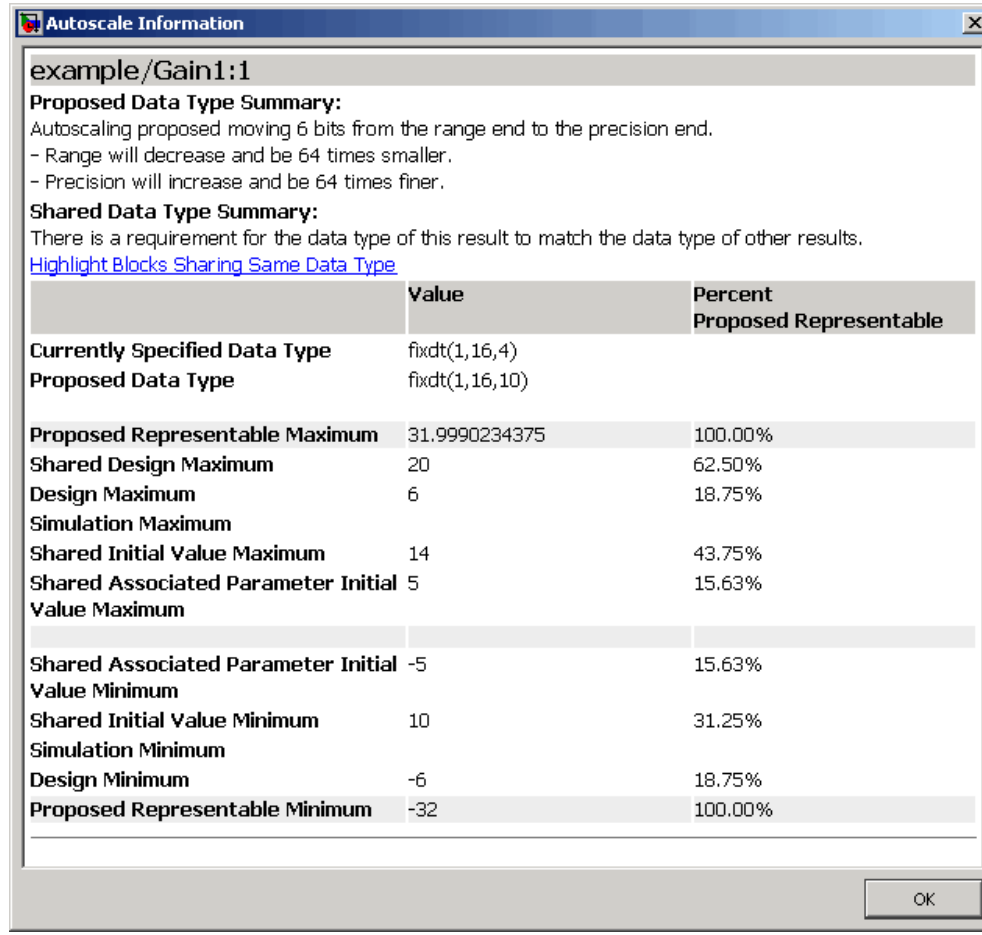
The proposed scaling poses potential issues for this object.

The proposed scaling will introduce data type errors if applied to this object.

You can review each scaling proposal using the Autoscale Information dialog, which displays the rationale underlying the proposed scaling. Also, this dialog describes potential issues or errors, and it suggests methods for resolving them. To open the dialog:

1. In the **Contents** pane, select an object that has proposed scaling.
2. Click the **Autoscale Information** button .

   The dialog appears as shown here.



The sections that follow describe the information that the dialog displays.

Proposed Data Type Summary

This section describes a scaling proposal in terms of how it differs from the object's current data type. For cases when the Fixed-Point Tool does not propose scaling, this section provides an explanation.

Needs Attention

This section lists potential issues and errors associated with scaling proposals. It displays the following icons to differentiate warnings from errors.

Indicates a warning message.

Indicates an error message.

Not only does this section describe such issues, but also it suggests methods for resolving them.

Shared Data Type Summary

This section notifies you when data type requirements associated with other model objects impact the data type of the selected object. In this case, the dialog provides a hyperlink that you can click to highlight those items in the model. To eliminate such highlighting, from the model's **View** menu, select **Remove Highlighting**.

Data Type Details

This section provides a table that lists a model object's attributes that influence its scaling 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 results from the proposed data type.
Design Maximum	Design maximum value that an object specifies using, e.g., its **Output maximum** parameter.
Simulation Maximum	Maximum value that occurs during simulation.

Item	Description
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 results from the proposed data type.

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 scaling, 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 scaling 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 scaling for model objects whose design and simulation ranges are unavailable. When data type dependencies exist, the tool considers how initial values impact the scaling proposals for neighboring objects.

- **Associated Parameters**

  Some model objects require the specification of numeric parameters to compute the value of their outputs. For example, the **Breakpoints** and **Table data** parameters of a Lookup Table (n-D) block specify values that the block requires to perform its lookup operation and generate output. Typically, blocks convert the data types of such parameters to match that of their input or output signals. When proposing scaling, the Fixed-Point

Tool considers how these “associated” parameter values impact the scaling proposals for neighboring objects.

Applying Scaling

The Fixed-Point Tool allows you to apply its scaling proposals selectively to objects in your model. Use the **Accept** check box in the **Contents** pane to specify the scaling proposals that you want to assign to model objects. The check box indicates the status of a proposal as follows:

- ☑ The Fixed-Point Tool will apply the proposed scaling to this object. By default, the tool selects the **Accept** check box when a scaling proposal differs from the object’s current scaling.
- ☐ The Fixed-Point Tool will ignore the proposed scaling and leave the current scaling intact for this object.
- ☐ No scaling proposal exists for this object. This occurs, for example, when the object specifies a data type inheritance rule or has its scaling locked.

Tip The Fixed-Point Tool enables you to customize its scaling 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.

To apply scaling proposals to objects in your model:

1. In the **Contents** pane, select the **Accept** check box associated with scaling proposals that you want to apply to model objects.
2. Click the **Apply Accepted Fraction Lengths** button .

   The Fixed-Point Tool replaces the fixed-point scaling of model objects with scaling proposals whose **Accept** check box is selected.

Introduction to the Tutorial

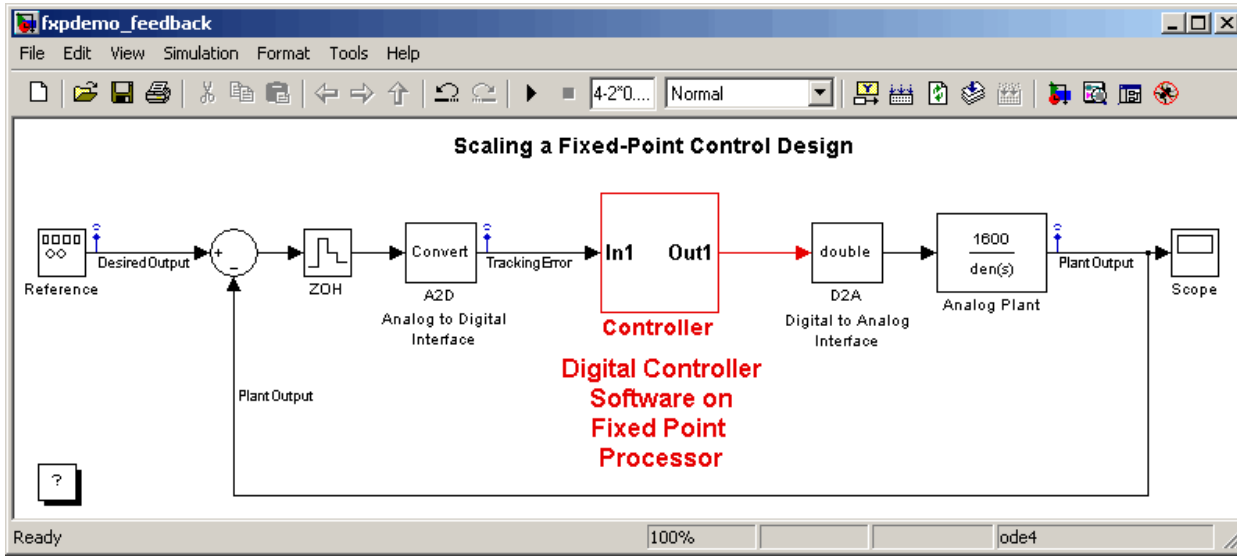
In this section...
“Opening the Demo Model” on page 6-12
“About the Demo Model” on page 6-13
“Simulation Setup” on page 6-14
“Idealized Feedback Design” on page 6-14
“Digital Controller Realization” on page 6-15

Opening the Demo Model

Open the Simulink® model of the feedback design by starting the Help browser, clicking the **Demos** tab, and selecting “Scaling a Fixed-Point Control Design” from the list of Simulink® Fixed Point™ demos. Alternatively, you can access the model directly by typing its name at the MATLAB® command line:

```
fxpdemo_feedback
```

The feedback design model is shown here.



About the Demo 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 6-15.

- **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.

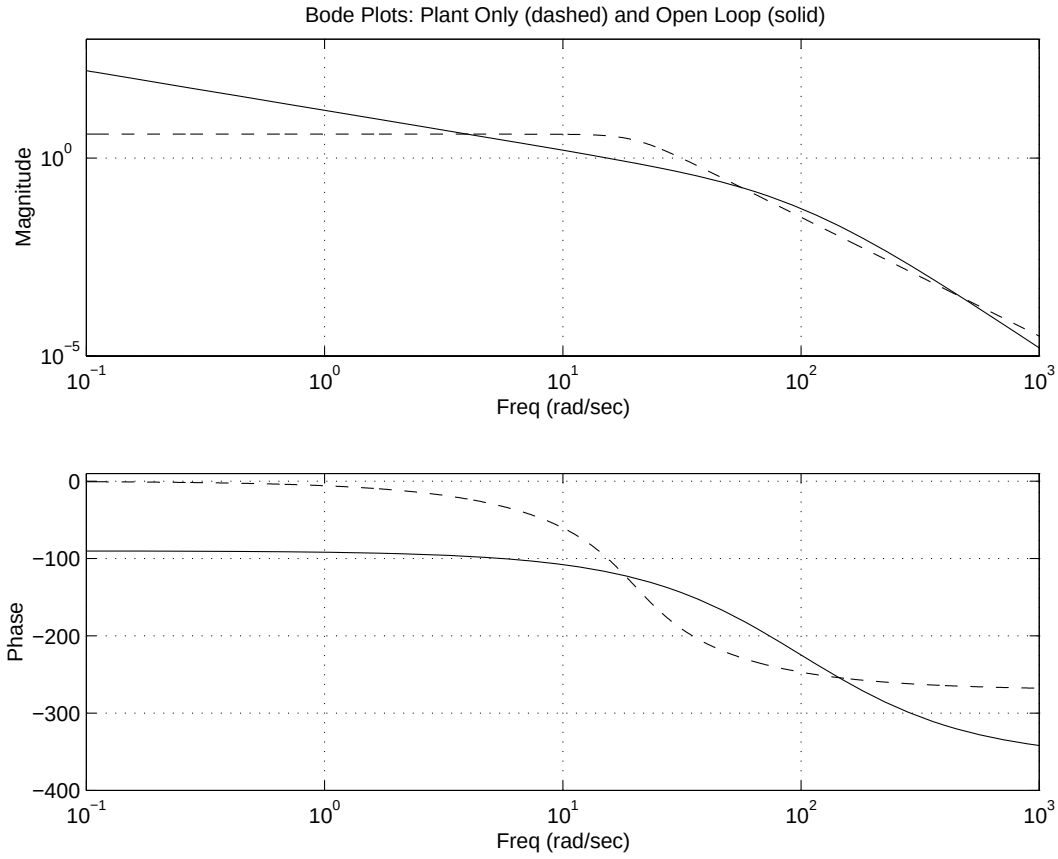
   You evaluate the behavior of the plant and the controller with a Bode plot. This evaluation is idealized, because all numbers, operations, and states are double-precision.

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

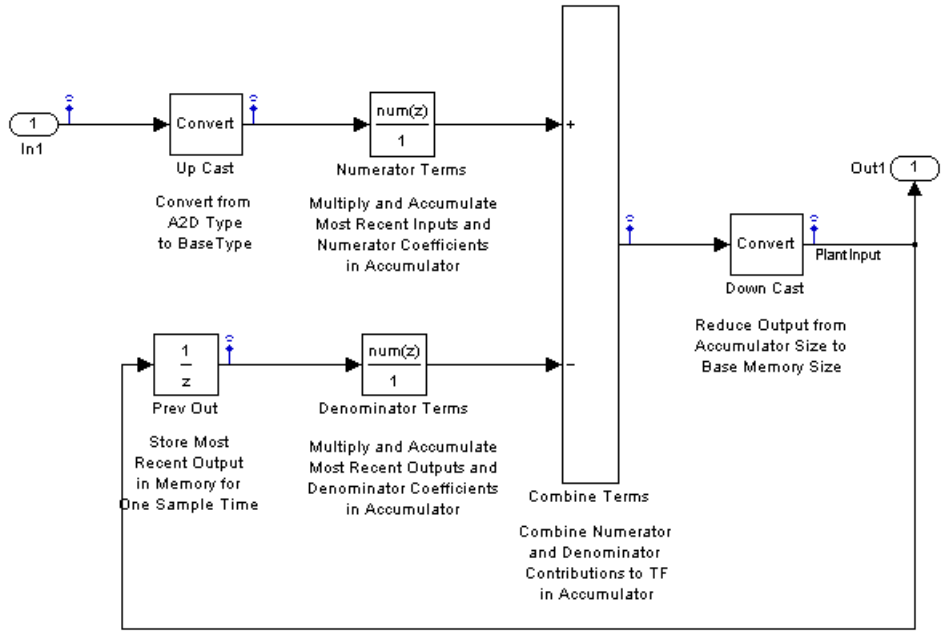
Open loop (controller and plant) and plant-only Bode plots for the “Scaling a Fixed-Point Control Design” demo are shown in the following figure. The open loop Bode plot results from a digital controller described in the idealized world of continuous time, double-precision coefficients, storage of states, 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 an M-file 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).

Tutorial: Feedback Controller

In this section...

"Before You Begin" on page 6-18

"Initial Guess at Scaling" on page 6-19

"Data Type Override" on page 6-20

"Automatic Scaling" on page 6-24

Before You Begin

The tutorial that follows 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 "Introduction to the Tutorial" on page 6-12). 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 tutorial demonstrates the following approach:

1. "Initial Guess at Scaling" on page 6-19. 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 6-20. 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 Scaling" on page 6-24. 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. Afterward, you determine the quality of the results by examining the input and output of the model's analog plant.

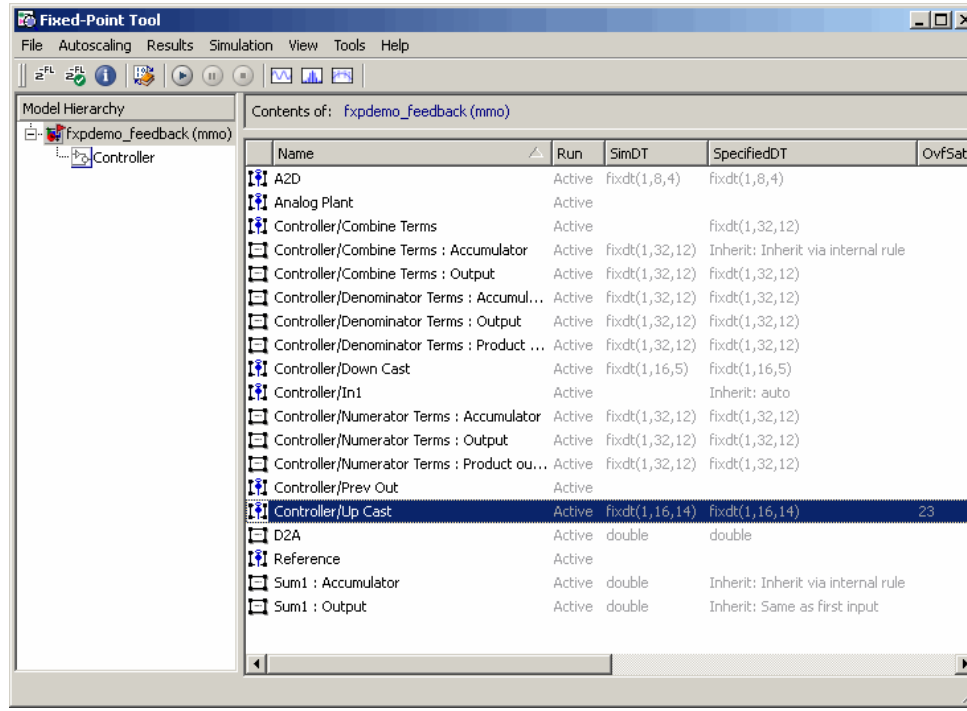
To begin, open both the `fxpdemo_feedback` model and the Fixed-Point Tool.

Initial Guess at Scaling

Initial guesses for the scaling of each block are already specified in each block mask in the model. This task is included only to illustrate the difficulty of guessing the best scaling.

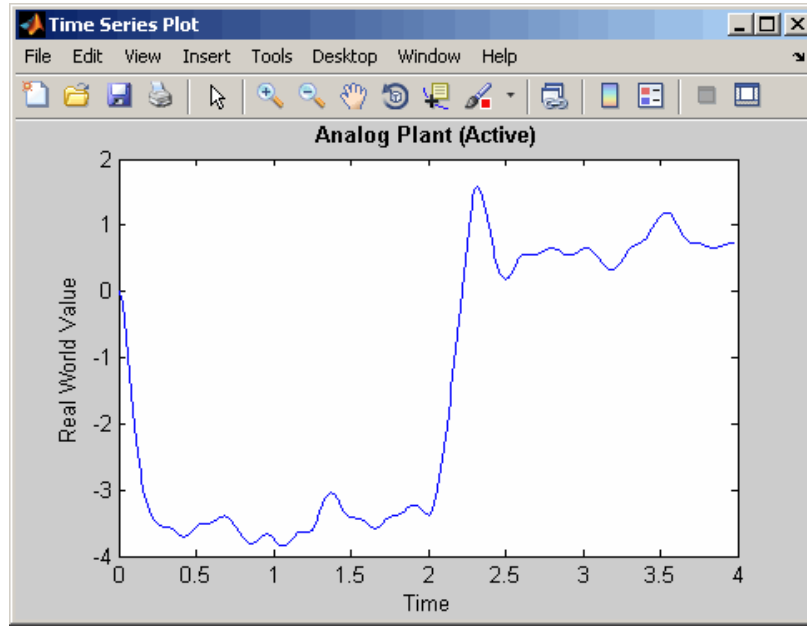
1 In the Fixed-Point Tool, click the **Start** button .

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. The tool stores the results as an active run, denoted by the `Active` label in the **Run** column. The **OvfSat** column reveals that the `Up Cast` block saturated 23 times, which indicates a poor guess for its scaling.



2 In the **Contents** pane of the Fixed-Point Tool, select the Transfer Fcn block named `Analog Plant` and then click the **Time Series Plot** 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.

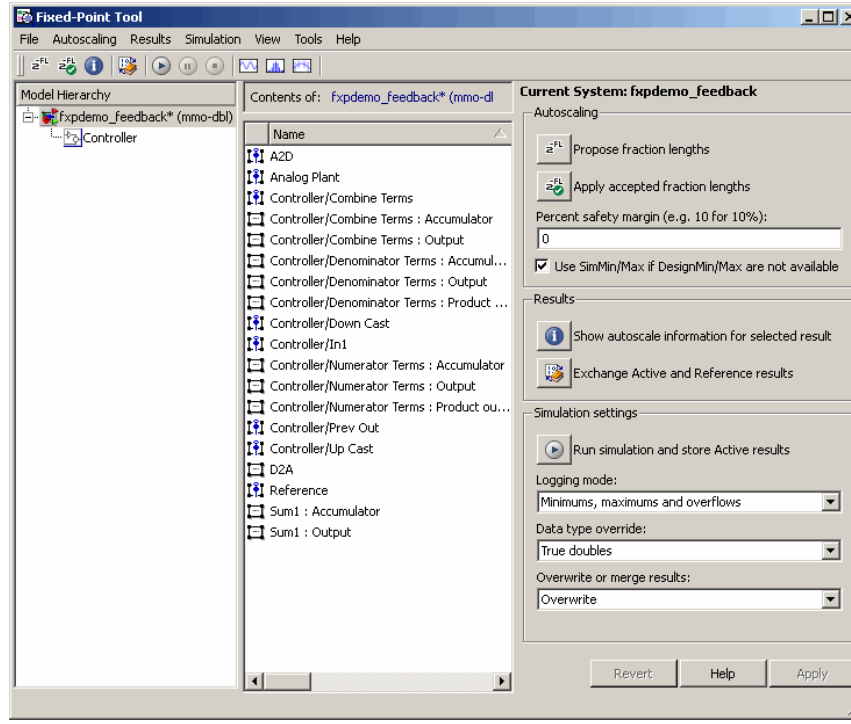
Data Type Override

Data type override mode enables you to perform a global override of the fixed-point data types and scaling using double-precision numbers, 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 **Dialog** pane of the Fixed-Point Tool:

- Click the **Exchange Active and Reference results** button. The tool swaps the results that it stores as an active run with those that it stores as a reference run. The existing results are now stored as a reference run, denoted by the `Reference` label in the **Run** column, and the active run is empty. This prevents the tool from overwriting the previous set of results.
- Ensure that the **Logging mode** parameter for the `fxpdemo_feedback` system specifies `Minimums, maximums and overflows`. This option enables logging for all the model's blocks that can log such data.
- Specify the **Data type override** parameter for the `fxpdemo_feedback` system as `True doubles`. This option overrides all local data type settings for the model's subsystems.

The Fixed-Point Tool appears as follows.



2 In the Fixed-Point Tool, click the **Start** button .

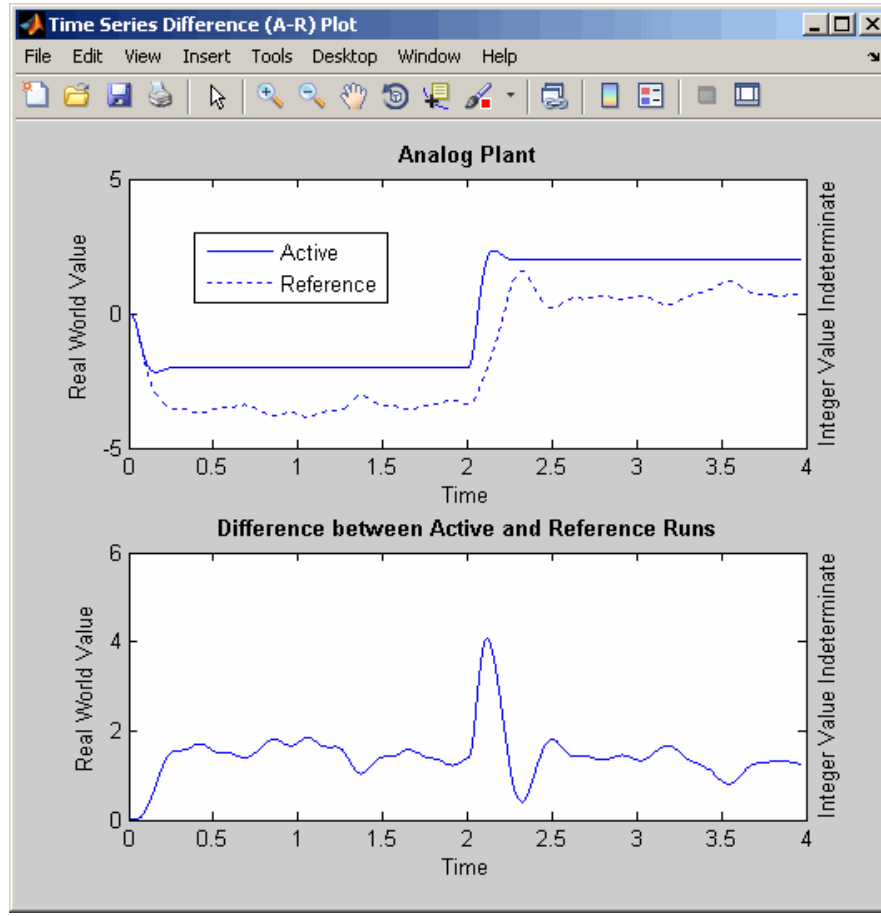
The Simulink software simulates the `fxpdemo_feedback` model in data type override mode and stores the results as an active run. Afterward, the Fixed-Point Tool displays in its **Contents** pane the active run results along with those of the reference run that you generated previously (see "Initial Guess at Scaling" on page 6-19). The **SimDT** (simulation data type) column for the active run shows that the model's blocks used a `double` data type during simulation.

Contents of: fxpdemo_feedback* (mmo-dbl)

Name	Run	SimDT	SpecifiedDT	Proposed
A2D	Active	double	fixdt(1,8,4)	
Analog Plant	Active			
Controller/Combine Terms	Active		fixdt(1,32,12)	
Controller/Combine Terms : Accumulator	Active	double	Inherit: Inherit via internal rule	
Controller/Combine Terms : Output	Active	double	fixdt(1,32,12)	
Controller/Denominator Terms : Accumul...	Active	double	fixdt(1,32,12)	
Controller/Denominator Terms : Output	Active	double	fixdt(1,32,12)	
Controller/Denominator Terms : Product ...	Active	double	fixdt(1,32,12)	
Controller/Down Cast	Active	double	fixdt(1,16,5)	
Controller/In1	Active		Inherit: auto	
Controller/Numerator Terms : Accumulator	Active	double	fixdt(1,32,12)	
Controller/Numerator Terms : Output	Active	double	fixdt(1,32,12)	
Controller/Numerator Terms : Product ou...	Active	double	fixdt(1,32,12)	
Controller/Prev Out	Active			
Controller/Up Cast	Active	double	fixdt(1,16,14)	
D2A	Active	double	double	
Reference	Active			
Sum1 : Accumulator	Active	double	Inherit: Inherit via internal rule	
Sum1 : Output	Active	double	Inherit: Same as first input	
A2D	Reference	fixdt(1,8,4)	fixdt(1,8,4)	
Analog Plant	Reference			
Controller/Combine Terms	Reference		fixdt(1,32,12)	
Controller/Combine Terms : Accumulator	Reference	fixdt(1,32,12)	Inherit: Inherit via internal rule	
Controller/Combine Terms : Output	Reference	fixdt(1,32,12)	fixdt(1,32,12)	
Controller/Denominator Terms : Accumul...	Reference	fixdt(1,32,12)	fixdt(1,32,12)	
Controller/Denominator Terms : Output	Reference	fixdt(1,32,12)	fixdt(1,32,12)	
Controller/Denominator Terms : Product ...	Reference	fixdt(1,32,12)	fixdt(1,32,12)	
Controller/Down Cast	Reference	fixdt(1,16,5)	fixdt(1,16,5)	
Controller/In1	Reference		Inherit: auto	
[illegible]	[illegible]rence	fixdt(1,32,12)	fixdt(1,32,12)	

3 In the **Contents** pane of the Fixed-Point Tool, select the Transfer Fcn block named `Analog Plant` that corresponds with either the active or the reference run, and then click the **Time Series Difference (A-R) Plot** button .

The Fixed-Point Tool plots both the active and reference 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.



Automatic Scaling

Using the automatic scaling 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 autoscaling results when the maximum and minimum simulation values cover the full intended operating range of your design. See the `autofixexp` reference page for more information.

Perform automatic scaling 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 **Dialog** pane, specify the **Percent safety margin** parameter as `20`.

   The **Percent safety margin** 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.

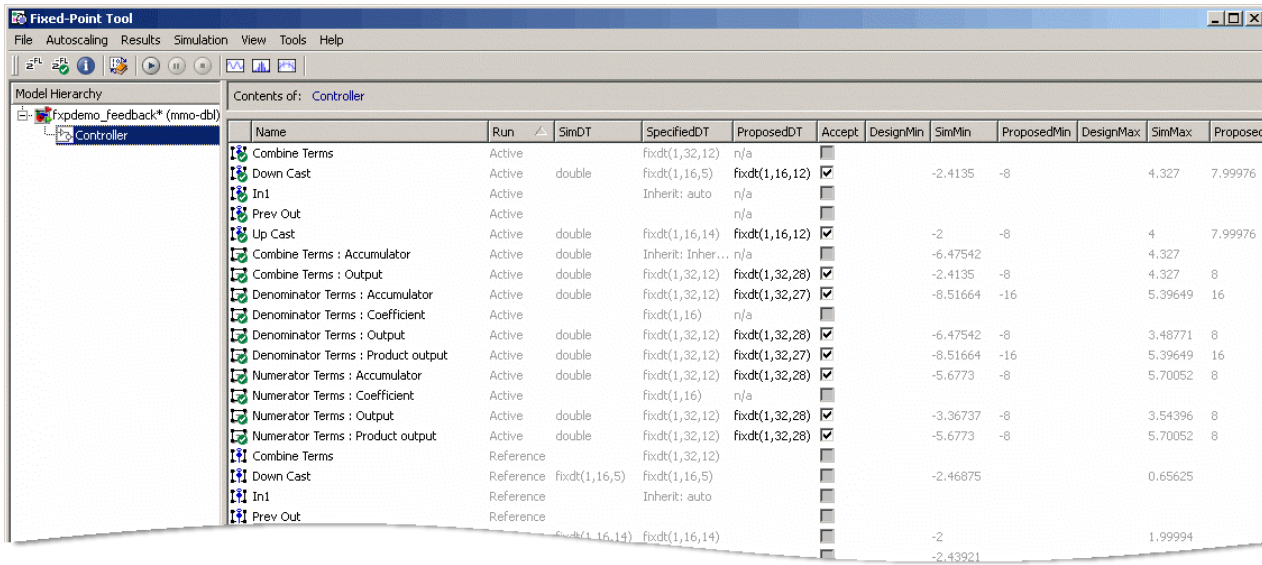
   Because of the nonlinear effects of quantization, a fixed-point simulation will produce results that are different from an idealized, doubles-based simulation. Signals in a fixed-point simulation can cover a larger or smaller range than in a doubles-based simulation. If the range increases enough, overflows 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, click the **Propose fraction lengths** button .

   The Fixed-Point Tool analyzes the scaling of all fixed-point blocks whose:

   - **Lock output scaling against changes by the autoscaling tool** parameter is not selected.
   - **Output data type** parameter specifies a generalized fixed-point number.

The Fixed-Point Tool uses the minimum and maximum values stored in the active 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 as shown here:



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 active run is selected for each of the Controller subsystem's blocks.

4 In the Fixed-Point Tool, click the **Apply accepted fraction lengths** button .

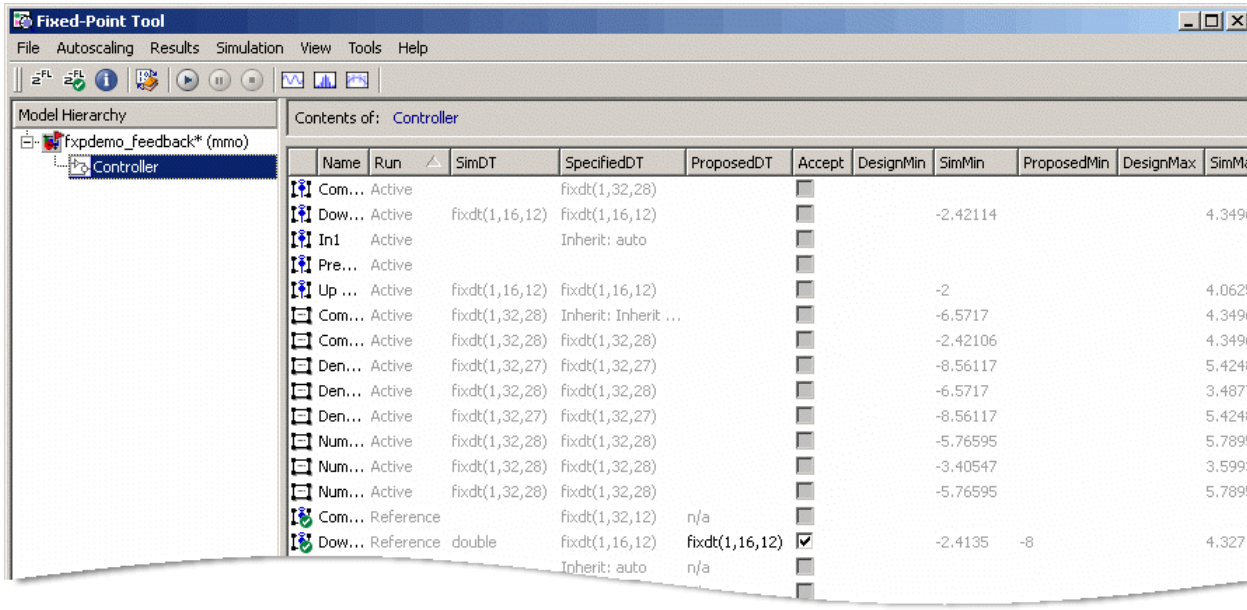
The Fixed-Point Tool applies to the Controller subsystem's blocks the scaling proposals that you accepted in the previous step.

5 In the **Model Hierarchy** pane of the Fixed-Point Tool, select the `fxpdemo_feedback` system. In the **Dialog** pane:

- Click the **Exchange Active and Reference results** button. The tool swaps the results that it stores as an active run with those that it stores as a reference run. The simulation results corresponding to `True doubles` override are now stored as a reference run. This prevents the tool from overwriting the results that we want to retain.
- Specify the **Data type override** parameter as `Use local settings`. This option enables each of the model's subsystems to use its locally specified data type settings.

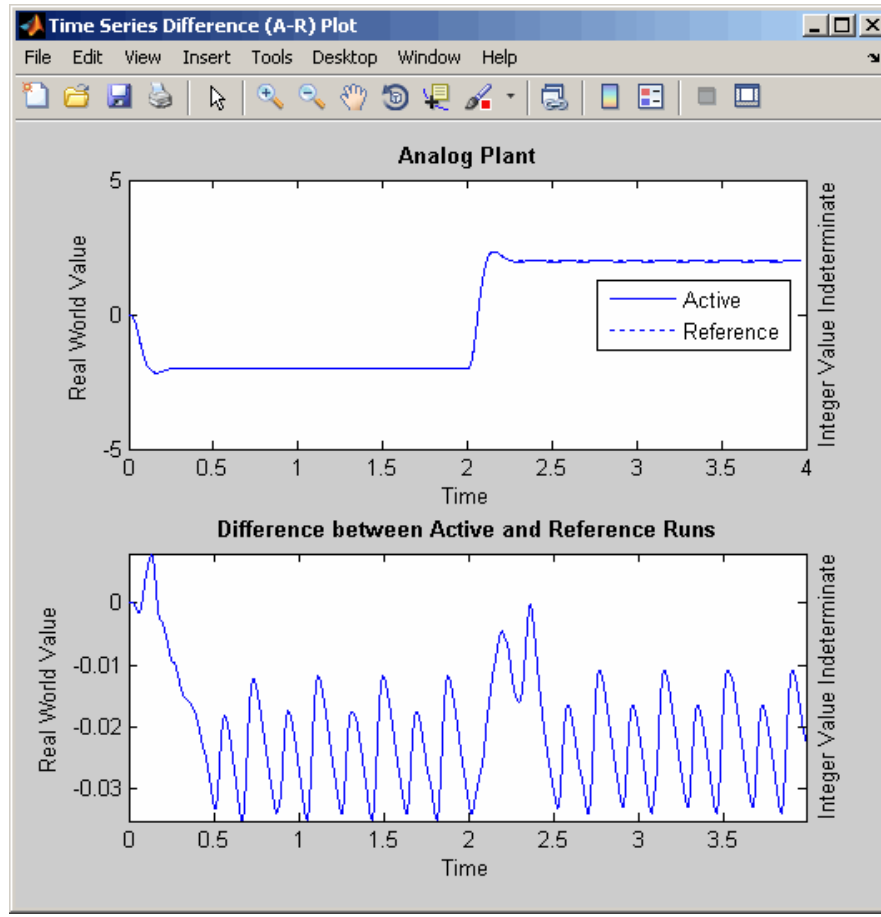
6 In the Fixed-Point Tool, click the **Start** button .

The Simulink software simulates the `fxpdemo_feedback` model using the new scaling that you applied in step 4. 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's blocks used fixed-point data types with the new scaling.

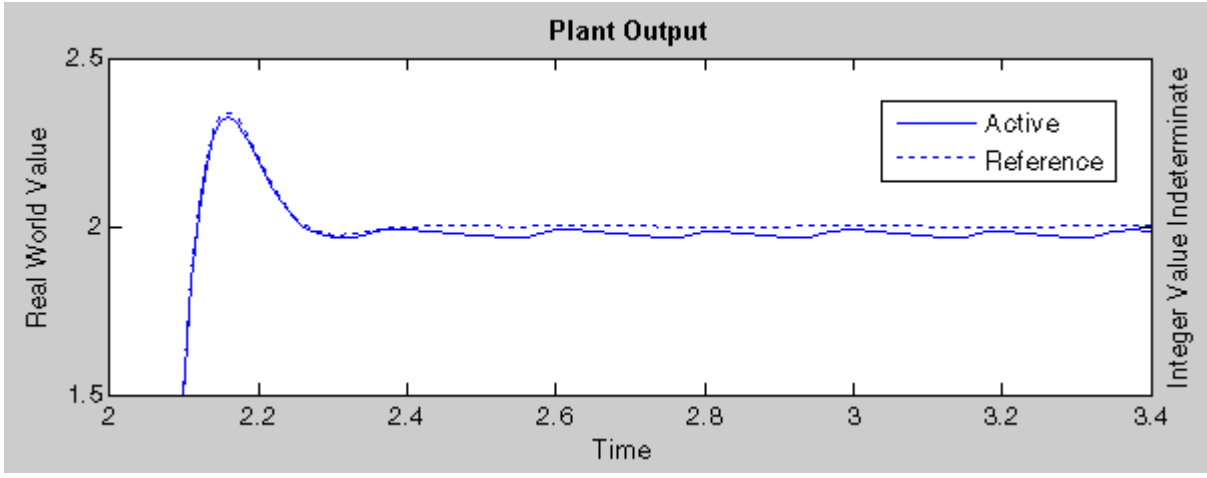


7 In the **Model Hierarchy** pane of the Fixed-Point Tool, select the `fxpdemo_feedback` system. In the **Contents** pane, select the Transfer Fcn block named `Analog Plant` that corresponds with either the active or the reference run, and then click the **Time Series Difference (A-R) Plot** button .

The Fixed-Point Tool plots the active and reference versions of the plant output signal, as well as their difference. Compare the ideal plant output signal (of data type `double`) in the reference run with its fixed-point version in the active run.



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 cursor 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.



Note that the plant output signal represented by the active run achieves a steady state, but a small limit cycle is present because of poor A/D design.

7

Tutorial: Producing Lookup Table Data

Overview (p. 7-2)	Provides an overview of the topics covered by the tutorial
Worst-Case Error for a Lookup Table (p. 7-3)	Describes worst-case error for a lookup table, and how to find it using the `fixpt_look1_func_plot` function
Creating Lookup Tables for a Sine Function (p. 7-6)	Provides a step-by-step tutorial on how to make lookup tables using the `fixpt_look1_func_approx` function
Summary: Using the Lookup Table Functions (p. 7-22)	Briefly summarizes conclusions from the tutorial on how to use `fixpt_look1_func_plot` and `fixpt_look1_func_approx` to create lookup tables
Effect of Spacing on Speed, Error, and Memory Usage (p. 7-23)	Compares lookup tables with differing spacing—uneven, even, and power of two

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?" on page 7-3

"Example: Square Root Function" on page 7-3

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.

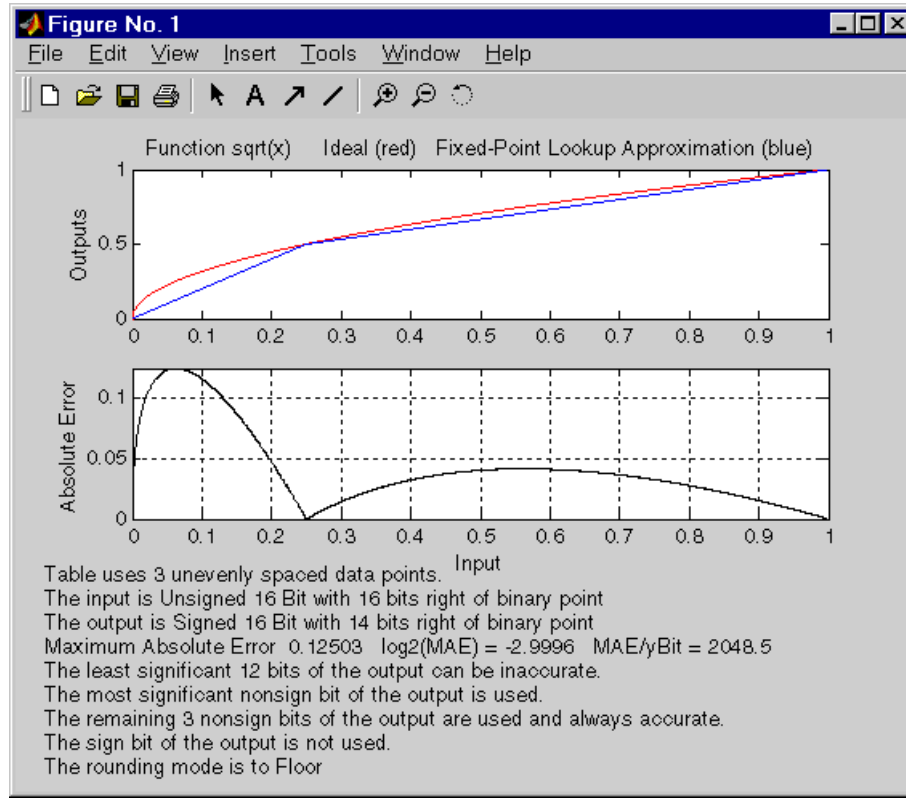


Figure No. 1
File Edit View Insert Tools Window Help
Function sqrt(x) Ideal (red) Fixed-Point Lookup Approximation (blue)
Outputs
Absolute Error
Input
Table uses 3 unevenly spaced data points.
The input is Unsigned 16 Bit with 16 bits right of binary point
The output is Signed 16 Bit with 14 bits right 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" on page 7-6
"Parameters for fixpt_look1_func_approx" on page 7-6
"Setting Function Parameters for the Lookup Table" on page 7-8
"Example: Using errmax with Unrestricted Spacing" on page 7-8
"Example: Using nptsmax with Unrestricted Spacing" on page 7-12
"Example: Using errmax with Even Spacing" on page 7-14
"Example: Using nptsmax with Even Spacing" on page 7-15
"Example: Using errmax with Power of Two Spacing" on page 7-16
"Example: Using nptsmax with Power of Two Spacing" on page 7-18
"Specifying Both errmax and nptsmax" on page 7-19
"Comparing the Examples" on page 7-20

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 7-8, 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 7-8.

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 7-12.

The section "Specifying Both errmax and nptsmax" on page 7-19 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 7-13 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 7-8, 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

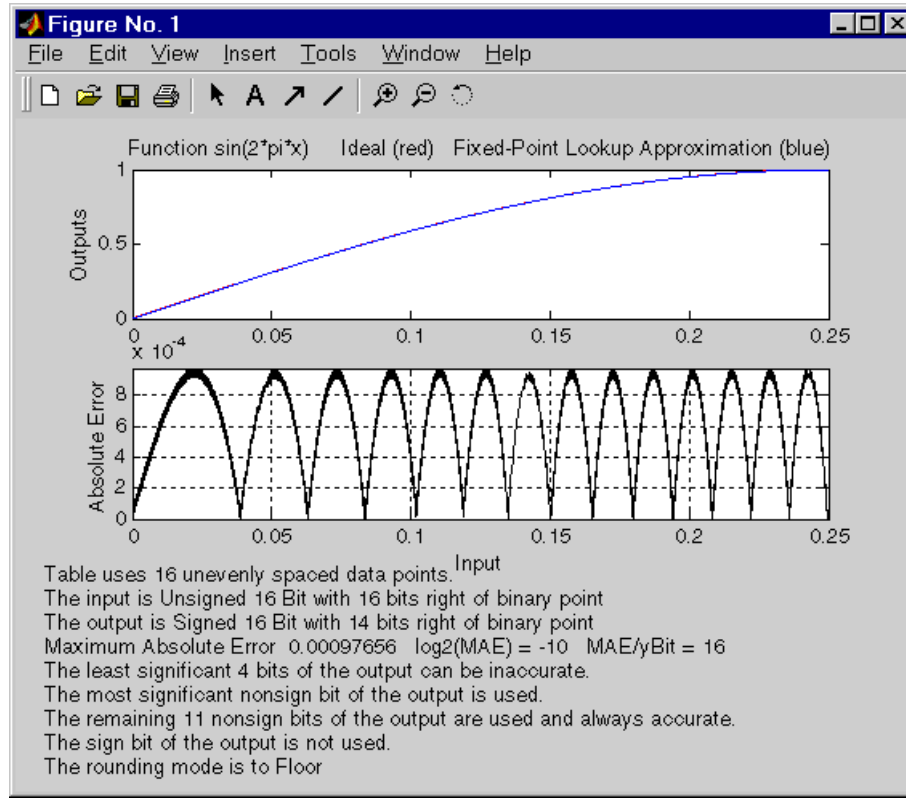
```
errworst =
  9.7656e-004
```

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 $\sin(2\pi x)$ 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 7-8, 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
```

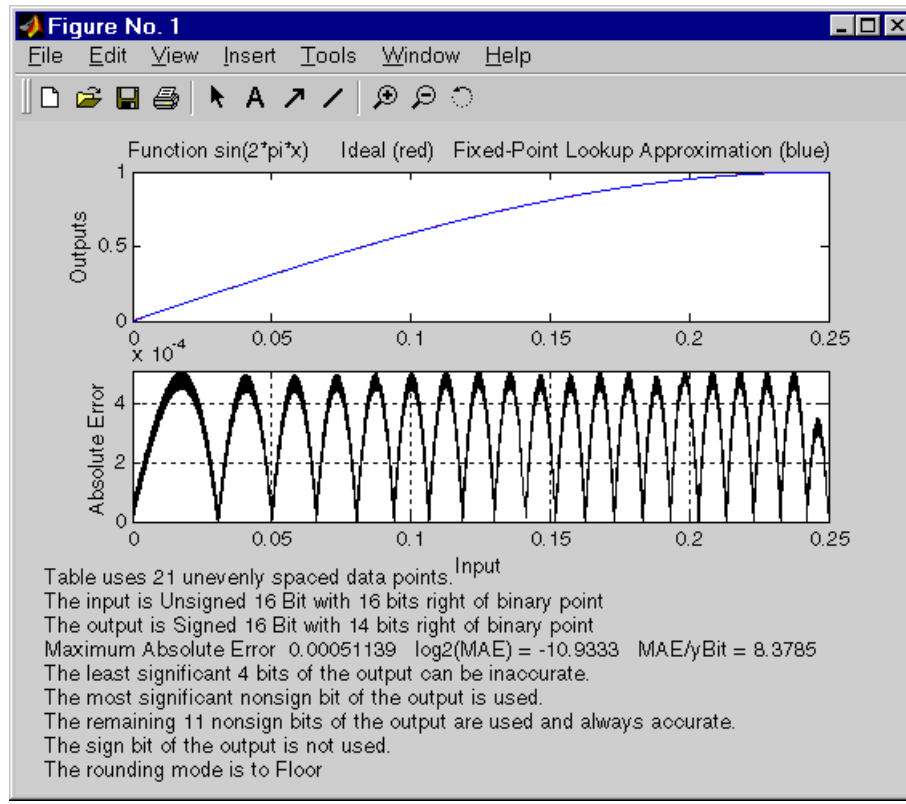
Plotting the Results

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 "Effect of Spacing on Speed, Error, and Memory Usage" on page 7-23 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 7-8, 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);
```

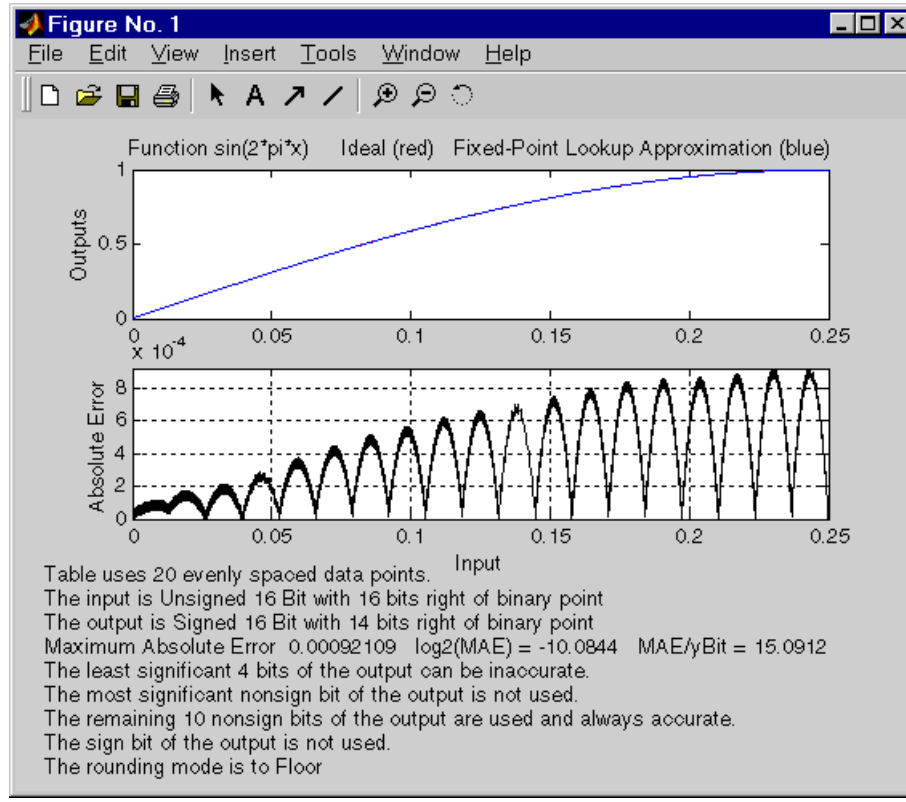
You can find the number of points in the lookup table by typing `length(xdata)`:

```
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 7-8, if you have not already done so in this MATLAB session.

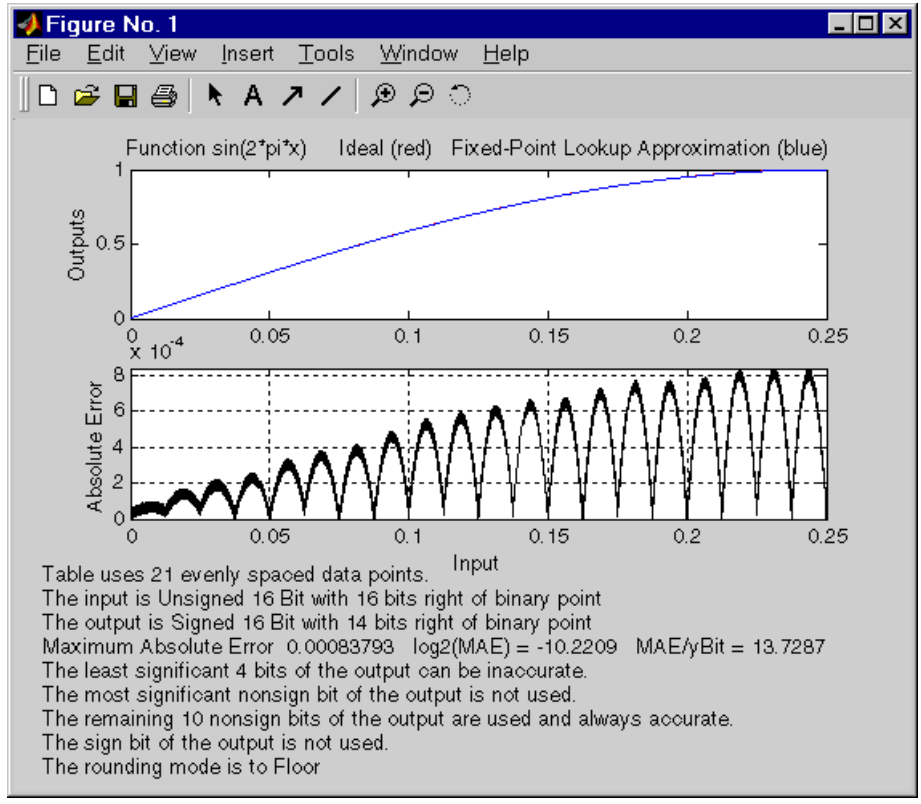
Next, at the MATLAB prompt type

```
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 7-8, 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
```

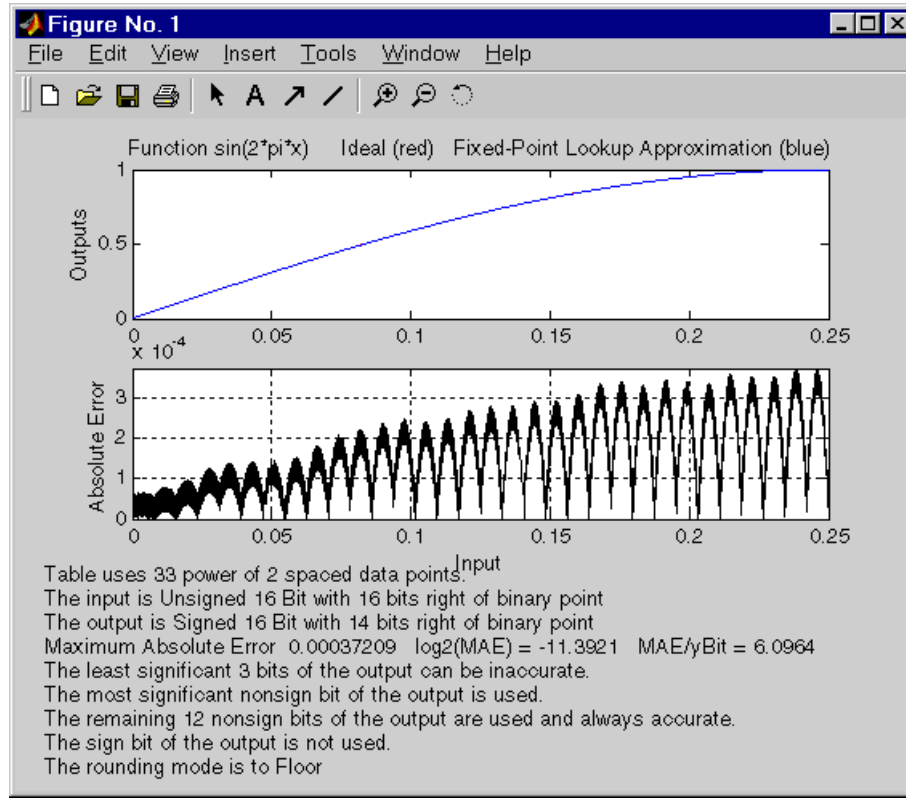
```
errworst =
  3.7209e-004
```

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 7-8, if you have not already done so in this MATLAB session:

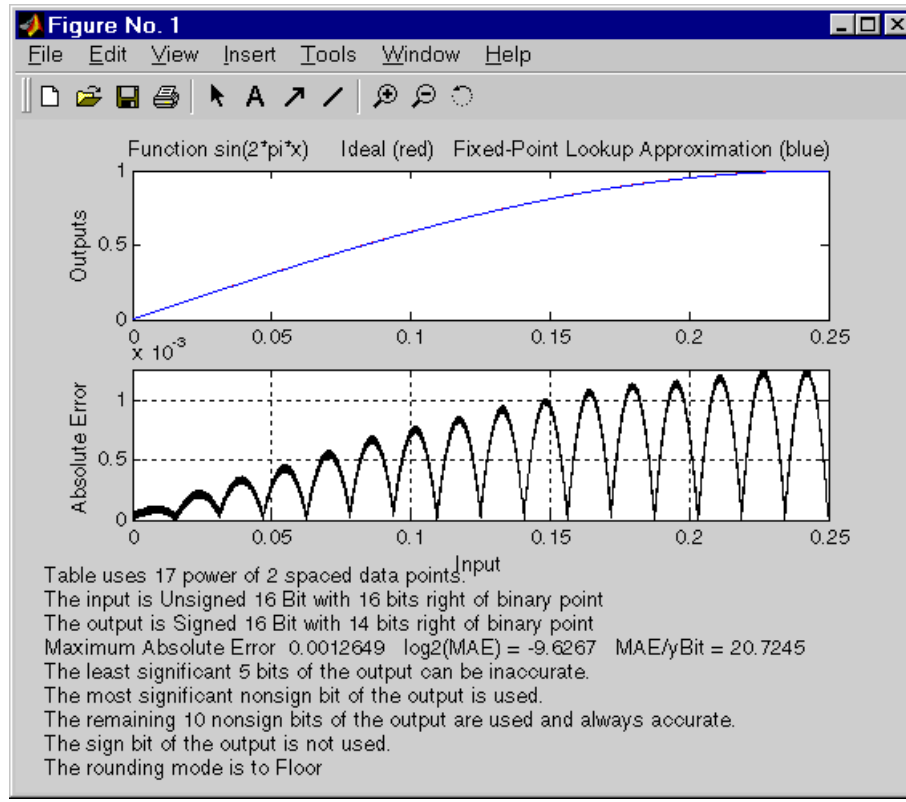
```
spacing = 'pow2';
[xdata, errworst] = fixpt_look1_func_approx(funcstr, ...
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,[],nptsmax,spacing);
```

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 7-8, 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,numptsmax);
```

The results for three different settings for `numptsmax` are as follows:

- `numptsmax=33` — The function creates the lookup table with 33 points having power of two spacing as in Example 3.
- `numptsmax=21` — Because the `errmax` and `numptsmax` 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.
- `numptsmax=16` — Because the `errmax` and `numptsmax` 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.

Comparing the Examples

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: Using the Lookup Table Functions

The following summarizes how to use the lookup table approximation functions:

1. Define
   a. The ideal function to be approximated
   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.

Effect of Spacing on Speed, Error, and Memory Usage

In this section...

"Introduction" on page 7-23
"Data ROM Required" on page 7-24
"Determining Out-of-Range Inputs" on page 7-25
"Determining Input Location" on page 7-25
"Interpolation" on page 7-27
"Conclusion" on page 7-29

Introduction

The sections that follow compare implementations of lookup tables that use breakpoints whose spacing is uneven, even, and power of two. This comparison is valid only when the breakpoints are not tunable. If the breakpoints can be tuned in the generated code, then all three cases generate the same code. The comparison focuses on the amount of read-only memory (ROM) used for data, the amount of ROM used for commands, and the speed with which the commands are executed.

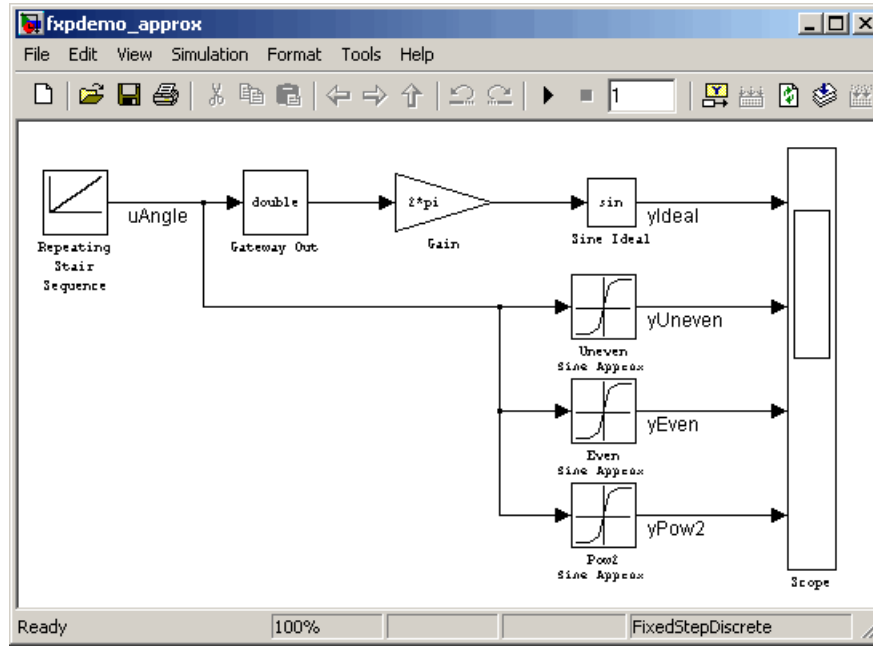
As a specific example, this comparison uses the demo `fxpdemo_approx_sin`. There are three fixed-point lookup tables in this model. All three lookup tables approximate the function `sin(2*pi*u)` over the first quadrant. All three achieve a worst-case error of less than `2^-8`. However, they have different restrictions on their breakpoint spacing.

You can use the model `fxpdemo_approx`, which this demo opens, to generate code with the Real-Time Workshop® product (Real-Time Workshop software license required). The sections that follow present several segments of the generated code. These segments of code are edited and arranged for clarity and to emphasize key differences.

To open the demo, type at the MATLAB® prompt

```
fxpdemo_approx_sin
```

This opens the following model.



Data ROM Required

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 needed.

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 needed.

Determining Out-of-Range Inputs

In all three cases you have to guard against the possibility 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.

Determining 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 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 subtraction is optimized out.

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 is located with respect to the breakpoints, the unevenly spaced case clearly requires much more code than the other two cases. This code requires additional command ROM. This ROM penalty can be reduced if many lookup tables share the binary search algorithm as a

function. 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, then 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 cased 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 to be used, 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. Quite a few processors do not provide a division instruction. Division on these processors is implemented via repeated subtractions. This is slow and requires a fair amount of machine code, but this code can be shared.

Most processors provide a way to do logical and arithmetic shifts left and right. A distinguishing 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 or not 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 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

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, i.e., even spacing, some

simplification is possible. If spacing is not just even but also a power of two, then very 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 can be somewhat involved and depend on the target processor. For example, these routines look quite different for a 16-bit processor than for a 32-bit processor.

Even Case

Evenly spaced breakpoints implement interpolation using just slightly different calculations than the uneven case. The key difference is that the calculations do not directly use the breakpoints. This means the breakpoints are not required in ROM, which can be a very significant savings:

```
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 uneven 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 very significant advantages over the uneven and even implementations. The key difference is that a subtraction and a division are replaced by a bitwise AND combined with a shift right at the end of the multiplication. Another advantage is that the term `(u - xData[iLeft] ) / ( xData[iRght] - xData[iLeft])` is computed with 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.

Conclusion

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 amount of advantage provided by this depends on the specific hardware, but you would 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.

8

Code Generation

Overview (p. 8-2)	Provides an overview of generating code from models using fixed-point blocks
Code Generation Support (p. 8-3)	Discusses the simulation features supported by code generation in the Simulink® Fixed Point™ software
Accelerating Fixed-Point Models (p. 8-6)	Information about using acceleration modes to increase the speed of some fixed-point models
Using External Mode or Rapid Simulation Target (p. 8-8)	Information on errors that can occur when using the Real-Time Workshop® external mode or rapid simulation target with fixed-point models
Optimizing Your Generated Code (p. 8-10)	Tips to help you optimize your generated code to reduce ROM or model execution time
Optimizing Your Generated Code with the Model Advisor (p. 8-15)	Discusses optimizing your fixed-point code with the help of the Model Advisor

Overview

You can generate C code with the Simulink® Fixed Point™ software by using the Real-Time Workshop® and Stateflow® Coder™ products. 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 Real-Time Workshop and Stateflow 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 nonreal-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 Real-Time Workshop product and acceleration modes in the Simulink® software are examples of where nonreal-time code is generated and run.

Code Generation Support

In this section...
“Introduction” on page 8-3
“Languages” on page 8-3
“Storage Class of Variables” on page 8-3
“Storage Class of Parameters” on page 8-4
“Rounding Modes” on page 8-4
“Overflow Handling” on page 8-4
“Blocks” on page 8-4
“Scaling” on page 8-5

Introduction

All fixed-point blocks support code generation, with the exception of 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.

Storage Class of Variables

- Fixed-point code generation can handle variables that both match and do not match the target compiler sizes for `char`, `short`, `int`, or `long` data types. Code generation supports any variable having a width less than or equal to the size of a `long` data type or 32 bits, whichever is smaller. For example, consider a compiler that defines a `long` data type to be 40 bits. On such a compiler, code generation supports only those variables that range between 1 and 32 bits.

  This capability is particularly useful if you want to

  - Create a prototype on a target chip, but use a different target chip for production

  - Provide bit-true simulation in a rapid prototyping environment for odd data type sizes used by FPGAs, ASICs, 24-bit DSPs, and so on
- Fixed-point code generation supports floating-point variables.

Note For information on how to specify the bit sizes of your target in the **Hardware Implementation** pane of the Configuration Parameters dialog, refer to "Configuring Hardware Properties" in the *Real-Time Workshop® User's Guide*.

Storage Class of Parameters

- The Real-Time Workshop product requires that parameters be 1 to 32 bits, either signed or unsigned. The parameter size must also be compatible with the target C compiler.
- Fixed-point code generation supports floating-point parameters.

Rounding Modes

All five rounding modes—`Zero`, `Nearest`, `Ceiling`, `Floor`, and `Simplest`—are supported.

Overflow Handling

- Saturation and wrapping are supported.
- Wrapping generates the most efficient code.
- Currently, you cannot choose to automatically exclude saturation code when hardware saturation is available. You must select wrapping in order for the Real-Time Workshop product to exclude saturation code.

Blocks

All blocks generate code for all operations with a few exceptions. The Lookup Table, Lookup Table (2-D), and Lookup Table Dynamic blocks generate 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

You can use Simulink® acceleration modes with your fixed-point model if the model meets the code generation restrictions. 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.

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 nonreal-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 a variety of situations. In an acceleration mode, code is generated that is specifically 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 will be faster. If the stop time is increased, then an acceleration mode will have an increasing speed advantage. Eventually, the acceleration mode speed advantage will be 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 very 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 *Using Simulink* for more information.

Using External Mode or Rapid Simulation Target

In this section...

"Introduction" on page 8-8

"External Mode" on page 8-8

"Rapid Simulation Target" on page 8-9

Introduction

If you are using the Real-Time Workshop® external mode or rapid simulation (rsim) target (see *Real-Time Workshop 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" on page 8-10
"Restrict Data Type Word Lengths" on page 8-11
"Avoid Fixed-Point Scalings with Bias" on page 8-11
"Wrap and Round to Floor or Simplest" on page 8-11
"Limit the Use of Custom Storage Classes" on page 8-13
"Limit the Use of Unevenly Spaced Lookup Tables" on page 8-13
"Minimize the Variety of Similar Fixed-Point Utility Functions" on page 8-14

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 8-11	Yes	Yes
"Avoid Fixed-Point Scalings with Bias" on page 8-11	Yes	Yes
"Wrap and Round to Floor or Simplest" on page 8-11	Yes	Yes
"Limit the Use of Custom Storage Classes" on page 8-13	Yes	No
"Limit the Use of Unevenly Spaced Lookup Tables" on page 8-13	Yes	Yes
"Minimize the Variety of Similar Fixed-Point Utility Functions" on page 8-14	Yes	No

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 3-23 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 hand-written 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 simply shifted right. 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 hand-written 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. If you know how your target handles rounding for signed division, entering this information on the **Hardware Implementation** pane of the Configuration Parameters dialog box improves efficiency. For Product blocks that are only doing division, setting the **Round integer calculations toward** parameter to the known rounding mode of your target gives the best results. You can also use the `Simplest` rounding mode on blocks where it is available. For more information, refer to "Simplest Rounding" on page 3-11.

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 hand 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 hand-written 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 hand 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 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 hand code.

Limit the Use of Custom Storage Classes

In addition to the tip mentioned in "Wrap and Round to Floor or Simplest" on page 8-11, to obtain the maximum benefits of expression folding you also need to make sure that the **RTW 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 Real-Time Workshop® Embedded Coder™ product generates fixed-point utility functions that are designed to handle specific situations efficiently. The Real-Time Workshop® 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.

Optimizing Your Generated Code with the Model Advisor

In this section...

"Introduction" on page 8-15
"Optimize Lookup Table Data" on page 8-16
"Reduce Cumbersome Multiplications" on page 8-16
"Optimize the Number of Multiply and Divide Operations" on page 8-17
"Reduce Multiplies and Divides with Nonzero Bias" on page 8-18
"Eliminate Mismatched Scaling" on page 8-18
"Minimize Internal Conversion Issues" on page 8-20
"Use the Most Efficient Rounding" on page 8-22

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 **Task Hierarchy** pane, expand the **By Product** node and select **Real-Time Workshop 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 "Effect of Spacing on Speed, Error, and Memory Usage" on page 7-23. 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 3-39 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 in the 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. These 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 placed 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. These 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 placed 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

You can specify rounding options for fixed-point operations both with the **Round integer calculations toward** parameter on many block masks, and with the **Signed integer division rounds to** parameter on the **Hardware Implementation** pane of the Configuration Parameters dialog box. Traditional hand-written 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 simply shifted right. 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 hand-written 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. If you know how your target handles rounding for signed division, entering this information in the **Hardware Implementation** pane of the Configuration Parameters dialog box improves efficiency. For Product blocks that are only doing division, setting the **Round integer calculations toward** parameter to the known rounding mode of your target gives the best results. You can also use the `Simplest` rounding mode on blocks where it is available. For more information, refer to "Simplest Rounding" on page 3-11. The Model Advisor alerts you when rounding optimizations are available with messages like the following:

- To obtain the most efficient generated code, you should change the **Round integer calculations toward** parameter of the following block to either `Floor` or `Simplest`.

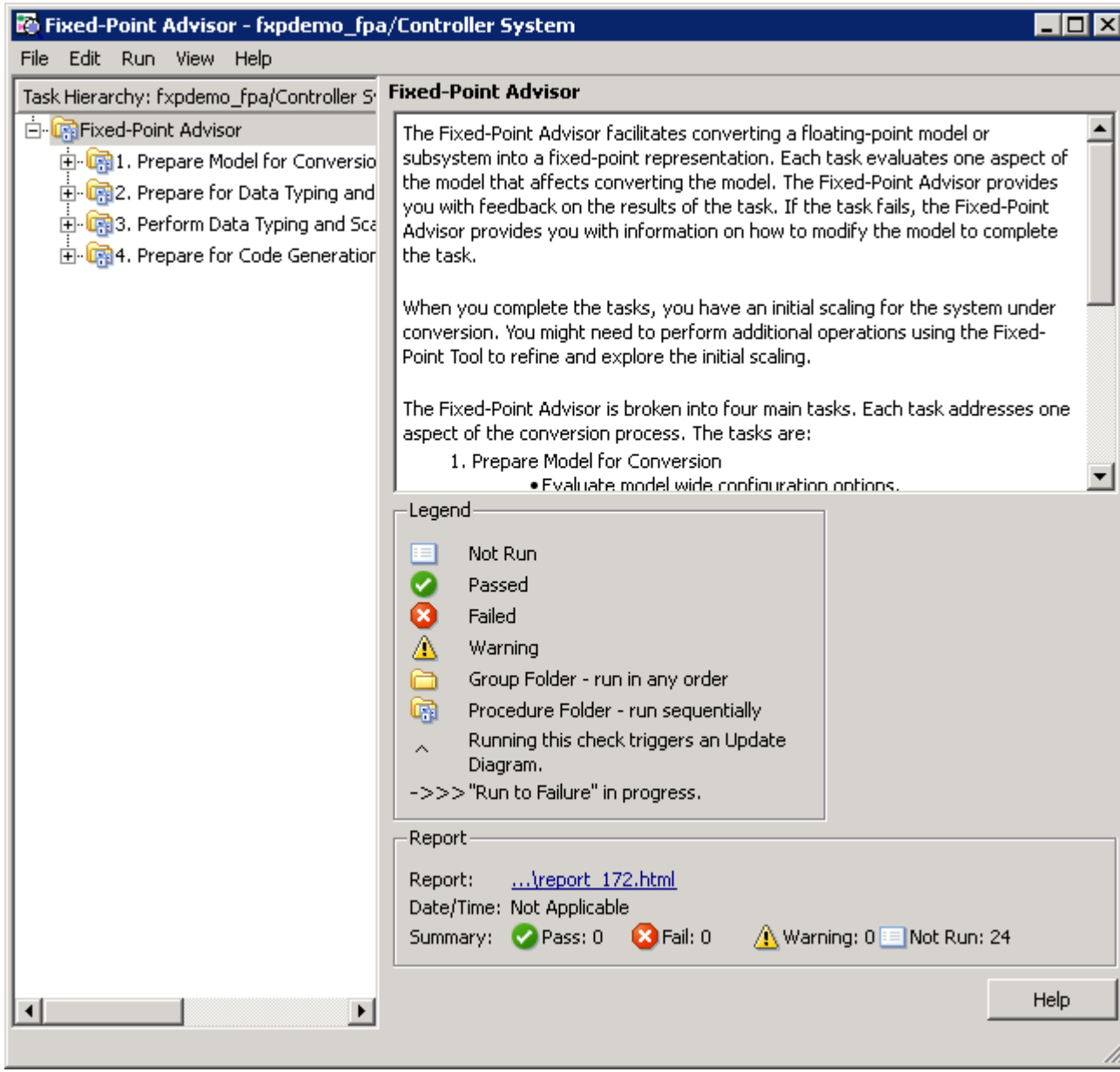
- Integer division generated code could be more efficient. The rounding behavior of signed integer division is not fully specified by the C language standards. When faced with this lack of specification, the code generated for division can be large in order to ensure bit-true agreement between simulation and code generation. The **Hardware Implementation** pane of the Configuration Parameters dialog allows you to specify the rounding behavior of signed integer division for the Embedded Hardware. For this model, the rounding behavior is currently set to `Undefined`. You can reduce the size of the code generated for division by determining and setting this information. The most common behavior is that signed integer division rounds to `Zero`.

9

Fixed-Point Advisor Reference

Fixed-Point Advisor (p. 9-2)	Introduces the Fixed-Point Advisor
Prepare Model for Conversion (p. 9-6)	Validate model settings
Prepare for Data Typing and Scaling (p. 9-18)	Set block configuration options and define block design minimum and maximum values
Perform Data Typing and Scaling (p. 9-24)	Specify data types and scaling
Prepare for Code Generation (p. 9-47)	Verify model configured for code generation optimizations

Fixed-Point Advisor



Fixed-Point Advisor Overview

The Fixed-Point Advisor is a tool you can use to guide you through the process of converting your model from an unknown floating-point data type to a known fixed-point data type.

Description

Use the Fixed-Point Advisor to:

- Set model-wide configuration options.
- Set block-specific dialog parameters.
- Set the initial scaling for the model.
- Validate the results of the initial scaling against the floating-point model.

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**.

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.

Tips

- The Fixed-Point Advisor is intended to assist in converting small models. Larger models can result in long processing 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 so that you can identify where errors are introduced accidentally.
- Isolate the system under conversion by placing Data Conversion blocks on the inputs and outputs of the system.
- Convert subsystems within your model, rather than the entire model. This saves time and unnecessary conversions.

See Also

- “Working With the Fixed-Point Advisor” on page 5-2
- “Tutorial: Converting a Model from Floating- to Fixed-Point” on page 5-5

Prepare Model for Conversion

In this section...

"Prepare Model for Conversion Overview" on page 9-7

"Verify model simulation settings" on page 9-8

"Address unsupported blocks" on page 9-9

"Verify update diagram status" on page 9-10

"Set up signal logging" on page 9-11

"Create simulation reference data" on page 9-12

"Verify hardware selection" on page 9-13

"Verify Fixed-Point Conversion Guidelines Overview" on page 9-14

"Check model configuration data validity diagnostic parameters settings" on page 9-15

"Implement logic signals as Boolean data" on page 9-16

"Check for proper bus usage" on page 9-17

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
- “Tutorial: Converting a Model from Floating- to Fixed-Point” on page 5-5

Verify model simulation settings

Validate that model logging options are consistent with fixed-point conversion goals.

Description

Ensures that fixed-point data can be logged in downstream tasks.

Analysis Results and Recommended Actions

Conditions	Recommended Action
The following Fixed-Point Tool settings are not set to the correct values: • **Overwrite or merge results** • **Logging mode** • **Data type override**	Set the following: • **Overwrite or merge results** to `Overwrite` • **Logging mode** to `Minimums, maximums and overflows` • **Data type override** to `Force off`
The model Configuration Parameters **Data Import/Export** > **Signal logging** check box is `off`.	Set to `on`
The `fipref DataTypeOverride` property is not set to `ForceOff`.	Set `DataTypeOverride` to `ForceOff`

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 6-20
- “Signal logging”
- “Using fipref Objects to Set Data Type Override Preferences”

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.

Analysis Results and Recommended Actions

Conditions	Recommended Action
Blocks that do not support fixed-point data types exist in model.	• Decouple the block by right-clicking the block and selecting **Insert Data Type Conversion** > **All Ports**. • Replace the block with the block specified in the Result pane.

Verify update diagram status

Verify update diagram succeeds.

Description

A model update diagram action is necessary for most down stream tasks.

Analysis 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

See “Updating a Block Diagram” in the Simulink® User’s Guide.

Set up signal logging

Specify at least one signal of interest to log during simulation.

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.	Specify at least one signal to be logged.

Action Result

Clicking **Modify All** enables logging on unique signals. The total number of signals logged is displayed.

Tips

Log inports and outports of the system under conversion.

Create simulation reference data

Simulate the model using the current solver settings, and create and archive reference signal data to use for comparison and analysis.

Description

Simulate the model using the current solver settings, create and archive reference signal data to use for comparison and analysis in downstream tasks.

Analysis Results and Recommended Actions

Conditions	Recommended Action
Simulation does not run	Fix errors so simulation will run.
Simulation did not generate output data because no signals have signal data logging enabled.	Enable signal data logging on at least one signal.

Tips

- If the simulation is set up to have a long simulation time, after starting the run of this check 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 hardware selection

Verify target hardware setting.

Description

Review the hardware device settings and verify they are the settings you intend to use.

Analysis Results and Recommended Actions

Conditions	Recommended Action
The model **Configuration Parameters** > **Hardware Implementation** device parameters are not specified.	Specify the **Configuration Parameters** > **Hardware Implementation** > **Device vendor** and **Device type** parameters.

See Also

- “Device type”
- “Device vendor”

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
- “Tutorial: Converting a Model from Floating- to Fixed-Point” on page 5-5

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 checks.

Analysis 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 the diagnostic data validity parameters to `warning` in Configuration Parameters > **Diagnostics** > **Data Validity** > **Parameters**

Action Results

Clicking **Modify All** sets the model **Configuration Parameters** > **Diagnostics** > **Data Validity** parameters to `warning`. A table displays the current and previous Data Validity parameter 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.

Analysis 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 (vs. double)** to `on`.

Action Results

Clicking **Modify All** selects the model **Configuration Parameters > Optimization > Implement logic signals as boolean data (vs. double)** 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 used as bus creators
- Bus signals treated as vectors in the top-level model

Analysis 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 signals treated as vector** to `error`.

Prepare for Data Typing and Scaling

In this section...
"Prepare for Data Typing and Scaling Overview" on page 9-19
"Remove output data type inheritance" on page 9-20
"Relax input data type settings" on page 9-21
"Verify Stateflow charts have strong data typing with Simulink" on page 9-22
"Specify block minimum and maximum values" on page 9-23

Prepare for Data Typing and Scaling Overview

Configures 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
- "Tutorial: Converting a Model from Floating- to Fixed-Point" on page 5-5

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. To prepare for scaling, replace inherited values with the current propagated data types.

Note This task is preparation for scaling, not actual scaling.

Analysis Results and Recommended Actions

Conditions	Recommended Action
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. A table lists 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.

Analysis Results and Recommended Actions

Conditions	Recommended Action
The input data types of blocks or Stateflow input data 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 input data types by setting the `Type` method to `Inherited`.

Action Results

Clicking **Modify All** explicitly configures input data types to the recommended value where possible. A table lists the previous and current data types for the reconfigured blocks.

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

Note This task is preparation for scaling, not actual scaling.

Analysis 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.

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.

Analysis 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.

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.

See Also

See “Model Advisor Result Explorer” in the Simulink User’s Guide.

Perform Data Typing and Scaling

In this section...

"Perform Data Typing and Scaling Overview" on page 9-25
"Propose scaling for Inport blocks" on page 9-26
"Propose scaling for Constant blocks" on page 9-30
"Propose scaling for blocks" on page 9-34
"Resolve scaling conflicts" on page 9-39
"Summarize data types" on page 9-41
"Propose scaling for parameters" on page 9-42
"Check for numeric errors" on page 9-45
"Analyze logged signals" on page 9-46

Perform Data Typing and Scaling Overview

Specify block data type information. Propose scaling for Simulink® blocks.

Description

This folder contains the tasks to assist you in performing data typing and scaling.

See Also

- “Working With the Fixed-Point Advisor” on page 5-2
- “Tutorial: Converting a Model from Floating- to Fixed-Point” on page 5-5

Propose scaling for Inport blocks

Provide fixed-point data type scaling for Inport blocks with floating-point data types.

Description

The Fixed-Point Advisor uses the type provided in the **Default Inport data type** menu with the output minimum and maximum values to provide recommended scaling for Inport blocks, as described in the table below.

How Scaling is Determined

Rationale	Output min / max available?	Fixed-point data type?	Word length / Fraction length	Overflow / Precision
Maintained current settings; overflow is possible.	No	Yes	Keep the current word and fraction lengths.	Overflow is possible.
Maintained current settings; overflow is not possible.	Yes	Yes	The calculated fraction length (based on the current word length and output minimum and maximum values) is greater than or equal to the current fraction length. Use the current word and fraction lengths.	Overflow is not possible, and there is no precision loss.

How Scaling is Determined (Continued)

Rationale	Output min / max available?	Fixed-point data type?	Word length / Fraction length	Overflow / Precision
Reduced precision for current word length to satisfy minimum and maximum range without overflow.	Yes	Yes	The calculated fraction length (based on the current word length and output minimum and maximum values) is less than the current fraction length. Use the current word length and the calculated fraction length.	Overflow is not possible, however precision loss is possible.
Converted using input parameter word length and minimum and maximum values; overflow is not possible.	Yes	No	Use the input parameter word length and the calculated fraction length (based on the input parameter word length and output minimum and maximum values).	Overflow is not possible.
All elements of bus objects have fixed-point types. Maintained current settings.	N/A	Yes	Keep the current word and fraction lengths.	Overflow is possible.
All elements of bus objects must have fixed-point types. Manually change the bus object elements to fixed-point types and resolve data type conflicts.	N/A	No	You must specify the word and fraction lengths manually.	N/A

Input Parameters

Default inport data type

Enter a default fixed-point data type to use for Inport blocks, or select one from the menu:

```
undefined
int8
uint8
int16
uint16
int32
uint32
fixdt(1,16,4)
```

Reported Conditions and Recommended Actions

Conditions	Recommended Action
The input parameter is invalid.	In the **Input Parameters > Default inport data type** menu, enter or select a valid value.
You haven't specified minimum and maximum values for all Inport blocks with a floating-point data type.	Specify minimum and maximum values for all Inport blocks with a floating-point data type.
The current data type and scaling is not optimal.	Change the data type and scaling based on the information provided in the Action Result pane.
Elements of bus objects have floating-point data types.	Manually change the bus object elements to fixed-point data types and resolve data type conflicts.

Action Results

Clicking **Modify All** applies the recommended scaling to the Inport blocks.

Tips

You can specify any **Default Inport data type** by typing in a value.

See Also

“Perform Data Typing and Scaling” on page 5-18

Propose scaling for Constant blocks

Provide fixed-point data type scaling for Constant blocks with floating-point data types.

Description

The Fixed-Point Advisor uses the type provided in the **Constant data type** menu with either the output minimum and maximum, or the Constant block values to provide recommended scaling for Constant blocks, as described in the table below.

How Scaling is Determined

Rationale	Output min / max available?	Constant value available?	Fixed-point data type?	Word length / Fraction length	Overflow / Precision
Maintained current settings; overflow is not possible.	Yes	NA	Yes	The calculated fraction length (based on the current word length and output minimum and maximum values) is greater than or equal to the current fraction length. Use the current word and fraction lengths.	Overflow is not possible, and there is no precision loss.

How Scaling is Determined (Continued)

Rationale	Output min / max available?	Constant value available?	Fixed-point data type?	Word length / Fraction length	Overflow / Precision
Reduced precision for current word length to satisfy minimum and maxaximum range without overflow.	Yes	NA	Yes	The calculated fraction length (based on the current word length and output minimum and maximum values) is less than the current fraction length. Use the current word length and the calculated fraction length.	Overflow is not possible, however precision loss is possible.
Maintained current settings; Constant value overflow is possible.	No	Yes	Yes	The calculated fraction length (based on the current word length and constant value) is greater than or equal to the current fraction length. Use the current word and fraction lengths.	Overflow is possible, and there is no precision loss.

How Scaling is Determined (Continued)

Rationale	Output min / max available?	Constant value available?	Fixed-point data type?	Word length / Fraction length	Overflow / Precision
Reduced precision for current word length to satisfy Constant value; Constant value overflow is possible.	No	Yes	Yes	The calculated fraction length (based on the current word length and constant value) is less than the current fraction length. Use the current word length and the calculated fraction length.	Overflow is possible, and precision loss is possible.
Converted using input parameter word length and output minimum and maximum values; overflow is not possible.	Yes	NA	No	Use the input parameter word length and the calculated fraction length (based on the input parameter word length and output minimum and maximum values).	Overflow is not possible.
Converted using input parameter word length and Constant value; Constant value overflow is possible.	No	Yes	No	Use the input parameter word length and the calculated fraction length (based on the input parameter word length and constant value).	Overflow is possible.

Input Parameters

Constant data type

Enter a default fixed-point data type to use for Constant blocks, or select one from the menu:

```
undefined
int8
uint8
int16
uint16
int32
uint32
fixdt(1,16,4)
```

Analysis Results and Recommended Actions

Conditions	Recommended Action
The input parameter is invalid.	In the **Input Parameters > Constant data type** menu, enter or select a valid value.
The current data type and scaling is not optimal.	Change the data type and scaling based on the information provided in the Action Result pane.

Action Results

Clicking **Modify All** applies the recommended scaling to the Constant blocks.

Tips

You can specify any **Constant data type** by typing in a value.

See Also

"Perform Data Typing and Scaling" on page 5-18

Propose scaling for blocks

Provide fixed-point data type scaling for nonsource Simulink blocks that have floating-point data types.

Description

The Fixed-Point Advisor uses the fixed-point data types provided in the Input Parameters menus with either the output or simulation minimum and maximum values to provide recommended scaling, as described in the table below.

How Scaling is Determined

Rationale	min / max available?	Fixed-point data type?	Word length / Fraction length	Overflow / Precision
Maintained current settings; overflow is possible.	No	Yes	Keep the current word and fraction lengths.	Overflow is possible.
Maintained current settings; overflow is not possible.	Yes	Yes	The calculated fraction length (based on the current word length and minimum and maximum values) is greater than or equal to the current fraction length. Use the current word and fraction lengths.	Overflow is not possible, and there is no precision loss.

How Scaling is Determined (Continued)

Rationale	min / max available?	Fixed-point data type?	Word length / Fraction length	Overflow / Precision
Reduced precision for current word length to satisfy minimum and maximum range without overflow.	Yes	Yes	The calculated fraction length (based on the current word length and minimum and maximum values) is less than the current fraction length. Use the current word length and the calculated fraction length.	Overflow is not possible, however precision loss is possible.
Converted using input parameter word length and min/max values; overflow is not possible.	Yes	No	Use the input parameter word length and the calculated fraction length (based on the input parameter word length and minimum and maximum values).	Overflow is not possible. **Note** If the object has the **Saturate on integer overflow** parameter selected, and output minimum and maximum values are available, clear the **Saturate on integer overflow** parameter.
Converted using input parameter; overflow is possible.	No	No	Use the word and fraction lengths based on the input parameter. It is not possible to report better scaling without minimum and maximum values.	Overflow is possible. **Note** Select the **Saturate on integer overflow** parameter to aviod overflow.

How Scaling is Determined (Continued)

Rationale	min / max available?	Fixed-point data type?	Word length / Fraction length	Overflow / Precision
All elements of bus objects have fixed-point types. Maintained current settings.	N/A	Yes	Keep the current word and fraction lengths.	Overflow is possible.
All elements of bus objects must have fixed-point types. Manually change the bus object elements to fixed-point types and resolve data type conflicts.	N/A	No	You must specify the word and fraction lengths manually.	N/A

Input Parameters

Output data type

Enter a default fixed-point data type to use, or select one from the menu:

```
undefined
int8
uint8
int16
uint16
int32
uint32
fixdt(1,16,4)
```

Accumulator data type

Enter a default fixed-point data type to use, or select one from the menu:

```
undefined
```

```
int8
uint8
int16
uint16
int32
uint32
fixdt(1,16,4)
```

Product output data type

Enter a default fixed-point data type to use, or select one from the menu:

```
undefined
int8
uint8
int16
uint16
int32
uint32
fixdt(1,16,4)
```

Analysis Results and Recommended Actions

Conditions	Recommended Action
An input parameter is invalid.	In the specified **Input Parameters** menu, enter or select a valid value.
The current data type and scaling is not optimal.	• Provide output minimum and maximum values for blocks and rerun the task for better recommendations. • Change the data type and scaling based on the information provided in the Action Result pane.
Elements of bus objects have floating-point data types.	Manually change the bus object elements to fixed-point data types and resolve data type conflicts.

Action Results

Clicking **Modify All** applies the recommended scaling to the nonsource Simulink blocks.

Tips

You can specify any data type by typing a value in the Input parameters menus.

See Also

"Perform Data Typing and Scaling" on page 5-18

Resolve scaling conflicts

Resolve scaling conflicts introduced in previous tasks.

Description

Find and fix all scaling conflicts introduced in previous tasks, as described in the table below.

How Scaling is Determined

Rationale	Description
Shared scaling with other blocks.	There is downstream block that requires same data types for all inputs.

Analysis Results and Recommended Actions

Conditions	Recommended Action
Different scaling is recommended to avoid scaling conflicts.	Consider applying the recommendations or specifying desired scaling.
Current word length does not satisfy both range and precision for the input values in Lookup tables. Increasing the word length for the input data does not solve the problem or the word length can not be increased.	Check whether the input values are scaleable and increased the word length to satisfy both range and precision for the input values in the Lookup tables downstream. . Rerun task 3.3. and task 3.4.

Action Results

Clicking **Modify All** applies the recommended scaling.

Limitations

- When the input data loaded from the workspace different than the inport data type, MATLAB generates an error because the data types do not agree. You can fix this error by doing one of the following:
  - Disable loading external inputs from the workspace as described in the error message.
  - Change the data type of the input data to match the inport data type, if possible.
- If you automatically fix this task by clicking the **Modify All** button, and the Fixed-Point Advisor cannot automatically resolve a scaling conflict, when you rerun the task it fails and the Fixed-Point Advisor displays an error message in the Analysis result. Use the information in the error message to manually resolve this data type conflict. An example of this is a conflict due to a bus objects.

See Also

"Perform Data Typing and Scaling" on page 5-18

Summarize data types

Summarize the data types used in the system

Description

Provides a summary of the number of:

- Floating-point data types
- Integer word lengths less than or equal to the native word size
- Integer word lengths greater than the native word size

Analysis Result and Recommended Actions

Conditions	Recommended Action
There is at least one signal that does not have a fixed-point data type.	This summary is provided for your information, no action is required.

Tips

If you have decoupled a block in your model, for example using a Data Type Conversion block, this task never passes because the decoupled block is not converted to have a fixed-point data type.

Propose scaling for parameters

Provide fixed-point data type scaling for block parameters and Stateflow® Parameters and Constants that have floating-point data types.

Description

The Fixed-Point Advisor uses the fixed-point data types provided in the **Parameter data type** menu with either the parameter minimum and maximum values or value, or initial value to provide recommended scaling, as described in the table below.

How Scaling is Determined

Rationale	min / max available?	Fixed-point data type?	Word length / Fraction length	Overflow / Precision
Maintained current settings	No	Yes	Keep the current word and fraction lengths.	Overflow is possible.
Maintained current settings; min/max overflow is not possible.	Yes	Yes	The calculated fraction length (based on the current word length and parameter minimum and maximum values) is greater than or equal to the current fraction length. Use the current word and fraction lengths.	Overflow is not possible, and there is no precision loss.

How Scaling is Determined (Continued)

Rationale	min / max available?	Fixed-point data type?	Word length / Fraction length	Overflow / Precision
Reduced precision for current word length to satisfy min/max range without overflow.	Yes	Yes	The calculated fraction length (based on the current word length and parameter minimum and maximum values) is less than the current fraction length. Use the current word length and the calculated fraction length.	Overflow is not possible, however precision loss is possible.
Inherit via internal rule	No	No	Inherit the word and fraction lengths based on block internal rules.	NA

Input Parameters

Parameter data type

Enter a default fixed-point data type to use for parameters, or select one from the menu:

```
undefined
int8
uint8
int16
uint16
int32
uint32
fixdt(1,16,4)
```

Analysis Results and Recommended Actions

Conditions	Recommended Action
An input parameter is invalid.	In the **Input Parameters > Parameter data type** menu, enter or select a valid value.
The current data type and scaling is not optimal.	Change the data type and scaling based on the information provided in the Action Result pane.

Action Results

Clicking **Modify All** applies the recommended scaling.

Tips

You can specify any **Parameter data type** by typing in a value.

See Also

"Perform Data Typing and Scaling" on page 5-18

Check for numeric errors

Summarize simulation numerical errors.

Description

Numeric errors logged are:

- Overflow
- Saturation
- Parameter saturation
- Division by zero

Analysis Results and Recommended Actions

Conditions	Recommended Action
The Fixed-Point Advisor found a numeric error.	Resolve numeric errors.

Analyze logged signals

Summarize the differences between current and reference simulation data.

Description

View differences between the baseline model and converted model to verify the model is accurate after the conversion completes. A simulation of the model occurs during this task creating the current (Active) simulation data. You created the reference simulation data (Reference) in "Create simulation reference data" on page 9-12. You can view the differences by clicking the *View Results* link in the Analysis Result box.

Input Parameters

Max number of Results

Specify the maximum number of signals to summarize.

Analysis Results and Recommended Actions

Conditions	Recommended Action
No signals are logged.	Enable signal logging on at least one line.

Prepare for Code Generation

In this section...

"Prepare for Code Generation Overview" on page 9-48

"Disable signal logging" on page 9-49

"Identify blocks that generate expensive saturation and rounding code" on page 9-50

"Identify questionable fixed-point operations" on page 9-51

Prepare for Code Generation Overview

Identify settings that might lead to nonoptimal results in code generation.

Limitations

All tasks in this folder require a Real-Time Workshop® license.

See Also

- "Working With the Fixed-Point Advisor" on page 5-2
- "Tutorial: Converting a Model from Floating- to Fixed-Point" on page 5-5

Disable signal logging

Disables unnecessary signal logging.

Description

Disabling unnecessary signal logging avoids declaring extra signal memory in generated code.

Analysis Results and Recommended Actions

Conditions	Recommended Action
Signals are logged.	Disable signal logging on all signals.

Action Results

Clicking **Modify All** disables signal logging on all logged signals.

Limitations

This task requires a Real-Time Workshop license.

Identify blocks that generate expensive saturation and rounding code

Check for blocks that generate expensive saturation or rounding code.

Description

- Setting the **Saturate on integer overflow** parameter can produce condition-checking code that your application might not require.
- Generated rounding code is inefficient because of **Round integer calculations toward** parameter setting.

Analysis Results and Recommended Actions

Conditions	Recommended Action
Blocks generate expensive saturation code.	Check each block to ensure that your application requires setting **Function Block Parameters** > **Signal Attributes** > **Saturate on integer overflow**. Otherwise, clear the **Saturate on integer overflow** parameter to ensure the most efficient implementation of the block in the generated code.
Generated code is inefficient.	Set the **Function Block Parameters** > **Round integer calculations toward** parameter to the recommended value.

Limitations

This task requires a Real-Time Workshop license.

Identify questionable fixed-point operations

Identify fixed-point operations that can lead to nonoptimal results.

Description

The following operations can lead to nonoptimal results:

- Division
  - The rounding behavior of signed integer division is not fully specified by C language standards. Therefore, the generated code for division is large to ensure bit-true agreement between simulation and code generation.
  - Integer division generated code contains protection against arithmetic exceptions such as division by zero, INT_MIN/-1, and LONG_MIN/-1. If you construct models making it impossible for exception triggering input combinations to reach a division operation, the protection code generated as part of the division operation is redundant.
  - The index search method `Evenly-spaced points` requires a division operation, which can be computationally expensive.
- Multiplication
  - Product blocks are configured to do more than one division operation. Multiplying all the denominator terms together first, and then computing only one division operation improves accuracy and speed in floating-point and fixed-point calculations.
  - Product blocks are configured to do more than one multiplication or division operation. Using several blocks, with each block performing one multiplication or one division operation, allows you 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.
  - Blocks that have the **Saturate on integer overflow** parameter selected, and have an ideal multiplication product with a larger integer size than the target integer size, must determine the ideal product in generated C code. The C code required to do this multiplication is large and slow.

  - Blocks with relative scaling of inputs and outputs must determine the ideal product in the generated C code. The C code required to do this multiplication is large and slow.
  - Blocks that multiply signals with nonzero bias require extra steps to implement the multiplication. Inserting Data Type Conversion blocks remove the biases, and allow you to control data type and scaling for intermediate calculations. The conversion is done once and all blocks in the subsystem benefit from simpler, bias-free math.
  - Blocks are multiplying signals with mismatched fractional slopes. This mismatch causes the overall operation to involve two multiply instructions.
  - the Real-Time Workshop software generates a reciprocal operation followed by a multiply operation for Product blocks that have a divide operation for the first input, and a multiply operation for the second input. If you reverse the inputs so that the multiplication occurs first and the division occurs second, the Real-Time Workshop software generates a single division operation for both inputs.
  - An input with an invariant constant value is used as the denominator in an online division operation. If the operation is changed to multiplication, and the invariant input is replaced by its reciprocal, then the division is done offline and the online operation is multiplication. This leads to faster and smaller generated code.

- Addition
  - 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 for the input and output exclude their biases.
  - Sum blocks can have a range error before an addition or subtraction operation. For simplicity of design, the Sum block always casts each input to the output data type and scaling before performing addition or subtraction. The input range is different than the output range, so a range error can occur when casting the input to the output data type.
  - A Sum block has an input with a fractional slope that does not equal the fractional slope of the output. This mismatch requires the Sum block to multiply the input by the net slope adjustment each time the input is converted to the output data type and scaling.

  - The net sum of the Sum block input biases does not equal the bias of the output. The generated code includes one extra addition or subtraction instruction to correctly account for the net bias adjustment.

- Using Relational Operator blocks
  - The data types of the Relational Operator block inputs are not the same. A conversion operation is required every time the block is executed. If one of the inputs is invariant, then changing the data type and scaling of the invariant input to match the other input improves the efficiency of the model.
  - The Relational Operator block inputs have different ranges, resulting in a range error when casting, and a precision loss each time a conversion is performed. You can insert Data Type Conversion blocks before the Relational Operator block to convert both inputs to a common data type that has sufficient range and precision to represent each input, making the relational operation error-free.
  - The inputs of the Relational Operator block have different fractional slopes. The mismatch causes the Relational Operator block to require a multiply operation each time the input with lesser positive range is converted to the data type and scaling of the input with greater positive range.
- Using MinMax blocks
  - The input and output of the MinMax block have different data types. A conversion operation is required every time the block is executed. The model is more efficient with the same data types.
  - The input of the MinMax block is converted to the data type and scaling of the output before performing a relational operation, resulting in a range error when casting, or a precision loss each time a conversion is performed.
  - The input of the MinMax block has a different fractional slope than the output. 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.
- Discrete-Time Integerator blocks have a complicated initial condition setting. The initial condition for the Discrete-Time Integrator blocks are

used to initialize the state and output. As a result, the output equation generates excessive code and an extra global variable is required.

Analysis Results and Recommended Actions

Conditions	Recommended Action
Integer division generated code is large.	Set the **Configuration Parameters > Hardware Implementation > Signed integer division rounds to** parameter to the recommended value.
Protection code generated as part of the division operation is redundant.	Verify that your model cannot cause exceptions in division operations and then remove redundant protection code by setting the **Configuration Parameters > Optimization > Remove code that protects against division arithmetic exceptions** parameter.
Generated code is inefficient.	Set the **Function Block Parameters > Round integer calculations toward** parameter to the recommended value.
Lookup table input data is not evenly spaced.	If the data is nontunable, adjust the table to be evenly spaced. See `fixpt_look1_func_approx`.
Lookup table input data is not evenly spaced when quantized, but it is very close to being evenly spaced.	If the data is nontunable, adjust 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 2.	If the data is nontunable, reimplement the table with even power-of-2 spacing. See `fixpt_look1_func_approx`.

Conditions	Recommended Action
Index search method is set to `Evenly-spaced points`.	Specify a different **Function Block Parameters** > **Index search method** to avoid the division operation.
Blocks require cumbersome multiplication.	Restrict multiplication operations: • So the product integer size is no larger than the target integer size. • To the recommended size.
Blocks multiply signals with nonzero bias.	Insert a Data Type Conversion block before and after the block containing the multiplication operation.
Product blocks are multiplying signals with mismatched fractional slopes.	Change the scaling of the output so that its fractional slope is the product of the input fractional slopes.
Product blocks are configured to do multiple division operations.	Multiply all the denominator terms together, and then do a single division using cascading Product blocks.
Product blocks are configured to do many multiplication or division operations.	Split the operations across several blocks, with each block performing one multiplication or one division operation.
Product blocks are configured with a divide operation for the first input and a multiply operation for the second input.	Reverse the inputs so the multiply operation occurs first and the division operation occurs second.
An input with an invariant constant value is used as the denominator in an online division operation.	Change the operation to multiplication, and replace the invariant input by its reciprocal.
A Sum block has a different input and output data type range.	Insert a Data Type Conversion block before and after the Sum block.

Conditions	Recommended Action
A Sum block has an input with a fractional slope that does not equal the fractional slope of the output.	Change the scaling of the input or output.
The net sum of the Sum block input biases does not equal the bias of the output.	Change the bias of the output, making the net bias adjustment zero.
The inputs of the Relational Operator block have different data types.	• Change the data type and scaling of the invariant input to match other inputs. • Insert Data Type Conversion blocks before the Relational Operator block to convert both inputs to a common data type.
The inputs of the Relational Operator block have different fractional slopes.	Change the scaling of either input.
The input and output of the MinMax block have different data types.	Change the data type of the input or output.
The input of the MinMax block has a different fractional slope than the output.	Change the scaling of the input or the output.
The initial condition of the Discrete-Time Integrator block is used to initialize both the state and the output.	Set the **Function Block Parameters** > **Use initial condition as initial and reset value for** parameter to `State only (most efficient)`.

Limitations

This task requires Real-Time Workshop and Simulink® Fixed Point™ licenses.

See Also

- Lookup Table block

- Remove code that protects against division arithmetic exceptions

Function Reference

Global Changes (p. 10-1)	Make global changes throughout system or subsystem
Tools (p. 10-1)	Get information about simulation or value

Global Changes

`autofixexp`	Automatically change scaling of fixed-point data types
`fixpt_instrument_purge`	Remove corrupt fixed-point instrumentation from model

Tools

`showfixptsimerrors`	Overflows from last simulation
`showfixptsimranges`	Logged maximum and minimum values from last fixed-point simulation

11

Functions — Alphabetical List

autofixexp

Purpose

Automatically change scaling of fixed-point data types

Syntax

```
autofixexp
```

Description

The `autofixexp` script automatically changes the scaling for model objects that specify fixed-point data types. However, if an object's **Lock output scaling against changes by the autoscaling tool** parameter is selected, the script refrains from scaling that object.

This script 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. Based on these values, the tool changes the scaling of fixed-point data types in a model so as to maximize precision and cover the range.

You can specify design minimum and maximum values for model objects using parameters typically titled **Output minimum** and **Output maximum**. See "Blocks That Allow Signal Range Specification" in *Using Simulink®* for a list of Simulink blocks that permit you to specify these values. In the autoscaling procedure that the `autofixexp` script executes, design minimum and maximum values take precedence over the simulation range.

If you intend to scale fixed-point data types using simulation minimum and maximum values, the script yields meaningful results when exercising the full range of values over which your design is meant to run. Therefore, the simulation you run prior to using `autofixexp` must simulate your design over its full intended operating range. It is especially important that you use simulation inputs with appropriate speed and amplitude profiles for dynamic systems. The response of a linear dynamic system is frequency dependent. For example, a bandpass filter will show almost no response to very slow and very fast sinusoid inputs, whereas the signal of a sinusoid input with a frequency in the passband will be passed or even significantly amplified. The response of nonlinear dynamic systems can have complicated dependence on both the signal speed and amplitude.

It is often good engineering practice to add a safety margin. The `autofixexp` script can set the binary points so as to cover an even larger

range, thereby reducing the chance of an overflow occurring. However, increased range results in reduced precision, so the safety margin that you choose must be limited. To specify a safety margin:

1. Open the Fixed-Point Tool (see "Opening the Fixed-Point Tool" on page 6-2).

2. In the **Dialog** pane, enter a value for the **Percent safety margin** parameter. The default value is 0.

   The **Percent safety margin** parameter allows you to specify a range that differs from that defined by the design or simulation range. For example, a value of 55 specifies that a range *at least* 55 percent larger is desired. A value of -15 specifies that a range *up to* 15 percent smaller is acceptable.

   You should be aware that the scaling is not exact for the binary-point-only case because the range is given (approximately) by a power of two. The lower limit is exact, but the upper limit is always one bit below a power of two. For example, if the maximum value is 5 and the minimum value is -0.5, then any **Percent safety margin** value from -20 to slightly under 60 would produce the same binary point because these limits are less than a factor of two from each other. The binary point selected will produce a range from -8 to +8 (minus a bit).

3. Click **Apply** to save your change.

Note If you already know the simulation range you need to cover, you can use an alternate autoscaling technique described in the `fixptbestprec` reference page.

See Also

`fxptdlg`, `showfixptsimranges`

fixpt_instrument_purge

Purpose

Remove corrupt fixed-point instrumentation from model

Syntax

```
fixpt_instrument_purge
fixpt_instrument_purge(modelName, interactive)
```

Description

The `fixpt_instrument_purge` script finds and removes fixed-point instrumentation from a model left by the Fixed-Point Tool and the fixed-point autoscaling script. The Fixed-Point Tool and the fixed-point autoscaling script each add callbacks to a model. For example, the Fixed-Point Tool appends commands to model-level callbacks. These callbacks make the Fixed-Point Tool respond to simulation events. Similarly, the autoscaling script adds instrumentation to some parameter values that gathers information required by the script.

Normally, these types of instrumentation are automatically removed from a model. The Fixed-Point Tool removes its instrumentation when the model is closed. The autoscaling script removes its instrumentation shortly after it is added. However, there are cases where abnormal termination of a model leaves fixed-point instrumentation behind. The purpose of `fixpt_instrument_purge` is to find and remove fixed-point instrumentation left over from abnormal termination.

`fixpt_instrument_purge(modelName, interactive)` removes instrumentation from model `modelName`. `interactive` is `true` by default, which prompts you to make each change. When `interactive` is set to `false`, all found instrumentation is automatically removed from the model.

See Also

`autofixexp`, `fxptdlg`

Purpose

Overflows from last simulation

Syntax

```
showfixptsimerrors
```

Description

The `showfixptsimerrors` script displays any overflows from the last fixed-point simulation. This information is also visible in the Fixed-Point Tool.

See Also

`fxptdlg`, `showfixptsimranges`

showfixptsimranges

Purpose

Logged maximum and minimum values from last fixed-point simulation

Syntax

```
showfixptsimranges
```

Description

The `showfixptsimranges` script displays the logged maximum and minimum values from the last fixed-point simulation.

The logged data is stored in the `FixPtSimRanges` cell array, which can be accessed by the `autofixexp` automatic scaling script.

See Also

`autofixexp`, `fxptdlg`, `showfixptsimerrors`

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 (p. A-3)	Lists the data types supported by the API and discusses the treatment of integers and data-type-overridden signals
Structure of the S-Function (p. A-6)	Displays the basic structure of an S-function that directly handles fixed-point data types
Storage Containers (p. A-8)	Discusses the containers used to hold signals in simulation and code generation
Data Type IDs (p. A-15)	Describes the creation, assignment, and usage of data type IDs, including how to get and set information about data types in an S-function

Overflow Handling and Rounding Methods (p. A-22)	Discusses the tokens you can use to define overflow handling and rounding methods in your fixed-point S-function, and describes the overflow logging structure
Creating MEX-Files (p. A-24)	Describes the extra steps that you need to take to create MEX-files for fixed-point S-functions
Fixed-Point S-Function Examples (p. A-26)	Walks through common tasks you might want to perform within your S-function
API Functions — Alphabetical List (p. A-35)	Contains reference pages for the API for user-written fixed-point S-functions in alphabetical order

Data Type Support

In this section...
“Supported Data Types” on page A-3
“The Treatment of Integers” on page A-4
“Data Type Override” on page A-4

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 Doubles`, such as
  - `flts16`
  - `flts16_En15`

- `fltu32_S3_B5`

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 `True singles` is treated as a Simulink built-in `single`.
- A signal that has been overridden with `True doubles` is treated as a Simulink built-in `double`.

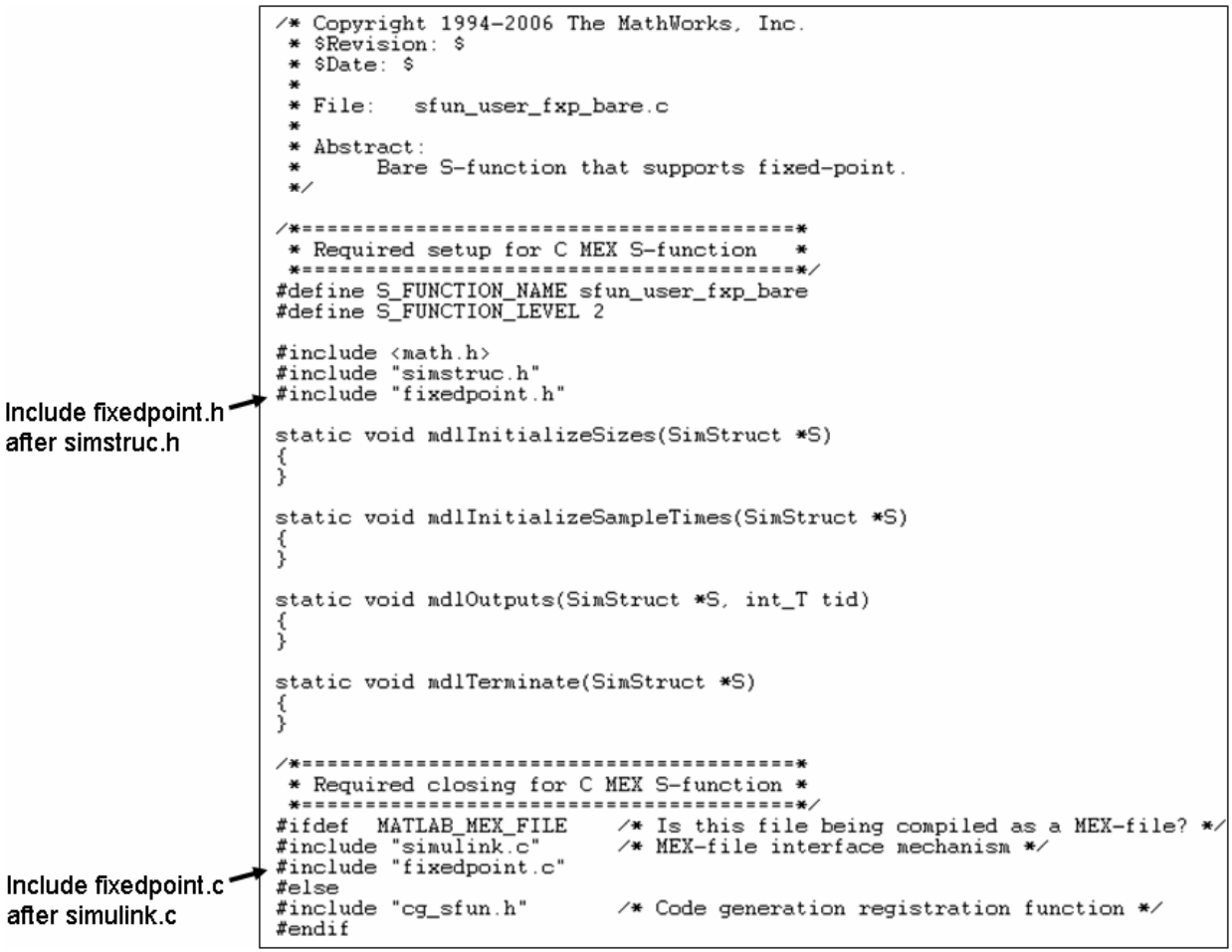
- A signal that has been overridden with `Scaled doubles` is treated as being of data type `ScaledDouble`.

`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.




```
/* 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 *Writing S-Functions* in the Simulink® documentation.

Storage Containers

In this section...

"Introduction" on page A-8

"Storage Containers in Simulation" on page A-8

"Storage Containers in Code Generation" on page A-12

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. Some signals are held in a structure called a "chunk array."

Chunk Arrays

Any signal with a word size greater than 32 bits is held in a multiword array called a "chunk array," which is comprised of an integer number of "chunks." For these signals, when the specified number of bits is less than the size of the chunk array, the signal bits are always stored in the least significant bits of the container. The unused bits of the chunk array are always cleared to zero.

To create a chunk array, declare a variable of data type `fxpChunkArray`.

You should use the tokens described in the following table to get information about chunk arrays in your S-function. Any time The MathWorks updates the API, a simple recompile will update the values of the tokens as necessary. Using the tokens will help you to maintain the readability and portability of your code.

Note In this appendix, "token" indicates enumerations or simple preprocessor macros that are set by The MathWorks for use with this API. You cannot change these tokens.

Tokens

Token	Description	Example of Use
FXP_ALL_ONES_CHUNK	Useful for creating bit masks	sfun_user_fxp_const.c Line 281
FXP_CHUNK_T	Data type of a chunk	sfun_user_fxp_const.c Line 281
FXP_BITS_PER_CHUNK	Number of bits in each chunk of a chunk array	sfun_user_fxp_const.c Line 224
FXP_INDEX_LEAST_SIGNIFICANT_ CHUNK	Position of the least significant chunk—at the right end or the left end of the chunk array	Not available
FXP_INDEX_MOST_SIGNIFICANT_ CHUNK	Position of the most significant chunk—at the right end or the left end of the chunk array	Not available
FXP_MAX_BITS	Maximum number of bits that a Simulink® supported data type can have	sfun_user_fxp_const.c Line 109
FXP_NUM_CHUNKS	Number of chunks in a chunk array	sfun_user_fxp_const.c Line 226

Storage Container Categories

During simulation, fixed-point signals are held in one of seven types of storage containers, as shown in the table below. In many cases, signals are represented in containers with more bits than their specified word length.

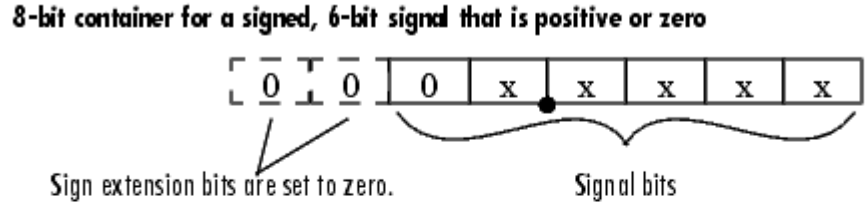
Fixed-Point Storage Containers

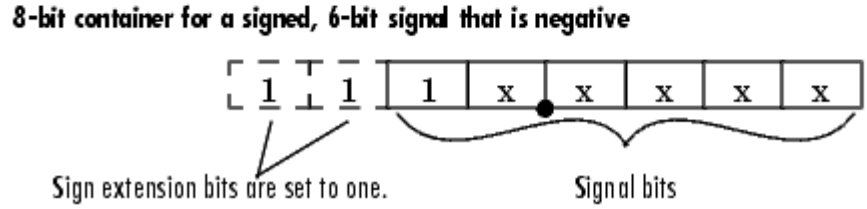
Container Category	Signal Word Length	Container Word Length	Container Size
`FXP_STORAGE_INT8` (signed) `FXP_STORAGE_UINT8` (unsigned)	1 to 8 bits	8 bits	1 byte
`FXP_STORAGE_INT16` (signed) `FXP_STORAGE_UINT16` (unsigned)	9 to 16 bits	16 bits	2 bytes
`FXP_STORAGE_INT32` (signed) `FXP_STORAGE_UINT32` (unsigned)	17 to 32 bits	32 bits	4 bytes
`FXP_STORAGE_ CHUNKARRAY`	33 to `FXP_MAX_BITS`	`FXP_MAX_BITS`	`FXP_NUM_CHUNKS*` `sizeof(FXP_CHUNK_T)`

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 set to specific values:

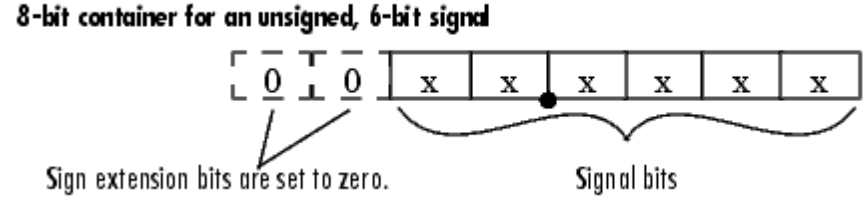
- If the signal is stored in a chunk array, the remaining bits must be cleared to zero.
- If the signal is not stored in a chunk array, the remaining bits must be sign extended:
  - If the data type is unsigned, the sign extension bits must be cleared to zero.
  - If the data type is signed, the sign 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_UNKOWN	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 doubles

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. If a data type used is too big to fit inside a long, code generation errors out. Multiword fixed-point signals are not supported in code generation.

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.

Chunk Arrays

In general, multiword fixed-point data types are not supported in code generation. However, fixed-point signals that use multiword chunk arrays in simulation are supported in code generation if they fit into a `long` for the target compiler.

For example, some compilers define a `long` to be 64 bits. For these compilers, a 33-to-64-bit signal that was held in a chunk array during simulation would be held in a `long` during code generation. Many other compilers, however, define a `long` to be 32 bits. Any 33-bit or greater signal held in a chunk array during simulation would cause code generation to error out for these targets.

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

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 Real-Time Workshop® 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.

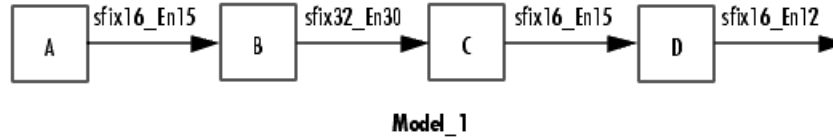
Data Type IDs

In this section...
"The Assignment of Data Type IDs" on page A-15
"Registering Data Types" on page A-16
"Setting and Getting Data Types" on page A-18
"Getting Information About Data Types" on page A-19
"Converting Data Types" on page A-21

The Assignment of Data Type IDs

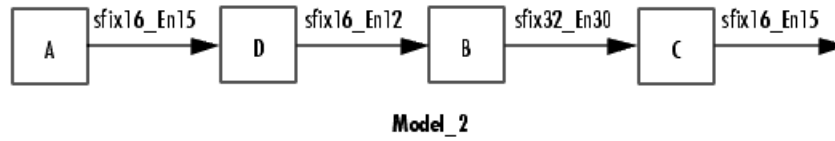
Each data type used in your S-function is assigned a data type ID. You should always use data type IDs to get and set information about data types in your S-function.

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-17. 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

Signal Data Type Information Functions (Continued)

Function	Description	Example of Use
`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
`ssGetDataTypeFracSlope`	Return the fractional slope of a registered data type	`sfun_user_fxp_dtprop.c` Line 234

Signal Scaling Information Functions (Continued)

Function	Description	Example of Use
`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" on page A-22

"Overflow Logging Structure" on page A-23

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_FLOOR`	Round to the closest representable number in the direction of negative infinity	Not Available
`FXP_ROUND_NEAR`	Round to the closest representable number, with the exact midpoint rounded in the direction of positive infinity	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" on page A-24

"MEX-Files on UNIX®" on page A-24

"MEX-Files on Windows®" on page A-24

Introduction

In addition to including `fixedpoint.c` and `fixedpoint.h` (see "Structure of the S-Function" on page A-6), 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 Real-Time Workshop® 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 the Simulink® software, are as follows:

- `extern\lib\win32\borland\libfixedpoint.lib`
- `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" on page A-26

"Getting the Input Port Data Type" on page A-27

"Setting the Output Port Data Type" on page A-29

"Interpreting an Input Value" on page A-30

"Writing an Output Value" on page A-32

"Using the Input Data Type to Determine the Output Data Type" on page A-34

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-17. 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 Functions — Alphabetical List

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_FLOOR`, `FXP_ROUND_NEAR`, 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-6.

Languages

C

TLC Functions

None

See Also

`ssFxpConvertFromRealWorldValue`, `ssFxpConvertToRealWorldValue`

ssFxpConvertFromRealWorldValue

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_FLOOR`, `FXP_ROUND_NEAR`, 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 `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-6.

Languages

C

TLC Functions

None

See Also

`ssFxpConvert`, `ssFxpConvertToRealWorldValue`

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-6.

Languages

C

TLC Functions

None

See Also

`ssFxpConvert`, `ssFxpConvertFromRealWorldValue`

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-6.

Languages C

See Also 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-6.

Languages

C

See Also

`ssFxpGetU32BitRegion`

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

$$real\text{-}world\ value = (slope \times integer) + bias$$

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-6.

Languages

C

TLC Functions

`FixPt_DataTypeBias`

See Also

`ssGetDataTypeFixedExponent`, `ssGetDataTypeFracSlope`, `ssGetDataTypeTotalSlope`

ssGetDataTypeFixedExponent

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-6.

Languages

C

TLC Functions

`FixPt_DataTypeFixedExponent`

See Also

`ssGetDataTypeBias`, `ssGetDataTypeFracSlope`, `ssGetDataTypeTotalSlope`

ssGetDataTypeFracSlope

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-6.

Languages

C

TLC Functions

FixPt_DataTypeFracSlope

See Also

ssGetDataTypeBias, ssGetDataTypeFixedExponent, ssGetDataTypeTotalSlope

ssGetDataTypeFractionLength

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-6.

Languages

C

TLC Functions

`FixPt_DataTypeFractionLength`

See Also

`ssGetDataTypeFxpWordLength`

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-6.

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.

ssGetDataTypeFxpContainWordLen

TLC Functions

`FixPt_DataTypeFxpContainWordLen`

See Also

`ssGetDataTypeFxpWordLength`, `ssGetDataTypeStorageContainCat`, `ssGetDataTypeStorageContainerSize`

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-6.

Languages

C

TLC Functions

`FixPt_DataTypeFxpIsSigned`

ssGetDataTypeFxpWordLength

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-6.

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`

ssGetDataTypeIsFixedPoint

Purpose

Determine whether registered data type is fixed-point data type

Syntax

```
extern int ssGetDataTypeIsFixedPoint(SimStruch *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-6.

Languages

C

TLC Functions

`FixPt_DataTypeIsFixedPoint`

See Also

`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-6.

Languages

C

TLC Functions

`FixPt_DataTypeIsFloatingPoint`

See Also

`ssGetDataTypeIsFixedPoint`

ssGetDataTypeIsFxpFltApiCompat

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-6.

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.

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

$$real\text{-}world\ value = (slope \times integer) + bias$$

where the slope can be expressed as

$$slope = fractional slope \times 2^{exponent}$$

When bias = 0 and fractional slope = 1, the only scaling factor that remains is a power of two:

$$real\text{-}world\ value = (2^{exponent} \times integer) = (2^{-fraction\ length} \times integer)$$

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-6.

Languages

C

TLC Functions

`FixPt_DataTypeIsScalingPow2`

See Also

`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

$$real\text{-}world\ value\ =\ (slope \times integer) + bias$$

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 simply equal to the stored integer value:

$$real\text{-}world\ value\ =\ integer$$

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`.

ssGetDataTypeIsScalingTrivial

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-6.

Languages C

TLC Functions `FixPt_DataTypeIsScalingTrivial`

See Also `ssGetDataTypeIsScalingPow2`

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-6.

Languages

C

See Also

`ssGetDataTypeStorageContainCat`

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 seven types of containers, as shown in the table below. 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 Size	Container `typedef`
`FXP_STORAGE_INT8` (signed) `FXP_STORAGE_UINT8` (unsigned)	1 to 8 bits	1 byte	`int8_T` or `uint8_T`
`FXP_STORAGE_INT16` (signed) `FXP_STORAGE_UINT16` (unsigned)	9 to 16 bits	2 bytes	`int16_T` or `uint16_T`
`FXP_STORAGE_INT32` (signed) `FXP_STORAGE_UINT32` (unsigned)	17 to 32 bits	4 bytes	`int32_T` or `uint32_T`
`FXP_STORAGE_CHUNKARRAY`	33 to `FXP_MAX_BITS`	`FXP_MAX_BITS`	`fxpChunkArray`

As shown by the last case in the table, any signal with a word size greater than 32 bits is held in a "chunk array," which is composed of an integer number of "chunks."

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 set to specific values:

- If the signal is stored in a chunk array, the remaining bits must be cleared to zero.
- If the signal is not stored in a chunk array, then the word length 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_UNKOWN`	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 doubles`

This function errors out when `ssGetDataTypeIsFxpFltApiCompat` returns `FALSE`.

ssGetDataTypeStorageContainCat

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-6.

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-13 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-6.

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

$$real\text{-}world\ value = (slope \times integer) + bias$$

where the slope can be expressed as

$$slope = fractional slope \times 2^{exponent}$$

This function returns the 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-6.

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

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 `True Doubles`, `True Singles`, `ScaledDouble`, 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-15.

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-6.

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 `True Doubles`, `True Singles`, `ScaledDouble`, 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-15.

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-6.

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 `True Doubles`, `True`

`Singles`, `ScaledDouble`, 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-15.

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-6.

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 `True Doubles`, `True Singles`, `ScaledDouble`, 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-15.

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-6.

ssRegisterDataTypeFxpSlopeBias

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`, `ssRegisterDataTypeFxpScaledDouble`

B

Selected Bibliography

[1] Burrus, C.S., J.H. McClellan, A.V. Oppenheim, T.W. Parks, R.W. Schafer, and H.W. Schuessler, *Computer-Based Exercises for Signal Processing Using MATLAB*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[2] Franklin, G.F., J.D. Powell, and M.L. Workman, *Digital Control of Dynamic Systems, Second Edition*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[3] *Handbook For Digital Signal Processing*, edited by S.K. Mitra and J.F. Kaiser, John Wiley & Sons, Inc., New York, 1993.

[4] Hanselmann, H., "Implementation of Digital Controllers — A Survey," *Automatica*, Vol. 23, No. 1, pp. 7-32, 1987.

[5] Jackson, L.B., *Digital Filters and Signal Processing, Second Edition*, Kluwer Academic Publishers, Seventh Printing, Norwell, Massachusetts, 1993.

[6] Middleton, R. and G. Goodwin, *Digital Control and Estimation — A Unified Approach*, Prentice Hall, Englewood Cliffs, New Jersey. 1990.

[7] Moler, C., "Floating points: IEEE Standard unifies arithmetic model," Cleve's Corner, The MathWorks, Inc., 1996. You can find this article at `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/index.html`.

[8] Ogata, K., *Discrete-Time Control Systems*, *Second Edition*, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[9] Roberts, R.A. and C.T. Mullis, *Digital Signal Processing*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

Glossary

This glossary defines terms related to fixed-point data types and numbers. These terms may appear in some or all of the documents that describe products from The MathWorks™ that have fixed-point support.

arithmetic shift

Shift of the bits of a binary word for which the sign bit is recycled for each bit shift to the right. A zero is incorporated into the least significant bit of the word for each bit shift to the left. In the absence of overflows, each arithmetic shift to the right is equivalent to a division by 2, and each arithmetic shift to the left is equivalent to a multiplication by 2.

See also binary point, binary word, bit, logical shift, most significant bit

bias

Part of the numerical representation used to interpret a fixed-point number. Along with the slope, the bias forms the scaling of the number. Fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times integer) + bias$$

where the slope can be expressed as

$$slope = fractional slope \times 2^{exponent}$$

See also fixed-point representation, fractional slope, integer, scaling, slope, [Slope Bias]

binary number

Value represented in a system of numbers that has two as its base and that uses 1's and 0's (bits) for its notation.

See also bit

binary point

Symbol in the shape of a period that separates the integer and fractional parts of a binary number. Bits to the left of the binary point are integer bits and/or sign bits, and bits to the right of the binary point are fractional bits.

See also binary number, bit, fraction, integer, radix point

binary-point-only scaling

Scaling of a binary number that results from shifting the binary point of the number right or left, and which therefore can only occur by powers of two.

See also binary number, binary point, scaling

binary word

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

See also bit, data type, word

bit

Smallest unit of information in computer software or hardware. A bit can have the value 0 or 1.

ceiling (round toward)

Rounding mode that rounds to the closest representable number in the direction of positive infinity. This is equivalent to the `ceil` mode in the Fixed-Point Toolbox™ software.

See also convergent rounding, floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

contiguous binary point

Binary point that occurs within the word length of a data type. For example, if a data type has four bits, its contiguous binary point must be understood to occur at one of the following five positions.

.0000
0.000
00.00
000.0
0000.

See also data type, noncontiguous binary point, word length

convergent rounding

Rounding mode that rounds to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to `0`.

See also ceiling (round toward), floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

data type

Set of characteristics that define a group of values. A fixed-point data type is defined by its word length, its fraction length, and whether it is signed or unsigned. A floating-point data type is defined by its word length and whether it is signed or unsigned.

See also fixed-point representation, floating-point representation, fraction length, word length

data type override

Parameter in the Fixed-Point Tool that allows you to set the output data type and scaling of fixed-point blocks on a system or subsystem level.

See also data type, scaling

exponent

Part of the numerical representation used to express a floating-point or fixed-point number.

1 Floating-point numbers are typically represented as

$$real\text{-}world\ value = mantissa \times 2^{exponent}$$

2 Fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times integer) + bias$$

where the slope can be expressed as

$$slope = fractional slope \times 2^{exponent}$$

The exponent of a fixed-point number is equal to the negative of the fraction length.

$$exponent = -1 \times fraction\ length$$

See also bias, fixed-point representation, floating-point representation, fraction length, fractional slope, integer, mantissa, slope

fixed-point representation

Method for representing numerical values and data types that have a set range and precision.

1 Fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times integer) + bias$$

where the slope can be expressed as

$$slope = fractional slope \times 2^{exponent}$$

The slope and the bias together represent the scaling of the fixed-point number.

2 Fixed-point data types can be defined by their word length, their fraction length, and whether they are signed or unsigned.

See also bias, data type, exponent, fraction length, fractional slope, integer, precision, range, scaling, slope, word length

floating-point representation

Method for representing numerical values and data types that can have changing range and precision.

1 Floating-point numbers can be represented as

$$real\text{-}world\ value = mantissa \times 2^{exponent}$$

2 Floating-point data types are defined by their word length.

See also data type, exponent, mantissa, precision, range, word length

floor (round toward)

Rounding mode that rounds to the closest representable number in the direction of negative infinity.

See also ceiling (round toward), convergent rounding, nearest (round toward), rounding, truncation, zero (round toward)

fraction

Part of a fixed-point number represented by the bits to the right of the binary point. The fraction represents numbers that are less than one.

See also binary point, bit, fixed-point representation

fraction length

Number of bits to the right of the binary point in a fixed-point representation of a number.

See also binary point, bit, fixed-point representation, fraction

fractional slope

Part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times integer) + bias$$

where the slope can be expressed as

$$slope = fractional slope \times 2^{exponent}$$

The term *slope adjustment* is sometimes used as a synonym for fractional slope.

See also bias, exponent, fixed-point representation, integer, slope

guard bits

Extra bits in either a hardware register or software simulation that are added to the high end of a binary word to ensure that no information is lost in case of overflow.

See also binary word, bit, overflow

integer

1. Part of a fixed-point number represented by the bits to the left of the binary point. The integer represents numbers that are greater than or equal to one.

2. Also called the "stored integer." The raw binary number, in which the binary point is assumed to be at the far right of the word. The integer is part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

   $$real\text{-}world\ value = 2^{-fraction\ length} \times integer$$

   or

   $$real\text{-}world\ value = (slope \times integer) + bias$$

   where the slope can be expressed as

$$slope = fractional slope \times 2^{exponent}$$

See also bias, fixed-point representation, fractional slope, integer, real-world value, slope

integer length

Number of bits to the left of the binary point in a fixed-point representation of a number.

See also binary point, bit, fixed-point representation, fraction length, integer

least significant bit (LSB)

Bit in a binary word that can represent the smallest value. The LSB is the rightmost bit in a big-endian-ordered binary word. The weight of the LSB is related to the fraction length according to

$$weight\ of\ LSB = 2^{-fraction\ length}$$

See also big-endian, binary word, bit, most significant bit

logical shift

Shift of the bits of a binary word, for which a zero is incorporated into the most significant bit for each bit shift to the right and into the least significant bit for each bit shift to the left.

See also arithmetic shift, binary point, binary word, bit, most significant bit

mantissa

Part of the numerical representation used to express a floating-point number. Floating-point numbers are typically represented as

$$real\text{-}world\ value = mantissa \times 2^{exponent}$$

See also exponent, floating-point representation

most significant bit (MSB)

Bit in a binary word that can represent the largest value. The MSB is the leftmost bit in a big-endian-ordered binary word.

See also binary word, bit, least significant bit

nearest (round toward)

Rounding mode that rounds to the closest representable number, with the exact midpoint rounded to the closest representable number in the direction of positive infinity. This is equivalent to the `nearest` mode in the Fixed-Point Toolbox™ software.

See also ceiling (round toward), convergent rounding, floor (round toward), rounding, truncation, zero (round toward)

noncontiguous binary point

Binary point that is understood to fall outside the word length of a data type. For example, the binary point for the following 4-bit word is understood to occur two bits to the right of the word length,

0000_ _.

thereby giving the bits of the word the following potential values.

$2^5 2^4 2^3 2^2 _ _$.

See also binary point, data type, word length

one's complement representation

Representation of signed fixed-point numbers. Negating a binary number in one's complement requires a bitwise complement. That is, all 0's are flipped to 1's and all 1's are flipped to 0's. In one's complement notation there are two ways to represent zero. A binary word of all 0's represents "positive" zero, while a binary word of all 1's represents "negative" zero.

See also binary number, binary word, sign/magnitude representation, signed fixed-point, two's complement representation

overflow

Situation that occurs when the magnitude of a calculation result is too large for the range of the data type being used. In many cases you can choose to either saturate or wrap overflows.

See also saturation, wrapping

padding

Extending the least significant bit of a binary word with one or more zeros.

See also least significant bit

precision

1. Measure of the smallest numerical interval that a fixed-point data type and scaling can represent, determined by the value of the number's least significant bit. The precision is given by the slope, or the number of fractional bits. The term *resolution* is sometimes used as a synonym for this definition.

2. Measure of the difference between a real-world numerical value and the value of its quantized representation. This is sometimes called quantization error or quantization noise.

See also data type, fraction, least significant bit, quantization, quantization error, range, slope

Q format

Representation used by Texas Instruments™ to encode signed two's complement fixed-point data types. This fixed-point notation takes the form

Qm.n

where

- *Q* indicates that the number is in Q format.
- *m* is the number of bits used to designate the two's complement integer part of the number.

- *n* is the number of bits used to designate the two's complement fractional part of the number, or the number of bits to the right of the binary point.

In Q format notation, the most significant bit is assumed to be the sign bit.

See also binary point, bit, data type, fixed-point representation, fraction, integer, two's complement

quantization

Representation of a value by a data type that has too few bits to represent it exactly.

See also bit, data type, quantization error

quantization error

Error introduced when a value is represented by a data type that has too few bits to represent it exactly, or when a value is converted from one data type to a shorter data type. Quantization error is also called quantization noise.

See also bit, data type, quantization

radix point

Symbol in the shape of a period that separates the integer and fractional parts of a number in any base system. Bits to the left of the radix point are integer and/or sign bits, and bits to the right of the radix point are fraction bits.

See also binary point, bit, fraction, integer, sign bit

range

Span of numbers that a certain data type can represent.

See also data type, precision

real-world value

Stored integer value with fixed-point scaling applied. Fixed-point numbers can be represented as

$$real\text{-}world\ value = 2^{-fraction\ length} \times integer$$

or

$$real\text{-}world\ value = (slope \times integer) + bias$$

where the slope can be expressed as

$$slope = fractional slope \times 2^{exponent}$$

See also integer

resolution

See **precision**

rounding

Limiting the number of bits required to express a number. One or more least significant bits are dropped, resulting in a loss of precision. Rounding is necessary when a value cannot be expressed exactly by the number of bits designated to represent it.

See also bit, ceiling (round toward), convergent rounding, floor (round toward), least significant bit, nearest (round toward), precision, truncation, zero (round toward)

saturation

Method of handling numeric overflow that represents positive overflows as the largest positive number in the range of the data type being used, and negative overflows as the largest negative number in the range.

See also overflow, wrapping

scaled double

A double data type that retains fixed-point scaling information. For example, in the Simulink® and Fixed-Point Toolbox™ software you can use data type override to convert your fixed-point data types to scaled doubles. You can then simulate to determine the ideal floating-point behavior of your system. After you gather that information you can turn data type override off to return to fixed-point data types, and your quantities still have their original scaling information because it was held in the scaled double data types.

scaling

1. Format used for a fixed-point number of a given word length and signedness. The slope and bias together form the scaling of a fixed-point number.

2. Changing the slope and/or bias of a fixed-point number without changing the stored integer.

See also bias, fixed-point representation, integer, slope

shift

Movement of the bits of a binary word either toward the most significant bit ("to the left") or toward the least significant bit ("to the right"). Shifts to the right can be either logical, where the spaces emptied at the front of the word with each shift are filled in with zeros, or arithmetic, where the word is sign extended as it is shifted to the right.

See also arithmetic shift, logical shift, sign extension

sign bit

Bit (or bits) in a signed binary number that indicates whether the number is positive or negative.

See also binary number, bit

sign extension

Addition of bits that have the value of the most significant bit to the high end of a two's complement number. Sign extension does not change the value of the binary number.

See also binary number, guard bits, most significant bit, two's complement representation, word

sign/magnitude representation

Representation of signed fixed-point or floating-point numbers. In sign/magnitude representation, one bit of a binary word is always the dedicated sign bit, while the remaining bits of the word encode the magnitude of the number. Negation using sign/magnitude representation consists of flipping the sign bit from 0 (positive) to 1 (negative), or from 1 to 0.

See also binary word, bit, fixed-point representation, floating-point representation, one's complement representation, sign bit, signed fixed-point, two's complement representation

signed fixed-point

Fixed-point number or data type that can represent both positive and negative numbers.

See also data type, fixed-point representation, unsigned fixed-point

slope

Part of the numerical representation used to express a fixed-point number. Along with the bias, the slope forms the scaling of a fixed-point number. Fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times integer) + bias$$

g

where the slope can be expressed as

$$slope = fractional slope \times 2^{exponent}$$

g

See also bias, fixed-point representation, fractional slope, integer, scaling, [Slope Bias]

slope adjustment

See **fractional slope**

[Slope Bias]
Representation used to define the scaling of a fixed-point number.

See also bias, scaling, slope

stored integer
See **integer**

trivial scaling
Scaling that results in the real-world value of a number being simply equal to its stored integer value.

In [Slope Bias] representation, fixed-point numbers can be represented as

$$real\text{-}world\ value = (slope \times integer) + bias$$

O

offline conversions
  addition and subtraction 3-41
  multiplication with zero bias and matching fractional slopes 3-49
  multiplication with zero bias and mismatched fractional slopes 3-48
  parameter conversions 3-36
  signals 3-37
online conversions
  addition and subtraction 3-41
  multiplication with zero bias and mismatched fractional slopes 3-49
  multiplication with zero biases and matching fractional slopes 3-50
  signals 3-37
optimization
  code 8-10
  using Model Advisor 8-15
overflows
  code generation 8-4
  definition 3-3
overrides with doubles
  global override 6-20

P

padding with trailing zeros
  definition 3-14
  feedback controller demo 6-16
parallel form realization 4-14
parameter conversions 3-35
  See also conversions 3-35
precision
  fixed-point numbers 2-10
  fixed-point parameters 3-36
  IEEE floating-point numbers 2-23

Q

quantization 3-3
  effects of fixed-point arithmetic 1-39
  feedback controller demo 6-20
  real-world value 2-7
  rounding 3-4

R

radix point 2-4
range
  fixed-point numbers 2-10
  IEEE floating-point numbers 2-22
rapid simulation (rsim) target 8-9
reading fixed-point data from workspace 1-32
Real-Time Workshop
  external mode 8-8
  rapid simulation (rsim) target 8-9
real-world values 2-6
realizations
  design constraints 4-7
  direct form 4-8
  parallel form 4-14
  series cascade form 4-11
registering fixed-point data types A-16
rounding modes 3-4
  code generation 8-4
  simplest 3-11
  toward ceiling 3-9
  toward floor 3-10
  toward nearest 3-7
  toward zero 3-5
rsim target 8-9
run-time API
  fixed-point data 1-35

S

S-functions

examples
fixed-point A-26
fixed-point A-1
fixed-point examples A-26
structure for fixed-point A-6
writing fixed-point A-1
saturation 3-18
scaling
accumulation 3-27
addition 3-24
binary-point-only 2-6
code generation 8-5
constant scaling for best precision 2-12
division 3-31
gain 3-29
multiplication 3-27
slope/bias 2-7
trivial A-4
scientific notation 2-17
series cascade form realizations 4-11
sharing fixed-point models 1-4
shifts 3-54
showfixptsimerrors function 11-5
showfixptsimranges function 11-6
sign
extension 3-21
sign bit for IEEE numbers 2-19
signal conversions 3-36
signal logging
fixed-point 1-35
simplest
rounding 3-11
Simulink acceleration modes 8-6
Simulink Fixed Point features 1-21
single-precision format 2-20
slope/bias scaling 2-7
ssFxpConvert A-36
ssFxpConvertFromRealWorldValue A-38
ssFxpConvertToRealWorldValue A-40
ssFxpGetU32BitRegion A-41
ssFxpSetU32BitRegion A-43
ssGetDataTypeBias A-45
ssGetDataTypeFixedExponent A-46
ssGetDataTypeFracSlope A-48
ssGetDataTypeFractionLength A-50
ssGetDataTypeFxpContainWordLen A-51
ssGetDataTypeFxpIsSigned A-53
ssGetDataTypeFxpWordLength A-54
ssGetDataTypeIsFixedPoint A-56
ssGetDataTypeIsFloatingPoint A-57
ssGetDataTypeIsFxpFltApiCompat A-58
ssGetDataTypeIsScalingPow2 A-59
ssGetDataTypeIsScalingTrivial A-61
ssGetDataTypeNumberOfChunks A-63
ssGetDataTypeStorageContainCat A-64
ssGetDataTypeStorageContainerSize A-67
ssGetDataTypeTotalSlope A-69
ssRegisterDataTypeFxpBinaryPoint A-71
ssRegisterDataTypeFxpFSlopeFix-ExpBias A-74
ssRegisterDataTypeFxpScaledDouble A-77
ssRegisterDataTypeFxpSlopeBias A-80
storage containers
fixed-point API A-8
stored integers 1-29
subtraction
See addition 3-26

T

targeting an embedded processor
design rules 4-5
operation assumptions 4-4
size assumptions 4-4
trivial scaling A-4
truncation 3-6
two's complement 2-4

U

underflow 2-22

W

wrapping 3-18
writing fixed-point data to workspace 1-32

Z

zero
    rounding 3-5