# Fixed-Point Blockset

**For Use with SIMULINK®**

**Modeling**

**Simulation**

**Implementation**

The MATH WORKS Inc.

User's Guide

*Version 2*

**How to Contact The MathWorks:**

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc.
24 Prime Park Way
Natick, MA 01760-1500 Mail

`http://www.mathworks.com` Web
`ftp.mathworks.com` Anonymous FTP server
`comp.soft-sys.matlab` Newsgroup

`support@mathworks.com` Technical support
`suggest@mathworks.com` Product enhancement suggestions
`bugs@mathworks.com` Bug reports
`doc@mathworks.com` Documentation error reports
`subscribe@mathworks.com` Subscribing user registration
`service@mathworks.com` Order status, license renewals, passcodes
`info@mathworks.com` Sales, pricing, and general information

*Fixed-Point Blockset User's Guide*



Printing History: March 1995 First printing
April 1997 Second printing (Revised for MATLAB 5)
January 1999 Third printing (Revised for MATLAB 5.3 (R11))

# Contents

## Before You Begin


**What Is the Fixed-Point Blockset?** . . . . . x
Intended Audience . . . . . x

**Related Products and Documentation** . . . . . xi
What Is MATLAB? . . . . . xi
What Is Simulink? . . . . . xii
What Is Real-Time Workshop? . . . . . xiii

**Chapter Quick Reference** . . . . . xiv

**Documentation Conventions** . . . . . xv

**Installation** . . . . . xvi


## 1 Introduction


**Why Use Fixed-Point Hardware?** . . . . . 1-2

**Why Use the Fixed-Point Blockset?** . . . . . 1-4

**The Development Cycle** . . . . . 1-5

**Fixed-Point Blocks** . . . . . 1-7

**Fixed-Point Blockset Features** . . . . . 1-9
Data Types . . . . . 1-9
Scaling . . . . . 1-10
Rounding . . . . . 1-11

Overflow Handling . . . . . 1-11
Locking the Output Scaling . . . . . 1-11
Overriding with Doubles . . . . . 1-11
Specialized Storage Capabilities . . . . . 1-12
Automatic Scaling Tool . . . . . 1-12
Standardization with Simulink . . . . . 1-12
Dynamic Block Configuration . . . . . 1-12
Updating Obsolete Fixed-Point Blocks . . . . . 1-13
Code Generation . . . . . 1-13

**How to Get Online Help** . . . . . 1-14

## 2 Getting Started

**Physical Quantities and Measurement Scales** . . . . . 2-2
Selecting a Measurement Scale . . . . . 2-2
Example: Selecting a Measurement Scale . . . . . 2-4

**Fixed-Point Blockset Library** . . . . . 2-9

**Block Configuration** . . . . . 2-10
Selecting a Data Type . . . . . 2-11
Selecting a Data Type Scaling . . . . . 2-13
Dynamic Parameter Dialog Box . . . . . 2-14

**Example: Converting from Doubles to Fixed-Point** . . . . . 2-15
Simulation Results . . . . . 2-16

**Additional Examples: Demos** . . . . . 2-20
Basic Demos . . . . . 2-20
Advanced Demos: Filters and Systems . . . . . 2-20

## 3 Data Types and Scaling


**Overview** . . . . . **3-2**

**Fixed-Point Numbers** . . . . . **3-3**
- Signed Fixed-Point Numbers . . . . . **3-3**
- Radix Point Interpretation . . . . . **3-4**
- Scaling . . . . . **3-4**
- Quantization . . . . . **3-6**
- Range and Precision . . . . . **3-8**
- Example: Fixed-Point Scaling . . . . . **3-9**
- Example: Constant Scaling for Best Precision . . . . . **3-11**

**Floating-Point Numbers** . . . . . **3-15**
- Scientific Notation . . . . . **3-15**
- The IEEE Format . . . . . **3-17**
- Range and Precision . . . . . **3-19**
- Exceptional Arithmetic . . . . . **3-21**


## 4 Arithmetic Operations


**Overview** . . . . . **4-2**

**Limitations on Precision** . . . . . **4-3**
- Rounding . . . . . **4-3**
- Padding with Trailing Zeros . . . . . **4-8**
- Example: Limitations on Precision and Errors . . . . . **4-9**
- Example: Maximizing Precision . . . . . **4-10**

**Limitations on Range** . . . . . **4-11**
- Saturation and Wrapping . . . . . **4-11**
- Guard Bits . . . . . **4-14**
- Example: Limitations on Range . . . . . **4-14**

**Recommendations for Arithmetic and Scaling** . . . . . **4-15**
Addition . . . . . **4-15**
Accumulation . . . . . **4-18**
Multiplication . . . . . **4-19**
Gain . . . . . **4-20**
Division . . . . . **4-22**
Summary . . . . . **4-24**

**Parameter and Signal Conversions** . . . . . **4-25**
Parameter Conversions . . . . . **4-26**
Signal Conversions . . . . . **4-26**

**Rules for Arithmetic Operations** . . . . . **4-29**
Computational Units . . . . . **4-29**
Addition and Subtraction . . . . . **4-29**
Multiplication . . . . . **4-34**
Division . . . . . **4-38**
Shifts . . . . . **4-40**

**Example: Conversions and Arithmetic Operations** . . . . . **4-45**

**5 Realization Structures**

**Overview** . . . . . **5-2**

**Realizations** . . . . . **5-4**
Direct Form II . . . . . **5-4**
Series Cascade Form . . . . . **5-7**
Parallel Form . . . . . **5-9**

**6 Tutorial: Feedback Controller Simulation**

**Overview** . . . . . **6-2**

**Simulink Model of a Feedback Design** . . . . . **6-3**

**Idealized Description of a Feedback Design** . . . . . **6-6**

**Digital Controller Realization** . . . . . **6-7**

**Simulation Results** . . . . . **6-9**
Simulation 1: Initial Guess at Scaling . . . . . **6-9**
Simulation 2: Global Override . . . . . **6-11**
Simulation 3: Automatic Scaling . . . . . **6-11**
Simulation 4: Individual Override . . . . . **6-16**

## 7 Building Systems and Filters

**Overview** . . . . . **7-2**
Realizations and Data Types . . . . . **7-3**
Realizations and Scaling . . . . . **7-3**

**Targeting an Embedded Processor** . . . . . **7-4**
Size Assumptions . . . . . **7-4**
Operation Assumptions . . . . . **7-4**
Design Rules . . . . . **7-5**

**Integrator Realizations** . . . . . **7-7**
Trapezoidal Integration . . . . . **7-7**
Backward Integration . . . . . **7-9**
Forward Integration . . . . . **7-10**

**Derivative Realizations** . . . . . **7-12**
Filtered Derivative . . . . . **7-12**
Unfiltered Derivative . . . . . **7-14**

**Lead Filter or Lag Filter Realization** . . . . . **7-18**

**State-Space Realization** . . . . . **7-21**

## 8 Command Reference

**Overview** . . . . . **8-2**

## 9 Block Reference

**The Block Reference Page** . . . . . **9-2**

**The Fixed-Point Blockset Library** . . . . . **9-10**

## A Code Generation

Languages . . . . . **A-2**
Storage Class of Variables . . . . . **A-2**
Storage Class of Parameters . . . . . **A-2**
Rounding Modes . . . . . **A-2**
Overflow Handling . . . . . **A-3**
Blocks . . . . . **A-3**
Scaling . . . . . **A-3**
Pure Integer Code . . . . . **A-3**

## B Bibliography

# Before You Begin

**What Is the Fixed-Point Blockset?** . . . . . x
Intended Audience . . . . . x

**Related Products and Documentation** . . . . . xi
What Is MATLAB? . . . . . xi
What Is Simulink? . . . . . xii
What Is Real-Time Workshop? . . . . . xiii

**Chapter Quick Reference** . . . . . xiv

**Documentation Conventions** . . . . . xv

**Installation** . . . . . xvi

# What Is the Fixed-Point Blockset?

The Fixed-Point Blockset includes a collection of block components that extend the standard Simulink® block library. You can use the fixed-point blocks the same way you use built-in Simulink blocks, combining them with blocks from other libraries to create sophisticated systems. The fixed-point blocks are grouped together as:

- **Arithmetic blocks**

  Multiply, divide, add, or subtract the inputs.
- **Conversion blocks**

  Convert from one data type to another. Using conversion blocks, you can create Simulink models that consist of both Simulink blocks and Fixed-Point Blockset blocks.
- **Look-up table blocks**

  Approximate a function using linear interpolation, linear extrapolation, or one of the input values.
- **Logical and comparison blocks**

  Connect the inputs with a Boolean or relational expression, bound the input range, or switch between multiple inputs.
- **Discrete-time blocks**

  Implement a discrete-time system.

With this set of blocks, you can create discrete-time dynamic systems that use fixed-point arithmetic. As a result, Simulink can simulate effects commonly encountered in fixed-point systems for applications such as control systems and time-domain filtering.

## Intended Audience

This book assumes you are familiar with both MATLAB® and Simulink. You should have a basic understanding of Boolean algebra and binary word representations.

# Related Products and Documentation

The Fixed-Point Blockset is a multiplatform product running on Microsoft Windows 95, Microsoft Windows NT, and UNIX systems.

The Fixed-Point Blockset requires:

- MATLAB 5.3 (R11)
- Simulink 3.0 (R11)

In addition, if you want to modify the fixed-point blocks, you need one of the C compilers supported by the `cmex` utility. If you want to generate code from your fixed-point models, you must have the Real-Time Workshop®. If you want to create an executable from the generated code, you must have the appropriate C compiler and linker.

## What Is MATLAB?

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include:

- Math and computation
- Algorithm development
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including graphical user interface building

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar noninteractive language such as C or Fortran.

The name MATLAB stands for *matrix laboratory*. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects, which together represent the state-of-the-art in software for matrix computation.

MATLAB has evolved over a period of years with input from many users. In university environments, it is the standard instructional tool for introductory and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the tool of choice for high-productivity research, development, and analysis.

MATLAB features a family of application-specific solutions called *toolboxes*. Very important to most users of MATLAB, toolboxes allow you to *learn* and *apply* specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.

See the MATLAB documentation set for more information.

## What Is Simulink?

Simulink is a software package for modeling, simulating, and analyzing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates.

For modeling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. With this interface, you can draw the models just as you would with pencil and paper (or as most textbooks depict them). This is a significant departure from previous simulation packages that require you to formulate differential equations and difference equations in a language or program. Simulink includes a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors. You can also customize and create your own blocks.

Models are hierarchical, so you can build models using both top-down and bottom-up approaches. You can view the system at a high-level, then double-click on blocks to go down through the levels to see increasing levels of model detail. This approach provides insight into how a model is organized and how its parts interact.

After you define a model, you can simulate it, using a choice of integration methods, either from the Simulink menus or by entering commands in

MATLAB's command window. The menus are particularly convenient for interactive work, while the command-line approach is very useful for running a batch of simulations (e.g., if you are doing Monte Carlo simulations or want to sweep a parameter across a range of values). Using scopes and other display blocks, you can see the simulation results while the simulation is running. In addition, you can change parameters and immediately see what happens, for "what if" exploration. The simulation results can be put in the MATLAB workspace for postprocessing and visualization.

Model analysis tools include linearization and trimming tools, which can be accessed from the MATLAB command line, plus the many tools in MATLAB and its application toolboxes. And because MATLAB and Simulink are integrated, you can simulate, analyze, and revise your models in either environment at any point.

The *Using Simulink* book describes how to work with Simulink. It explains how to manipulate Simulink blocks, access block parameters, and connect blocks to build models. It also provides reference descriptions of each block in the standard Simulink libraries.

## What Is Real-Time Workshop?

The Real-Time Workshop, for use with MATLAB and Simulink, produces code directly from Simulink models and automatically builds programs that can be run in a variety of environments, including real-time systems and stand-alone simulations. With the Real-Time Workshop, you can run your Simulink model in real-time on a remote processor. The Real-Time Workshop also enables you to run high-speed stand-alone simulations on your host machine or on an external computer. Features include support for multirate systems, as well as loop-rolling and S-function inlining, which allow you to optimize your code for size and efficiency.

The Real-Time Workshop enables you to use the Fixed-Point Blockset for rapid prototyping of real-time fixed-point systems, which can shorten development cycles and reduce costs. All of the blocks in the Fixed-Point Blockset are fully qualified for code generation with the Real-Time Workshop. See the *Real-Time Workshop User's Guide* for complete details on code generation.

# Chapter Quick Reference

Chapter 1, “Introduction,” provides an overview of the Fixed-Point Blockset features.

Chapter 2, “Getting Started,” shows you how to use many Fixed-Point Blockset features. After reading this chapter, you should be able to create simple fixed-point models.

Chapter 3, “Data Types and Scaling,” describes fixed-point data types, floating-point data types, and data type scaling. Examples illustrating how these features are incorporated into the Fixed-Point Blockset are provided.

Chapter 4, “Arithmetic Operations,” describes fixed-point arithmetic and its limitations. Examples illustrating how these features are incorporated into the Fixed-Point Blockset are provided.

Chapter 5, “Realization Structures,” describes how to create fixed-point realization structures using fixed-point blocks.

Chapter 6, “Tutorial: Feedback Controller Simulation,” describes how to simulate a fixed-point digital controller design.

Chapter 7, “Building Systems and Filters,” describes how to create and use fixed-point systems and filters.

Chapter 8, “Command Reference,” describes MATLAB M-file scripts and functions provided with the blockset.

Chapter 9, “Block Reference,” describes each fixed-point block in detail.

Appendix A, “Code Generation,” describes the simulation features that are available for code generation. Recommendations for producing efficient code are included as well.

Appendix B, “Bibliography,” provides an alphabetical list of references.

# Documentation Conventions

The documentation conventions used in this book are shown below.

| To Indicate | This Guide Uses | Example |
|---|---|---|
| Example code | `Monospace type` | To assign the value 5 to `A`, enter<br>`A = 5` |
| Function names/syntax | `Monospace type` | The `cos` function finds the cosine of each array element. |
| Keys | **Boldface** with an initial capital letter | Press the **Return** key. |
| Mathematical expressions | Variables in *italics*.<br>Functions, operators, and constants in standard type. | This vector represents the polynomial<br>$p = x^2 + 2x + 3$ |
| MATLAB output | `Monospace type` | MATLAB responds with<br>`A =`<br>`5` |
| Menu names, menu items, and controls | **Boldface** with an initial capital letter | Choose the **File** menu. |
| New terms | NCS *italics* | An *array* is an ordered collection of information. |

# Installation

The *MATLAB Installation Guide* provides essentially all of the information you need to install the Fixed-Point Blockset.

Prior to installation, you must obtain a License File or Personal License Password from The MathWorks. The License File or Personal License Password identifies the products you can install and use.

If you experience installation difficulties you can:

- Connect to The MathWorks home page (`http://www.mathworks.com`) and select the `Support & Services` link. Look for the license manager and installation information under the `Technical Notes` link under `Technical Support`.
- Send e-mail to Technical Support at `support@mathworks.com`.
- Call Technical Support at 508-647-7000 extension 4.

1

# Introduction


**Why Use Fixed-Point Hardware?** . . . . . 1-2

**Why Use the Fixed-Point Blockset?** . . . . . 1-4

**The Development Cycle** . . . . . 1-5

**Fixed-Point Blocks** . . . . . 1-7

**Fixed-Point Blockset Features** . . . . . 1-9
Data Types . . . . . 1-9
Scaling . . . . . 1-10
Rounding . . . . . 1-11
Overflow Handling . . . . . 1-11
Locking the Output Scaling . . . . . 1-11
Overriding with Doubles . . . . . 1-11
Specialized Storage Capabilities . . . . . 1-12
Automatic Scaling Tool . . . . . 1-12
Standardization with Simulink . . . . . 1-12
Dynamic Block Configuration . . . . . 1-12
Updating Obsolete Fixed-Point Blocks . . . . . 1-13
Code Generation . . . . . 1-13

**How to Get Online Help** . . . . . 1-14

# Why Use Fixed-Point Hardware?

Digital hardware is becoming the primary means in which control systems and signal processing filters are implemented. Digital hardware can be classified as either off the shelf hardware (e.g., microcontrollers, microprocessors, general purpose processors, 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 and/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? The answer to this question in many cases is cost and size:

- **Cost**

  Fixed-point hardware is more cost effective where price/cost is an important consideration. When using digital hardware in a product, especially mass-produced products, fixed-point hardware, costing much less than floating-point hardware, can result in significant savings.

- **Size**

  The logic circuits of fixed-point hardware are much less complicated than those of floating-point hardware. This means 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 was used, a large heat sink and battery would also be needed resulting in a costly, large, and heavy portable phone.

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

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

# Why Use the Fixed-Point Blockset?

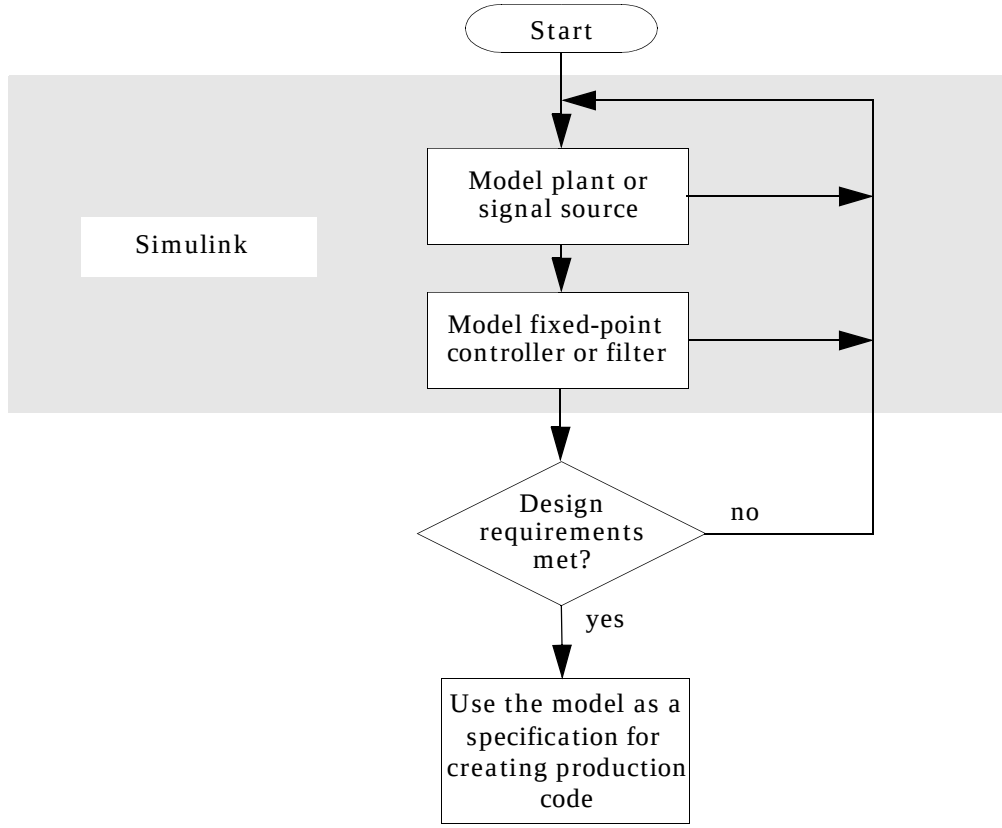
The Fixed-Point Blockset bridges the gap between designing a dynamic system (e.g., a control system or digital filter) and implementing it on fixed-point digital hardware. To do this, the blockset provides basic fixed-point Simulink building blocks that are used to design and simulate dynamic systems using fixed-point arithmetic. With the Fixed-Point Blockset, you can:

- Use fixed-point arithmetic to develop and simulate fixed-point Simulink models.
- Change the fixed-point data type, scaling, rounding mode, or overflow handling mode while the model is simulating. This allows you to explore issues related to numerical overflow, quantization errors, and computational noise.
- Generate fixed-point model code ready for execution on a floating-point processor. This allows you to emulate the effects of fixed-point arithmetic in a floating-point rapid prototyping system.
- Generate fixed-point model code ready for execution on a fixed-point processor.
- Modify and/or add new fixed-point blocks. Source code is provided for all fixed-point blocks.

The Fixed-Point Blockset addresses the issues of fixed-point single instruction, single data stream processors. Extensions to multiple instruction, multiple data stream processing units can be made. However, hardware consisting of multiple instruction and/or multiple data streams generally also has floating-point support.

# The Development Cycle

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



Using MATLAB, Simulink, and the Fixed-Point Blockset, the development cycle follows these steps:

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

2. Design and simulate a fixed-point dynamic system (e.g., a control system or digital filter) with the Fixed-Point Blockset that meets the design, performance, and other constraints.

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

When the design requirements have been met, you can use the model as a specification for creating production code using the Real-Time Workshop.

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

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

# Fixed-Point Blocks

The Fixed-Point Blockset includes a number of building blocks to assist you in designing and simulating dynamic systems using fixed-point arithmetic. The fixed-point blocks are grouped together as shown below. The fixed-point blocks are discussed in detail in Chapter 9, “Block Reference.”

## Arithmetic Blocks

| Block Name | Description |
|---|---|
| FixPt Constant | Generate a constant value |
| FixPt Gain | Multiply the input by a constant |
| FixPt Matrix Gain | Multiply the input by a constant matrix |
| FixPt Product | Multiply or divide inputs |
| FixPt Sum | Add or subtract inputs |

## Conversion Blocks

| Block Name | Description |
|---|---|
| FixPt Conversion | Convert from one Fixed-Point Blockset data type to another |
| FixPt Conversion Inherited | Convert input two to the data type of input one |
| FixPt Gateway In | Convert a Simulink data type to a Fixed-Point Blockset data type |
| FixPt Gateway Out | Convert a Fixed-Point Blockset data type to a Simulink data type |

## Look-Up Table Blocks

| Block Name | Description |
|---|---|
| FixPt Look-Up Table | Approximate a one-dimensional function using a selected look-up method. |
| FixPt Look-Up Table (2D) | Approximate a two-dimensional function using a selected look-up method |

## Logical and Comparison Blocks

| Block Name | Description |
|---|---|
| FixPt Logical Operator | Perform the specified logical operation on the inputs |
| FixPt Relational Operator | Perform the specified relational operation on the inputs |
| FixPt Relay | Switch output between two constants |
| FixPt Saturation | Bound the range of the input |
| FixPt Switch | Switch output between input one or input three based on the value of input two |

## Discrete-Time Blocks

| Block Name | Description |
|---|---|
| FixPt FIR | Implement a fixed-point finite impulse response (FIR) filter |
| FixPt Unit Delay | Delay a signal one sample period |
| FixPt Zero-Order Hold | Implement a zero-order hold of one sample period |

# Fixed-Point Blockset Features

A synopsis of the Fixed-Point Blockset features is given below. This feature "snapshot" is provided to give you a comprehensive summation of the blockset capabilities. Each feature is described in detail elsewhere in this book.

## Data Types

The Fixed-Point Blockset supports several fixed-point and floating-point data types, which are collectively referred to as the "Fixed-Point Blockset data types." The supported data types and related features are described below.

### Fixed-Point Data Types

- Integer, fractional, and generalized fixed-point data types are supported.
- Unsigned and two's complement formats are supported.
- The fixed-point word size can range from 1 to 128 bits.
- The radix (binary) point is not required to be contiguous with the fixed-point word.

### Floating-Point Data Types

- IEEE-style singles and doubles are supported.
- A nonstandard IEEE-style data type is supported. For this data type, the fraction (mantissa) can range from 1 to 52 bits and the exponent can range from 1 to 11 bits.

The label "Fixed-Point Blockset data types" indicates that data types supported by this blockset are unique to it, and not directly compatible with Simulink. This means that a double generated by Simulink cannot be passed directly into a Fixed-Point Blockset block, and a double generated by the Fixed-Point Blockset cannot be passed directly into a Simulink block. Instead, the FixPt Gateway In and FixPt Gateway Out blocks must be used as interfaces between the Fixed-Point Blockset and Simulink.

---

**Note:** Although doubles generated by Simulink and the Fixed-Point Blockset are not directly compatible, they have exactly the same range and precision.

---

## Scaling

The Fixed-Point Blockset supports two general scaling modes: radix point-only scaling and slope/bias scaling. Additionally, some blocks support scaling modes that maximize the precision for constant vectors or matrices. These scaling modes are described below.

### General Scaling Methods

Fixed-point numbers can be scaled in these ways:

- **Radix Point-Only**

  This is “powers of two” scaling since it only involves moving the radix point. Radix point-only scaling does not require the radix point to be contiguous with the data word. The advantage of this scaling mode is the number of processor arithmetic operations are minimized.

- **Slope/Bias**

  With this scaling mode, you can provide a slope and a bias. The advantage of slope/bias scaling is that it typically provides more efficient use of a finite number of bits.

### Constant Scaling for Best Precision

In addition to the general scaling modes described above, the Fixed-Point Blockset provides you with block-specific scaling modes for constant vectors and constant matrices. These scaling modes are based on radix point-only scaling and are designed to maximize precision.

- **Constant Vector Scaling**

  With this mode, you have the option of scaling a constant vector such that the precision is maximized for each element, or a common radix point can be found based on the maximum precision for the largest value of the vector.

- **Constant Matrix Scaling**

  With this mode, you have the option of scaling a constant matrix such that the precision is maximized for each element, or a common radix point can be found based on the maximum precision for the largest value of each row, each column, or the whole matrix.

The advantage of finding a common radix point is increased simulation speed, while the disadvantage is reduced precision.

## Rounding

Fixed-point numbers can be rounded in these ways:

- **Toward Zero**

  This mode rounds toward zero and is equivalent to MATLAB's `fix` command.
- **Toward Nearest**

  This mode rounds toward the nearest representable number, with the exact midpoint rounded toward positive infinity. Rounding toward nearest is equivalent to the MATLAB `round` command.
- **Toward Ceiling**

  This mode rounds toward positive infinity and is equivalent to MATLAB's `ceil` command.
- **Toward Floor**

  This mode rounds toward negative infinity and is equivalent to MATLAB's `floor` command.

## Overflow Handling

Operations on fixed-point numbers that produce an overflow condition can be dealt with in these ways:

- **Saturate**

  Overflows are set to either the maximum or minimum value represented by the word.
- **Wrap**

  Overflows can be set to any value represented by the word.

## Locking the Output Scaling

If the output data type is a generalized fixed-point number, then you have the option of locking its scaling. When locked, the automatic scaling tool will not change the output scaling. Otherwise, the automatic scaling tool is free to adjust the scaling

## Overriding with Doubles

The fixed-point data type can be overridden with doubles either globally or for individual blocks. This feature is useful when debugging a simulation.

## Specialized Storage Capabilities

The maximum and minimum values encountered during a simulation can be logged to the MATLAB workspace. These values can then be accessed by the automatic scaling tool, `autofixexp`.

## Automatic Scaling Tool

A script is provided that automatically changes the scaling for each 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, an automatic scaling GUI is provided. This interface allows you to easily control the parameters associated with automatic scaling and display the simulation results for a given model. With the automatic scaling GUI, you can:

- Turn on or turn off logging for all blocks.
- Override the output data type with doubles for all blocks.
- Invoke the automatic scaling script.
- Run the simulation.
- Display the scaling results for each block that had its scaling changed.

## Standardization with Simulink

The fixed-point blocks are feature-compatible with standard Simulink blocks. Standardization with Simulink provides these expanded capabilities:

- Vectorization of inputs and outputs.
- Variable number of input ports on appropriate blocks such as FixPt Sum.
- More powerful blocks that combine and expand the features of the basic blocks. For example, scalar addition and subtraction are combined in the vectorized FixPt Sum block.

## Dynamic Block Configuration

Generally, there are a large number of parameters to set for a typical Fixed-Point Blockset model. Therefore, a dynamic parameter management

system has been implemented for the parameter dialog box. The system works by turning off parameter choices that are not needed based on the output data type. If the data type parameter can't be evaluated, then the default is to leave all dialogs visible.

Dynamic block configuration is available for Simulink and all Simulink-based products.

## Updating Obsolete Fixed-Point Blocks

Obsolete fixed-point blocks from previous Fixed-Point Blockset releases can be updated to current fixed-point blocks.

## Code Generation

With the Real-Time Workshop, you can generate C code for execution on a fixed-point embedded processor. 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 code is structured so that key operations can be readily replaced by optimized target-specific libraries that you supply. You can also use the Target Language Compiler™ to customize the generated code.

All fixed-point blocks support code generation, but not every simulation feature is supported. For example, 13-bit numbers can be used for simulation but not for code generation.

# How to Get Online Help

The Fixed-Point Blockset provides several ways to get online help:

- **Block, System, and Filter Help**

  Press the **Help** button in any block, system, or filter dialog box to view the reference documentation for that block.

- **Help Desk**

  Type `helpdesk` or `doc` at the MATLAB command line to load the main MATLAB Help Desk page into your Web browser.

- **Release Information**

  Type `info fixpoint` at the MATLAB command line to view information related to the version of the Fixed-Point Blockset that you're using.

2

# Getting Started

**Physical Quantities and Measurement Scales** . . . . . 2-2
Selecting a Measurement Scale . . . . . 2-2
Example: Selecting a Measurement Scale . . . . . 2-4

**Fixed-Point Blockset Library** . . . . . 2-9

**Block Configuration** . . . . . 2-10
Selecting a Data Type . . . . . 2-11
Selecting a Data Type Scaling . . . . . 2-13
Dynamic Parameter Dialog Box . . . . . 2-14

**Example: Converting from Doubles to Fixed-Point** . . 2-15
Simulation Results . . . . . 2-16

**Additional Examples: Demos** . . . . . 2-20
Basic Demos . . . . . 2-20
Advanced Demos: Filters and Systems . . . . . 2-20

# Physical Quantities and Measurement Scales

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 degrees 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 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.

## Selecting a Measurement Scale

Suppose that measurements of liquid water are to be made, and suppose that these measurements must be represented using 8-bit unsigned integers. Fortunately, the temperature range of liquid water is limited. No matter what scale is used, liquid water can only go from the freezing point to the boiling point. Therefore, this range of temperatures must be captured 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 down side, 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. The units are roughly 0.3921568 degrees 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 “semi-standard” 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 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*, greater precision can be obtained 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

$$-\frac{S}{2} \leq Error \leq \frac{S}{2}$$

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.

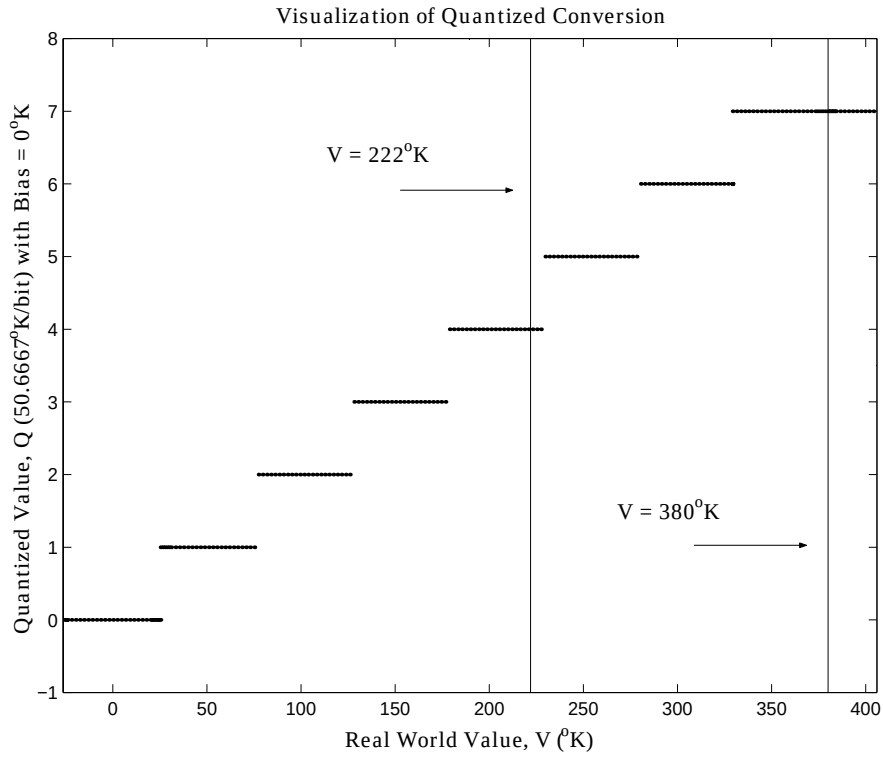
The temperature of the air flowing into the engine has a limited range. On a typical engine, the radiator is designed to keep the block below the boiling point of the cooling fluid. Let's assume a maximum of $225^{o}$ F ($380^{o}$ K). As the air flows through the intake manifold, it can be heated up 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

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

The quantized conversion and range of interest are illustrated below.

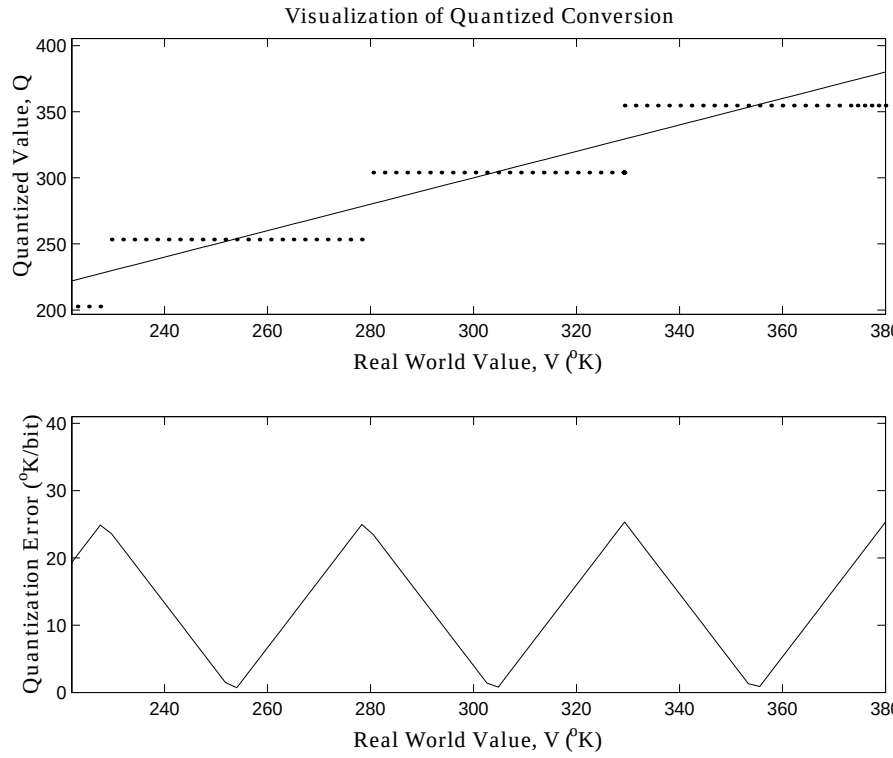



Notice that there are 7.5 possible quantization values. This is because only half of the first bit corresponds to temperatures (real-world values) greater than zero.

The quantization error is

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

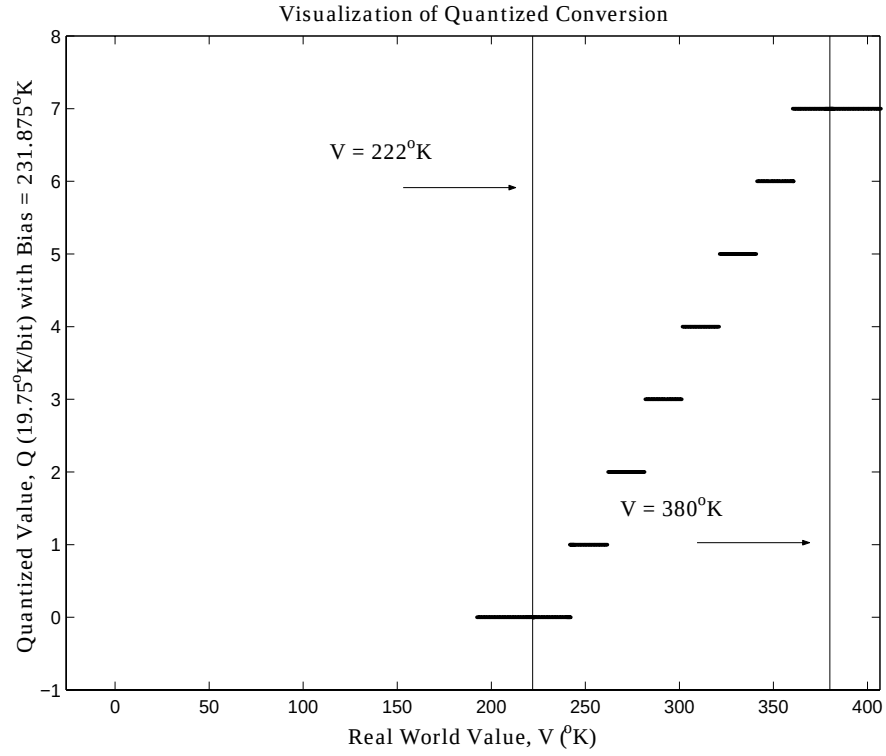
The range of interest of the quantized conversion and the absolute value of the quantized error are illustrated below.




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

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

The quantized conversion and range of interest are illustrated below.

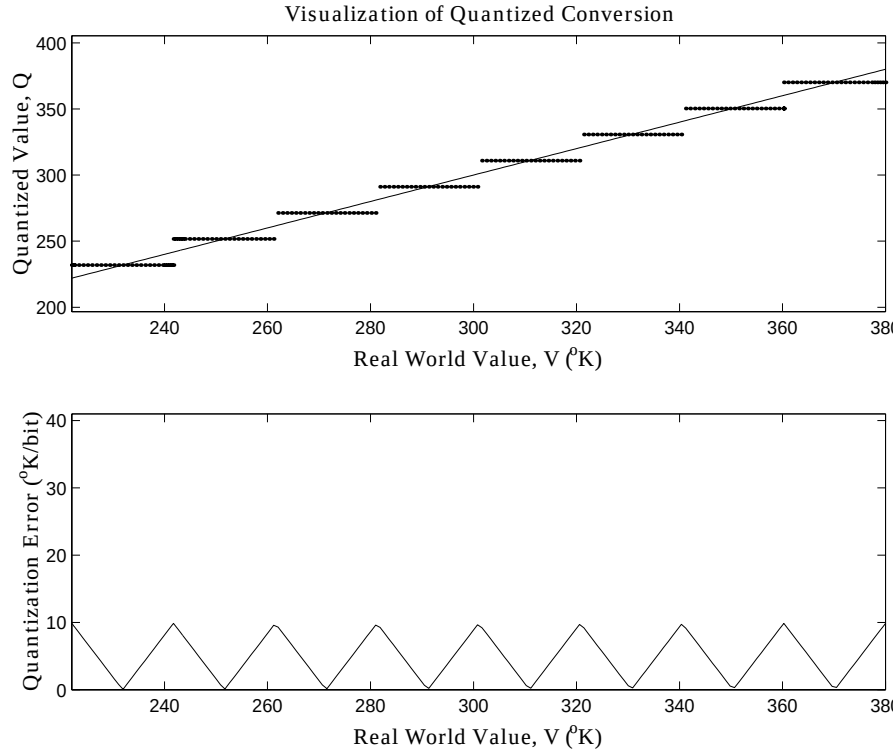



The quantization error is

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

which is approximately 2.5 times smaller than the error associated with the purely linear conversion.

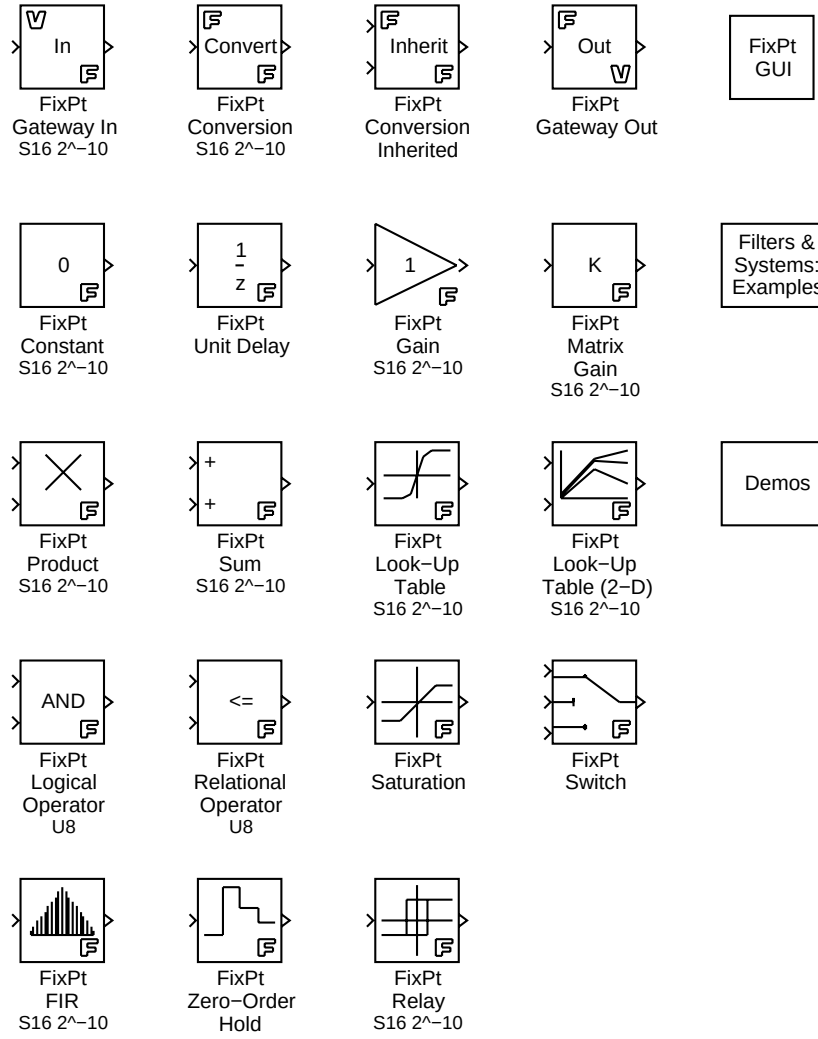
The range of interest of the quantized conversion and the absolute value of the quantized error are illustrated below.




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

# Fixed-Point Blockset Library

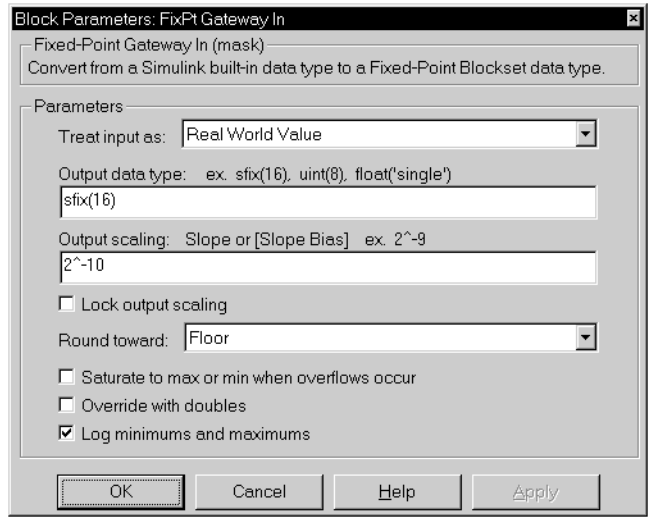
To display the Fixed-Point Blockset library, type `fixptlib` at the MATLAB prompt or double-click on the Toolboxes and Blocksets block under Simulink and select the Fixed-Point Blockset. The Fixed-Point Blockset library is shown below.



# Block Configuration

This section presents a brief overview of Fixed-Point Blockset block configuration. After reading this section and “Example: Converting from Doubles to Fixed-Point” on page 2-15, you should be able to build and configure simple fixed-point models.

Fixed-point blocks are configured with a parameter dialog box. Block configuration consists of supplying values for parameters via editable text fields, checkboxes, and pull-down menus as shown below with the **FixPt Gateway In** dialog box.




In what follows, block parameters are divided into two main categories: those related to selecting a data type, and those related to selecting a scaling. Understanding what these parameters mean and how they are configured will go a long way toward helping you understand the Fixed-Point Blockset features and capabilities.

These and other parameters are discussed in detail for each block in Chapter 9, “Block Reference.”

## Selecting a Data Type

For many fixed-point blocks, you must select a data type for the output and other parameters.

The Fixed-Point Blockset supports several fixed-point and floating-point data types. Fixed-point data types are characterized by their word size in bits and radix point. The radix point is the means by which fixed-point values are scaled. Floating-point data types are characterized by their sign bit, fraction (mantissa) field, and exponent field. The Fixed-Point Blockset adheres to the IEEE 754 standard for floating-point numbers.

The supported data types and MATLAB command syntax are described below.

---

**Note:** These commands are automatically called when the data type is specified in a block dialog box.

---

### Integers

- `uint(TotalBits)` returns a MATLAB structure that describes the data type of an unsigned integer with a size specified by `TotalBits`. The default radix point is assumed to lie to the right of all bits.
- `sint(TotalBits)` returns a MATLAB structure that describes the data type of a signed integer with a size specified by `TotalBits`. The default radix point is assumed to lie to the right of all bits.

### Fractional Numbers

- `ufrac(TotalBits)` returns a MATLAB structure that describes the data type of an unsigned fractional number with a size specified by `TotalBits`. The default radix point is assumed to lie to the left of all bits.

  `ufrac(TotalBits, GuardBits)` returns a MATLAB structure that describes the data type of an unsigned fractional number with a size specified by `TotalBits` and guard bits specified by `GuardBits`. The guard bits lie to the left of the default radix point.

- `sfrac(TotalBits)` returns a MATLAB structure that describes the data type of a signed fractional number with a size specified by `TotalBits`. The default radix point is assumed to lie to the right of the sign bit.

  `sfrac(TotalBits, GuardBits)` returns a MATLAB structure that describes the data type of a signed fractional number with a size specified by `TotalBits` and guard bits specified by `GuardBits`. The default radix point is assumed to be to the right of the sign bit, and the guard bits and sign bit lie to the left of the default radix point.

### Generalized Fixed-Point Numbers

- `ufix(TotalBits)` returns a MATLAB structure that describes the data type of an unsigned fixed-point number with a size specified by `TotalBits`. A default radix point is not included in the data type description. Instead, the scaling must be explicitly specified with the **Output scaling** parameter.
- `sfix(TotalBits)` returns a MATLAB structure that describes the data type of a signed fixed-point number with a size specified by `TotalBits`. A default radix point is not included in the data type description. Instead, the scaling must be explicitly specified with the **Output scaling** parameter.

The scaling for generalized fixed-point numbers is described in "Selecting a Data Type Scaling" on page 2-13.

### Standard IEEE-Style Numbers

- `float('single')` returns a MATLAB structure that describes the data type of an IEEE single (32 total bits and 8 exponent bits).
- `float('double')` returns a MATLAB structure that describes the data type of an IEEE double (64 total bits and 11 exponent bits).

### Nonstandard IEEE-Style Number

The command `float(TotalBits, ExpBits)` returns a MATLAB structure that describes a floating-point data type. The data type mimics the IEEE style. For example, 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 infinity and NaNs.

## Selecting a Data Type Scaling

Most data types supported by the Fixed-Point Blockset have a default scaling which cannot be changed. However, the scaling for a generalized fixed-point data type must be explicitly specified. The general encoding scheme and supported scaling modes are described below.

### General Slope/Bias Encoding Scheme

When representing an arbitrarily precise real-world value with a fixed-point number, it is often useful to define a general slope/bias encoding scheme

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

where:

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

The slope is partitioned into two components:

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

### Radix Point-Only Scaling

Radix point-only (or “powers of two”) scaling is specified 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 radix point is at a location 10 places to the left of the least significant bit.

### Slope/Bias Scaling

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

## Dynamic Parameter Dialog Box

Generally, there are a large number of parameters to set for a typical Fixed-Point Blockset model. Therefore, a dynamic parameter management system has been implemented for the parameter dialog box. The system works by turning off parameter choices that are not needed based on the parameter or output data type. If the data type parameter can't be evaluated then by default, all dialog parameters are visible. The data type-specific hidden parameters are given below.

**Table 2-1: Dynamic Parameters**

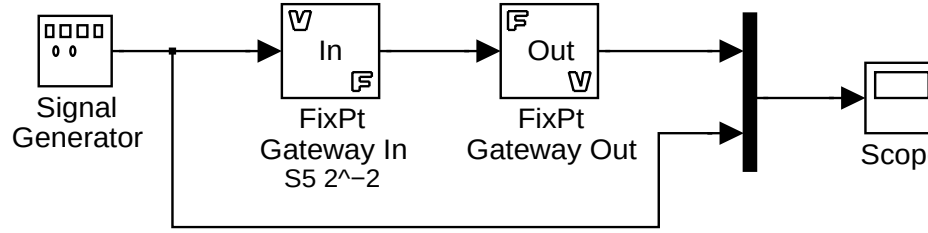
| Name | Data Type | Hidden Parameters | Explanation |
|---|---|---|---|
| Integers | `sint`, `uint` | Output scaling | When the data type is an integer, the default scaling is used. |
| | | Lock output scaling | Only generalized fixed-point data types can have their scaling locked. |
| Fractionals | `sfrac`, `ufrac` | Output scaling | When the data type is an integer, the default scaling is used. |
| | | Lock output scaling | Only generalized fixed-point data types can have their scaling locked. |
| Generalized Fixed-Point | `sfix`, `ufix` | None | |
| Floating-Point | `float('single')`<br>`float('double')`<br>`float(TotalBits, ExpBits)` | Output scaling | Applies only to fixed-point data types. |
| | | Lock output scaling | |
| | | Rounding | |
| | | Overflow handling | |

# Example: Converting from Doubles to Fixed-Point

The purpose of this example is to show you how to simulate a continuous real-world signal using a generalized fixed-point data type. The model used is the simplest possible model and employs only two Fixed-Point Blockset blocks. Although simple in design, the model gives you the opportunity to explore many of the important features of the Fixed-Point Blockset including:

- Data types
- Scaling
- Rounding
- Logging min/max simulation values to the workspace
- Overflow handling

The model used in this example is basically identical to `fxpdemo_dbl2fix.mdl`, which you can access via the fixed-point library's Demos block as Converting Doubles to Fixed-Point. Alternatively, you can access the model directly by typing its name at the command line. The model is shown below.




The FixPt Gateway In block must be used as the interface between Simulink and the Fixed-Point Blockset. Its function is to convert a Simulink double into one of the Fixed-Point Blockset data types. Conversely, the FixPt Gateway Out block must be used as the interface between the Fixed-Point Blockset and Simulink. Its function is to convert one of the Fixed-Point Blockset data types into a Simulink double.

When performing the simulation, you must consider these factors:

- **Data format**

  The data format you select depends on the characteristics of the input signal. For this example, the signal generator output is defined on the interval `[-5 5]`. The fixed-point output word is limited to 5 bits, and the data format must be signed.

- **Scaling and Range**

  When dealing with a limited number of bits, the trade-off between scaling and range can be very important. Overflow handling is an important consideration as well.

- **Rounding**

  Four rounding options are available. For most simulations, rounding to the nearest representable number is the most useful option.

Additionally, the FixPt Gateway In and FixPt Gateway Out blocks allow you to treat the input or output as real-world values or as stored integers. In terms of the general slope/bias encoding scheme described on page 2-13, the real-world value is given by *V* and the stored integer value is given by *Q*. For example, you might want to treat numbers as stored integer values if you are modeling hardware that produces integers as output.

## Simulation Results

The results of two simulation trials are given below. The first trial uses radix point-only scaling while the second trial uses slope/bias scaling.

### Trial 1: Radix Point-Only Scaling

When using radix point-only scaling, you must find the fixed power-of-two exponent, *E*, defined in "Selecting a Data Type Scaling" on page 2-13. For this scaling mode, the fractional slope *F* is set to 1 and no bias is required.

The FixPt Gateway In block is configured in this way:

- **Output data type**

  The output data type is given by `sfix(5)`. This creates a MATLAB structure that is a 5 bit, signed generalized fixed-point number.

- **Output scaling**

  The output scaling is given by `2^–2`, which puts the radix 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 $(1/2)^2 = 0.25$.

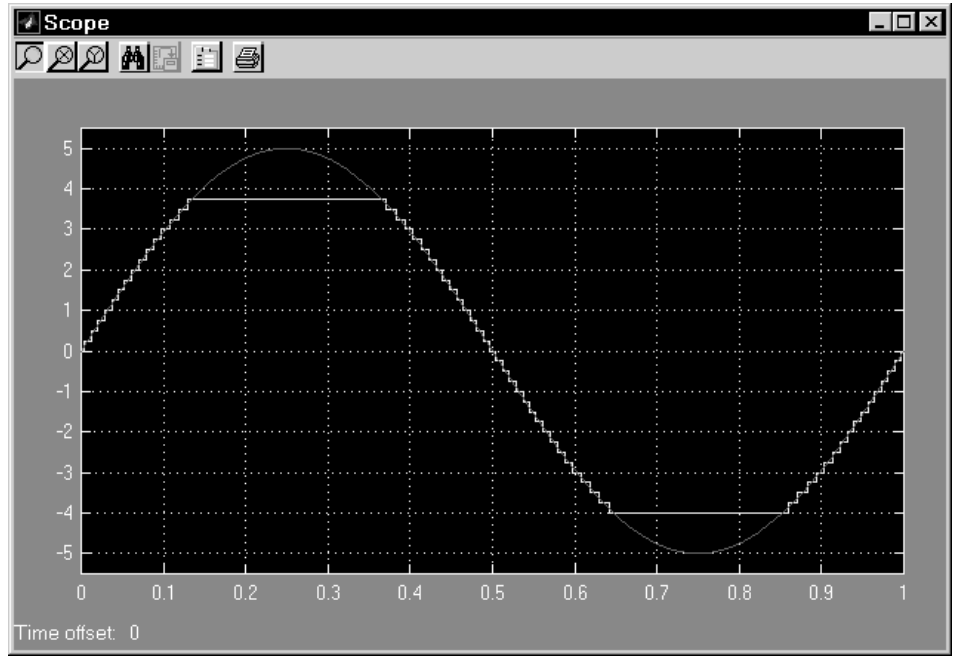
- **Rounding**

  The rounding mode is given by `Nearest`. This rounds the fixed-point result to the nearest representable number with the exact midpoint rounded towards positive infinity.

- **Overflows**

  Fixed-point values that overflow will saturate to the maximum or minimum value represented by the word.

The resulting real-world and fixed-point simulation results are shown below.




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 radix point-only scaling, then your only option is to sacrifice precision. Hence, the output scaling would be given by `2^-1`, which puts the radix 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, you must find the fractional slope, *F*, and the fixed power of two exponent, *E*, which are defined in "Selecting a Data Type Scaling" on page 2-13. No bias is required for this example since the sine wave is defined on the interval `[-5 5]`. The FixPt Gateway In 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 radix 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. This value is specified in the dialog box **Output scaling** parameter as `[0.3333]`.

If the global variable `FixDispPref` = 1, then the block icon displays *F* as 1.3332 and $2^E$ as $2^{-2}$. If `FixDispPref` = 2, then the block icon displays the slope as 0.3333.

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 due to the normalization condition of $1 \le F < 2$.

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.

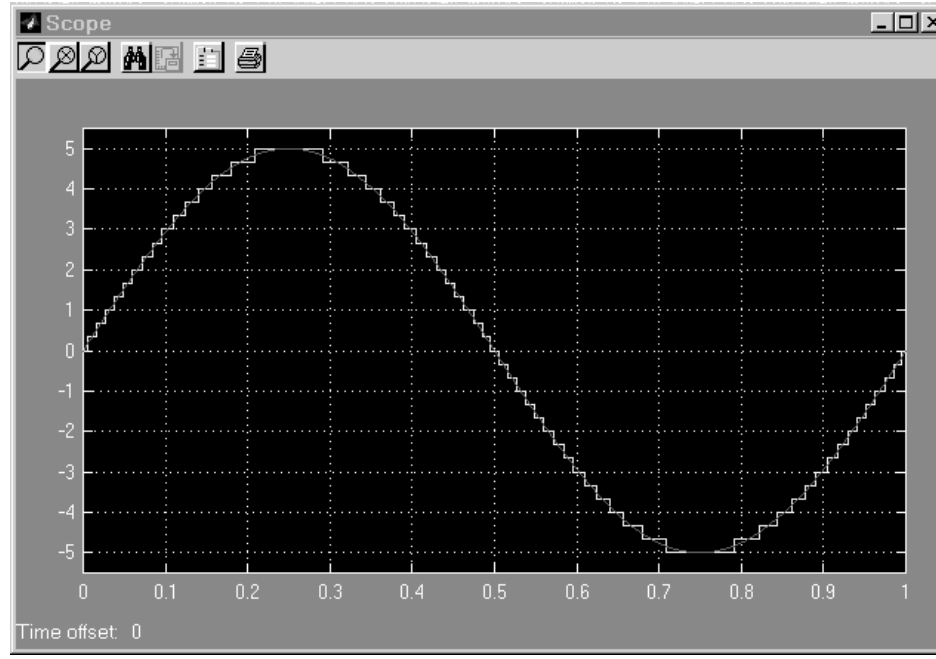
In general, you can achieve reasonable simulation results by selecting your scaling based on the formula

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

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 whose size is given by *ws*.

For this example, the formula produces a slope of 0.32258. The resulting real-world and fixed-point simulation results are shown below.



# Additional Examples: Demos

The Fixed-Point Blockset provides a collection of useful demos, which allow you to easily explore the Fixed-Point Blockset features by changing block parameters and observing the affects of those changes.

The demos are divided into two groups: basic demos that are designed to illustrate the basic functionality of the Fixed-Point Blockset, and advanced demos that illustrate the functionality of systems and filters built with fixed-point blocks. All demos are found in the `fxpdemos` directory.

## Basic Demos

The basic demos listed below can be accessed through the fixed-point library's Demos block.

| Demo Name | Description |
|---|---|
| Converting Doubles to Fixed-Point | Convert a double precision value to a fixed-point value |
| Converting Fixed-Point to Fixed-Point | Convert a fixed-point value to another fixed-point value |
| Inherited Fixed-Point to Fixed-Point Conversion | Convert a fixed-point value to an inherited fixed-point value |
| Fixed-Point Sine Wave Example | Add and multiply two fixed-point sine wave signals |
| Automatic Scaling in Feedback Control | Simulate a fixed-point feedback design |

Converting Doubles to Fixed-Point is discussed on page 2-15, while Automatic Scaling in Feedback Control is the subject of Chapter 6.

## Advanced Demos: Filters and Systems

The filter and system demos are intended to be used as a design aid so you can easily see how to build and test filters and systems suited to your particular needs. The output of these demos is compared to that of analogous built-in Simulink blocks with identical input.

The filter and system demos listed below can be accessed through the fixed-point library's Filters & Systems: Examples block, or you can type `fixptsys` at the command line.

| Demo Name | Description |
|---|---|
| Derivatives | Compare output from the FixPt Derivative and FixPt Derivative: Filtered blocks to that of analogous Simulink derivatives built using the Discrete Filter and Transfer Fcn blocks. |
| Integrators | Compare output from the FixPt Integrator: Trapezoidal, FixPt Integrator: Backward, and FixPt Integrator: Forward blocks to that of analogous Simulink integrators supported by the Discrete Integrator block. |
| Lead and Lag | Compare output from the FixPt Lead and Lag Filter block to that of analogous Simulink filters built using the Discrete Filter block. |
| State Space | Compare output from the FixPt State-Space Realization block to that of the analogous built-in Simulink block. |

You can invoke a filter or system demo by double-clicking the appropriate subsystem. For example, to invoke the derivative demos, double-click the Demo: Derivative subsystem. For more information about filters and systems refer to Chapter 7, “Building Systems and Filters.”

Additional fixed-point demos exist for the direct form II, series cascade form, and parallel form realizations. These demos and realizations are discussed in Chapter 5, “Realization Structures.”

3

# Data Types and Scaling

**Overview** . . . . . 3-2

**Fixed-Point Numbers** . . . . . 3-3
Signed Fixed-Point Numbers . . . . . 3-3
Radix Point Interpretation . . . . . 3-4
Scaling . . . . . 3-4
Quantization . . . . . 3-6
Range and Precision . . . . . 3-8
Example: Fixed-Point Scaling . . . . . 3-9
Example: Constant Scaling for Best Precision . . . . . 3-11

**Floating-Point Numbers** . . . . . 3-15
Scientific Notation . . . . . 3-15
The IEEE Format . . . . . 3-17
Range and Precision . . . . . 3-19
Exceptional Arithmetic . . . . . 3-21

# 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 its word size in bits and radix (binary) point. The radix point is the means by which binary words are scaled. Within this blockset, signed and unsigned fixed-point data types can be integers, fractionals, or generalized fixed-point numbers. The main difference between the fixed-point data types is the default radix point. Floating-point data types are characterized by a mantissa (or fraction) field, an exponent field, and a sign bit. This blockset adheres to the IEEE 754 Standard for floating-point numbers and supports IEEE singles, IEEE 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 Fixed-Point Blockset, 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, you can generate production code based on that model.

# Fixed-Point Numbers

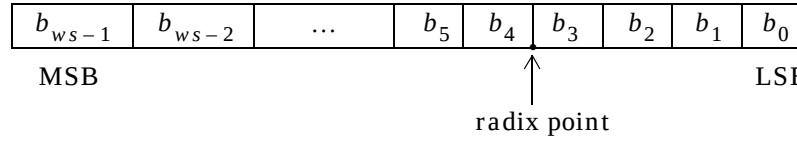
Fixed-point numbers have the radix point in a specific location of a computer word. The Fixed-Point Blockset supports signed and unsigned integers, fractionals, and generalized fixed-point numbers.

---

**Note:** This blockset supports fixed-point word sizes up to 128 bits.

---

A common representation of a binary fixed-point number (either signed or unsigned) is




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, represented by location $b_{ws-1}$.
- The least significant bit (LSB) is the rightmost bit, represented by location $b_0$.
- The radix (binary) point is shown four places to the left.

## Signed Fixed-Point Numbers

Computer hardware typically represent 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 this blockset.

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 (i.e., there is no sign bit). Instead, the sign information is implicitly defined within the computer architecture.

## Radix Point Interpretation

The radix point is the means by which fixed-point numbers are scaled. It is usually the software, however, that determines the radix point. The hardware, when performing basic math functions such as addition or subtraction, 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 radix point is to the right of $b_0$.

Within this blockset, the main differences between fixed-point data types is whether they are signed or unsigned, and the default radix point. Data types are specified with the **Parameter data type** and **Output data type** dialog box parameters The supported fixed-point data types are described below.

### Integers

The default radix point for signed and unsigned integer data types is assumed to be just to the right of the LSB. Unsigned integers are designated by `uint` and signed integers are designated by `sint`.

### Fractionals

The default radix point for unsigned fractional data types is just to the left of the MSB while for signed fractionals, the radix point is just to the right of the MSB. Unsigned fractionals are designated by `ufrac` and signed fractionals are designated by `sfrac`.

### Generalized Fixed-Point Numbers

For signed and unsigned generalized fixed-point numbers, there is no default radix point. Instead, the scaling must be explicitly given in the parameter dialog box. Unsigned generalized fixed-point numbers are designated by `ufix` and signed generalized fixed-point numbers are designated by `sfix`.

## Scaling

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

With the Fixed-Point Blockset, you can select a fixed-point data type whose scaling is defined by its default radix point, or you can select a generalized fixed-point data type and choose an arbitrary linear scaling that suits your needs. Scaling is specified with the **Parameter scaling** and **Output scaling** dialog box parameters. 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 $V$.
- $S = F \cdot 2^E$ is the slope.
- $B$ is the bias.

The slope is partitioned into two components:

- $2^E$ specifies the radix 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 choices available within this encoding scheme are described below.

## Radix Point-Only Scaling

As the name implies, radix point-only (or “powers of two”) scaling involves only moving the radix point within the generalized fixed-point word. With this scaling option, 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.

Radix point-only scaling is specified 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 radix point is at a location 10 places to the left of the least significant bit.

### Slope/Bias Scaling

When scaling by slope and bias, the slope $S$ and bias $B$ of the quantized real-world number can take on any value. Scaling by slope and bias is specified with the syntax `[slope bias]`, which creates a MATLAB structure with the given slope and bias. For example, `[5/9 10]` defines a scaling with a slope of 5/9 and a bias of 10. The slope must be a positive number.

The advantages and disadvantages of the radix point and slope/bias scaling modes are discussed in “Recommendations for Arithmetic and Scaling” on page 4-15.

## 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

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

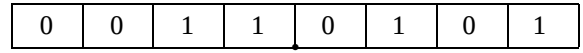
where:

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

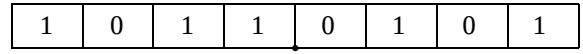
$b_i$ are called *bit multipliers* and $2^i$ are called the *weights*.

### Example: Fixed-Point Format

The formats for 8-bit signed and unsigned fixed-point values is given below.




Unsigned data type




Signed data type

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

The binary number 0011.0101 yields the same value for the unsigned and two's complement representation since 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

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

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

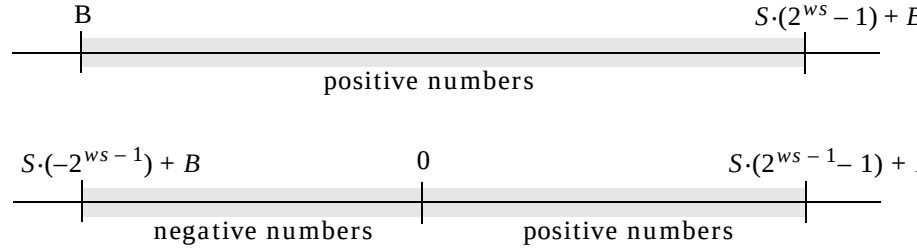
$$= -4.6875$$

## 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 depends on the length of the word and the scaling.

### Range

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



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 $2^{ws} - 1$ since 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 radix point. For generalized fixed-point data types, the scaling must be explicitly defined as either slope/bias or radix point-only. In either case, the precision is given by the slope.

### Data Type Parameters

The low limit, high limit, and default scaling (radix point-only) for the fixed-point data types supported by this blockset are given below.

**Table 3-1: 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 |

## Example: Fixed-Point Scaling

Table 3-2 gives the precision, range of signed values, and range of unsigned values for an 8-bit generalized fixed-point data type with radix point-only

scaling. Note that the first scaling value ($2^1$) represents a radix point that is not contiguous with the word.

**Table 3-2: Range of an 8-Bit Fixed-Point Data Type—Radix Point-Only Scaling**

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

Table 3-3 gives the precision, and range of signed and unsigned values for an 8-bit fixed-point data type with slope/bias scaling using a bias of 1.0 and starting with a slope of 1.25. Note that the slope is the same as the precision.

**Table 3-3: Range of an 8-Bit Fixed-Point Data Type—Slope/Bias Scaling**

| 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 |

**Table 3-3: Range of an 8-Bit Fixed-Point Data Type—Slope/Bias Scaling**

| Bias | Slope/Precision | Range of Signed Values (low, high) | Range of Unsigned Values (low, high) |
|---|---|---|---|
| 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 |

## Example: Constant Scaling for Best Precision

The Fixed-Point Blockset provides you with block-specific modes for scaling constant vectors and constant matrices. These scaling modes are based on radix point-only scaling and are described below:

- **Constant Vector Scaling**

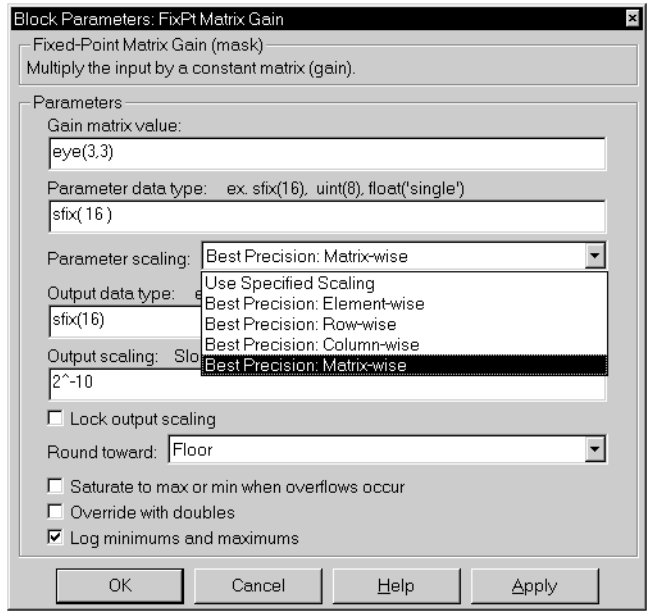
  Using this mode, you can scale a constant vector such that its precision is maximized element-by-element, or a common radix point is found based on the best precision for the largest value of the vector.

- **Constant Matrix Scaling**

  Using this mode, you can scale a constant matrix such that its precision is maximized element-by-element, or a common radix point is found based on the best precision for the largest value of each row, each column, or the whole matrix.

Constant matrix and constant vector scaling is available only for generalized fixed-point data types. All other fixed-point data types use their default scaling.

The available constant matrix scaling modes are shown below for the FixPt Matrix Gain block.




To understand how you might use these scaling modes, 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 `M` is input into the FixPt Matrix Gain block, and you want to scale it using one of the constant matrix scaling modes.

The results of using these modes are described below.

- `Use Specified Scaling`

  Suppose the matrix elements are converted to a signed, 10-bit generalized fixed-point data type with radix point-only scaling of $2^{-7}$ (that is, the radix 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.

- `Best Precision: Element-wise`

  If `M` is scaled such that the precision is maximized for each matrix element, you obtain

  ```
   3.3379e-005   3.3304e-006   3.3341e-007   3.3295e-008
   3.3379e-004   3.3379e-005   3.3304e-006   3.3341e-007
   3.3340e-003   3.3379e-004   3.3379e-005   3.3304e-006
   3.3325e-002   3.3340e-003   3.3379e-004   3.3379e-005
   3.3301e-001   3.3325e-002   3.3340e-003   3.3379e-004
  ```

- `Best Precision: Row-wise`

  If `M` is scaled based on the largest value for each row, you obtain

  ```
   3.3379e-005   3.3379e-006   3.5763e-007             0
   3.3379e-004   3.3379e-005   2.8610e-006             0
   3.3340e-003   3.3569e-004   3.0518e-005             0
   3.3325e-002   3.2959e-003   3.6621e-004             0
   3.3301e-001   3.3203e-002   2.9297e-003             0
  ```

- `Best Precision: Column-wise`

  If `M` is scaled based on the largest value for each column, you obtain

```
             0             0             0             0
             0             0             0             0
   2.9297e-003   3.6621e-004   3.0518e-005   2.8610e-006
   3.3203e-002   3.2959e-003   3.3569e-004   3.3379e-005
   3.3301e-001   3.3325e-002   3.3340e-003   3.3379e-004
```

- `Best Precision: Matrix-wise`

  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 scaling the matrix column-wise, row-wise, or matrix-wise is reduced precision resulting from the use of a common radix point. The advantage of using a common radix point is reduced code size and possibly increased processor speed.

# 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 is overcome by using scientific notation. With scientific notation, you can dynamically place the radix point at a convenient location and use powers of the radix to keep track of that location. Thus, a range of very large and very small numbers can be represented with only a few digits.

Any binary floating-point number can be represented 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 Fixed-Point Blockset supports single precision and double precision floating-point numbers as defined by the IEEE 754 Standard for binary floating-point data types. Additionally, a nonstandard IEEE-style number is supported.

To link the world of fixed-point numbers with the world of floating-point numbers, the concepts behind scientific notation are reviewed below.

## Scientific Notation

A direct analogy exists between scientific notation and radix point notation. For example, scientific notation using five decimal digits for the mantissa 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 5 bits for the mantissa 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 mantissa

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 the radix point must be contiguous with the mantissa. For example, a word consisting of three unsigned bits is usually represented in scientific notation in 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.

$$bbb00000. = bbb. \times 2^{5}$$

$$bbb00. = bbb. \times 2^{2}$$

$$.00bbb = bbb. \times 2^{-5}$$

$$.00000bbb = bbb. \times 2^{-8}$$

However, these extra zeros never change to ones 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:** The restriction of the radix point being contiguous with the mantissa is unnecessary, and this blockset allows the radix point to be extended to any arbitrary location.

---

## The IEEE Format

The IEEE 754 Standard for binary floating-point arithmetic has been widely adopted and is used on virtually all floating-point processors and arithmetic coprocessors.

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, an exponent field, and a fraction field. These components as well as the specific formats for singles and doubles are discussed below.

### 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 Exponent Field

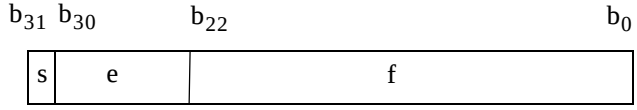
In the IEEE format, exponent representations are biased. This means a fixed value (the bias) is subtracted from the field to get the true exponent value. For example, if the exponent field is 8 bits, then the numbers 0 through 255 are represented. If there is a bias of 127, then the true exponent values range from -127 to 128.

### 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 radix point and decreasing or increasing the exponent of the radix by a corresponding amount. To simplify operations on these numbers, they are *normalized* in the IEEE format. A normalized binary number has a fraction with 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.

### Single Precision Format

The IEEE 754 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.



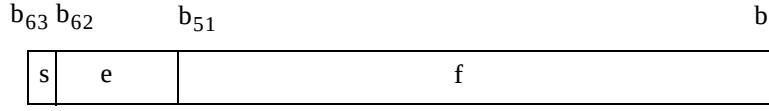

The relationship between this format and the representation of real numbers is given by

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

Denormalized values are discussed in “Exceptional Arithmetic” on page 3-21.

### Double Precision Format

The IEEE 754 double precision (64-bit) floating-point format consists of a 1-bit sign indicator *s*, an 11-bit biased exponent *e*, and a 52-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

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

Denormalized values are discussed later in this chapter.

### Nonstandard IEEE Format

This blockset 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.

Nonstandard floating-point numbers are designated by `float(TotalBits, ExpBits)` where `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 numbers. 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.

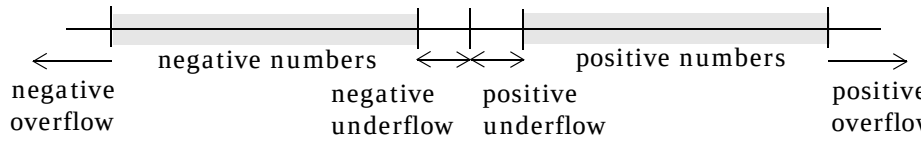
When specifying a nonstandard format, you should remember that the number of exponent bits largely determines the range of the result and the number of fraction bits largely determines the precision of the result.

## 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




where:

- Positive numbers are defined within the range $2^{1-bias}$ to $(1 - 2^{-f}) \cdot 2^{bias}$.
- Negative numbers are defined within the range $-2^{1-bias}$ to $-(1 - 2^{-f}) \cdot 2^{bias}$.
- Positive numbers greater than $(1 - 2^{-f}) \cdot 2^{bias}$ are called positive overflow.
- Positive numbers less than $2^{1-bias}$ are called positive underflow.
- Negative numbers greater than $-(1 - 2^{-f}) \cdot 2^{bias}$ are called negative overflow.
- Negative numbers less than $-2^{1-bias}$ are called negative underflow.
- Zero is given by a special bit pattern.

Overflows and underflows result from exceptional arithmetic conditions. Exceptional arithmetic is discussed later in this chapter.

---

**Note:** The dynamic range of floating-point values for your computer can be evaluated with the MATLAB commands `realmin` and `realmax`.

---

### Precision

Due to 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 MATLAB, floating-point relative accuracy is given by the command `eps` which returns the distance from 1.0 to the next largest floating point number. For a computer that supports the IEEE Standard for floating-point numbers, `eps` = $2^{-52}$ or $2.2204 \times 10^{-16}$.

### Data Type Parameters

The high and low limits, exponent bias, and precision for the floating-point formats supported by this blockset are given below.

**Table 3-4: Floating-Point Data Type Parameters**

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

Due to 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 754 Standard for binary floating-point arithmetic specifies practices and procedures so that predictable results are produced independent of the hardware platform. Specifically, denormalized numbers, infinity, and NaNs are defined to deal with exceptional arithmetic (underflow and overflow). These three aspects are discussed below.

### 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 incrementing the exponent for each shift until the exponent is within a

representable range. 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.

### Infinity

Arithmetic involving 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 (NaNs), any arithmetic operation involving infinity yields infinity. Infinity is represented by the largest biased exponent allowed by the format and a fraction of zero.

### NaNs

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

Both types of NaNs are represented by the largest biased exponent allowed by the format and a fraction that is nonzero. The bit pattern for a nonsignaling 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.

### Exceptional Arithmetic for Embedded Processors

If an overflow condition is handled as infinity or NaN, then significant processor overhead is required to deal with this exception. Although the IEEE 754 Standard specifies practices and procedures to deal with exceptional arithmetic in a consistent manner, microprocessor manufacturers may handle exceptional arithmetic conditions in ways that depart from the standard. Some of these alternative approaches such as saturation and wrapping are discussed in the next chapter.

4

# Arithmetic Operations


**Overview** . . . . . 4-2

**Limitations on Precision** . . . . . 4-3
Rounding . . . . . 4-3
Padding with Trailing Zeros . . . . . 4-8
Example: Limitations on Precision and Errors . . . . . 4-9
Example: Maximizing Precision . . . . . 4-10

**Limitations on Range** . . . . . 4-11
Saturation and Wrapping . . . . . 4-11
Guard Bits . . . . . 4-14
Example: Limitations on Range . . . . . 4-14

**Recommendations for Arithmetic and Scaling** . . . . 4-15
Addition . . . . . 4-15
Accumulation . . . . . 4-18
Multiplication . . . . . 4-19
Gain . . . . . 4-20
Division . . . . . 4-22
Summary . . . . . 4-24

**Parameter and Signal Conversions** . . . . . 4-25
Parameter Conversions . . . . . 4-26
Signal Conversions . . . . . 4-26

**Rules for Arithmetic Operations** . . . . . 4-29
Computational Units . . . . . 4-29
Addition and Subtraction . . . . . 4-29
Multiplication . . . . . 4-34
Division . . . . . 4-38
Shifts . . . . . 4-40

**Example: Conversions and Arithmetic Operations** . . . 4-45

# 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 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 introduces 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

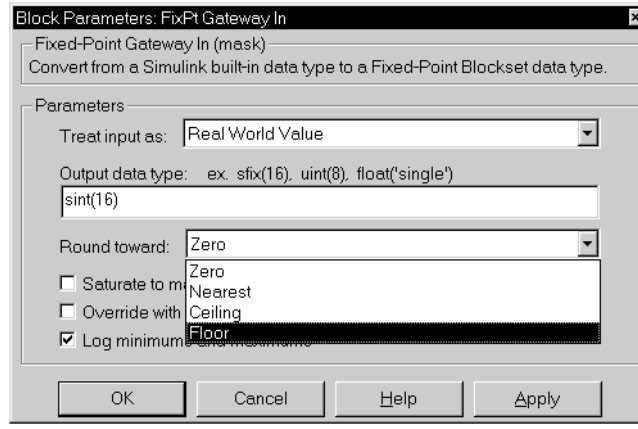
Computer words consist of a finite numbers of bits. This means that the binary encoding of variables is only an approximation of an arbitrarily precise real-world value. Therefore, the limitations of the binary representation automatically introduce limitations on the precision of the value.

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

## 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.

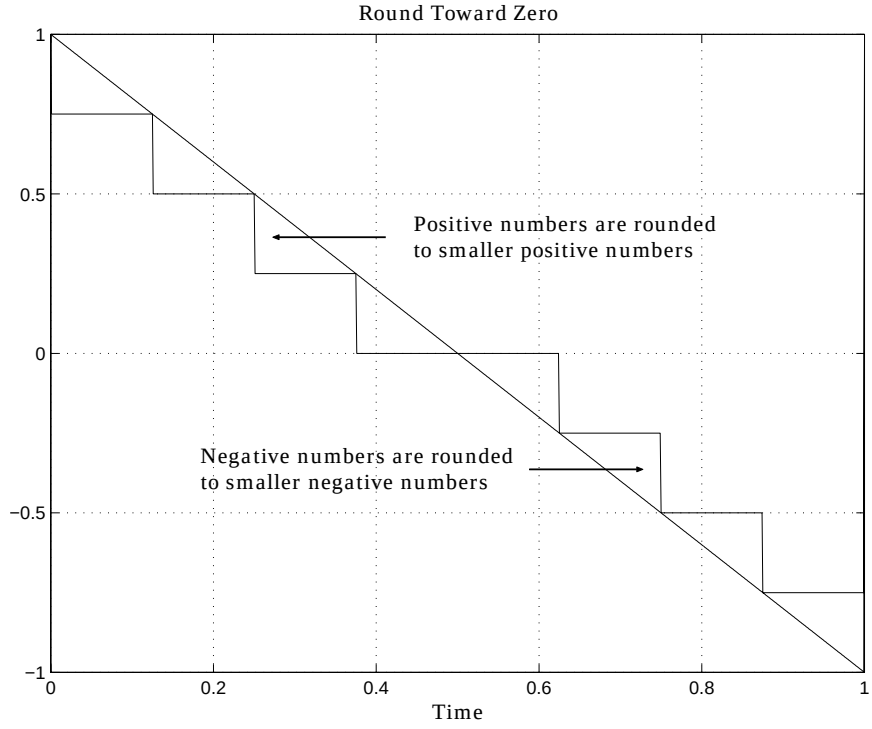
This blockset provides four rounding modes. These modes are available to you through the **Round toward** pull-down menu as shown below.



The Fixed-Point Blockset rounding modes are discussed below. Included with each discussion is a graph of Simulink doubles and equivalent Fixed-Point Blockset data using the specified rounding mode. The data is generated using Simulink's Signal Generator block and doubles are converted to signed 8-bit numbers with radix point-only scaling of $2^{-2}$.

## Round Toward Zero

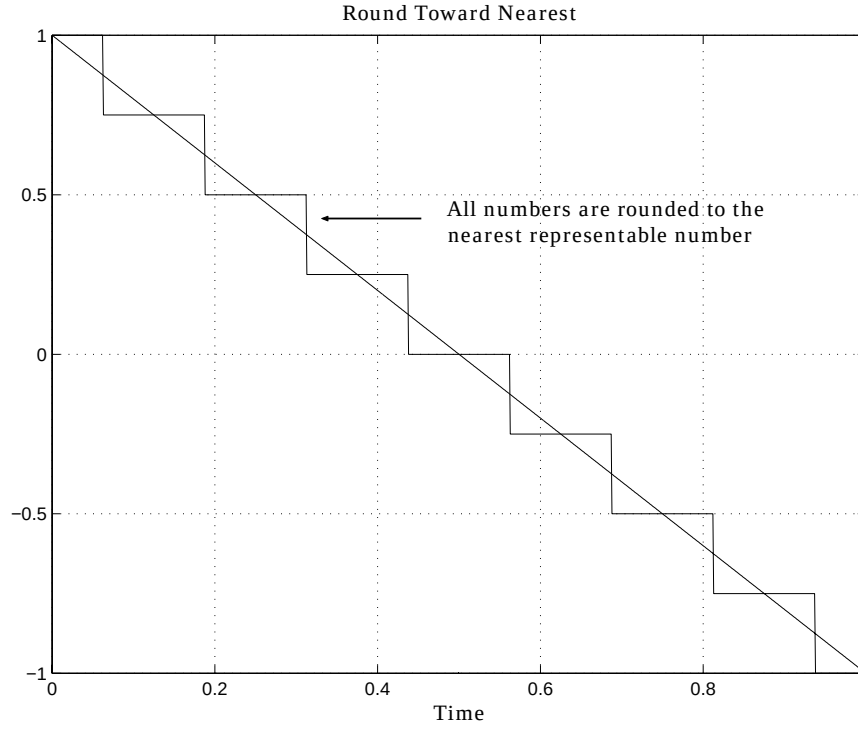
The computationally simplest rounding mode is to drop all digits beyond the number required. This mode is referred to as rounding toward zero, and it results in a number whose magnitude is always less than or equal to the more precise original value. Rounding toward zero introduces a cumulative downward bias in the result for unsigned numbers and a cumulative upward bias in the result for signed 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 illustrated below.



An example comparing rounding to zero and truncation for unsigned and two's complement numbers is given on page 4-8.

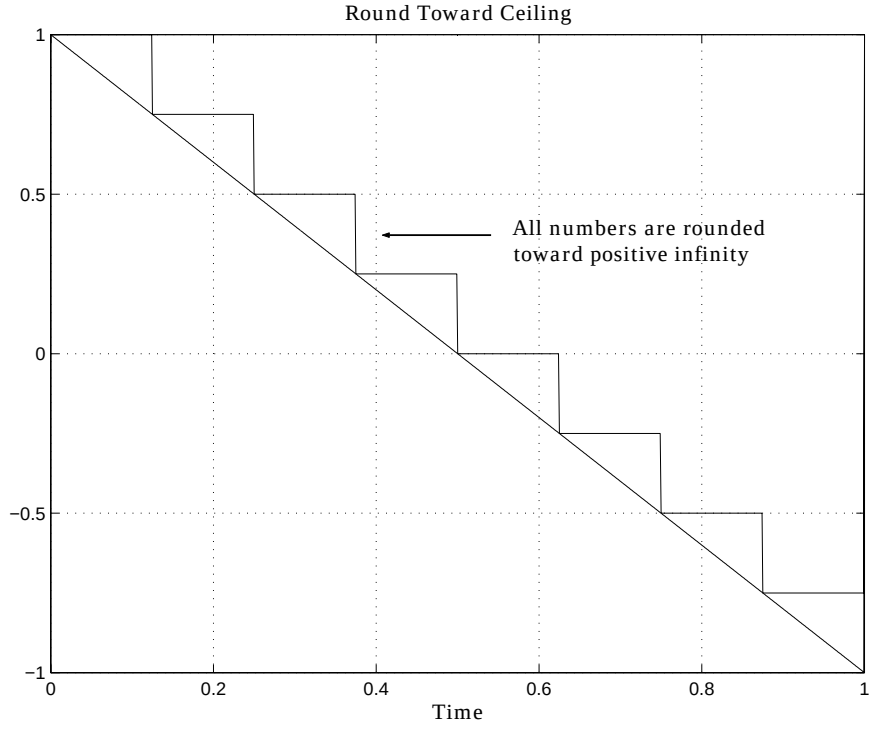
## Round Toward Nearest

When rounding 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. Rounding toward nearest is illustrated below.



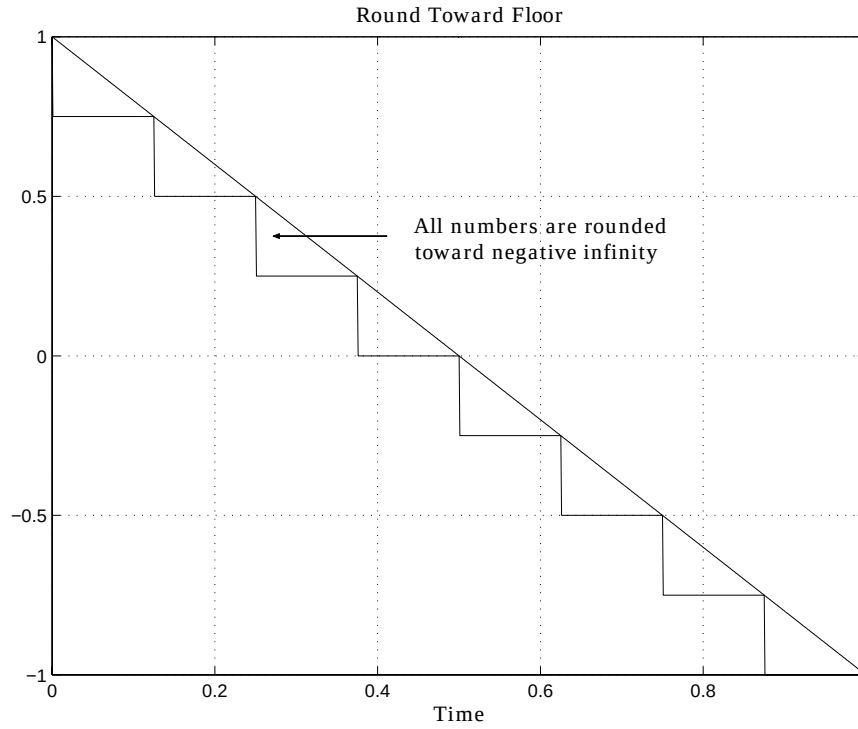
## Round Toward Ceiling

When rounding toward ceiling, both positive and negative numbers are rounded toward positive infinity. As a result, a positive cumulative bias is introduced in the number. Rounding toward ceiling is illustrated below.



## Round Toward Floor

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




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.

**Note:** In MATLAB, you can round to zero, nearest, ceil, and floor using the `fix`, `round`, `ceil`, and `floor` commands, respectively.

### 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 the blockset, and four explicit rounding modes are used instead.

## Padding with Trailing Zeros

Padding with trailing zeros involves extending the least significant bits (LSB's) of an arithmetic result 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.

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

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

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

which produces a more precise result. An example of padding with trailing zeros using the Fixed-Point Blockset is illustrated in “Digital Controller Realization” on page 6-7.

## 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 presented in “Scaling” on page 3-4, 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.

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

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.

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

## Example: Maximizing Precision

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

Assume the maximum and minimum real-world value is given by $max(V)$ and $min(V)$, respectively. These limits may 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 presented in "Scaling" on page 3-4, 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

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

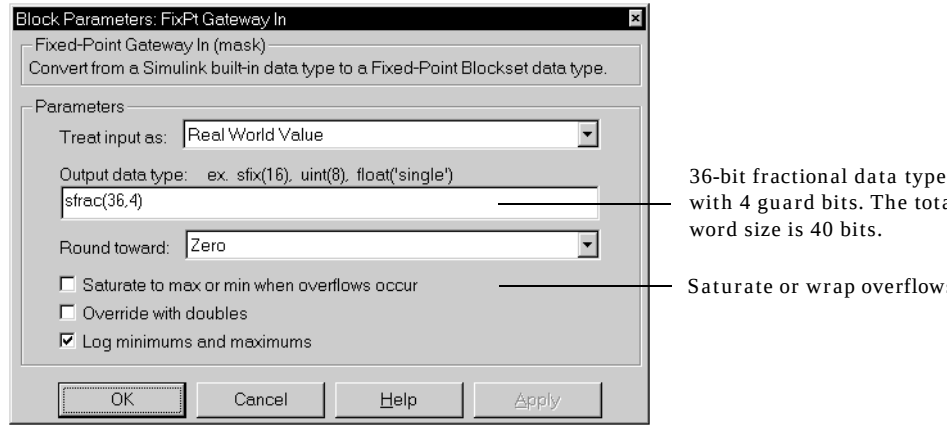
This formula is independent of rounding and overflow issues, and depends only on the word size, $ws$.

# Limitations on Range

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.

In binary arithmetic, it is often necessary for a processor to take an $n$-bit fixed-point number and store it in $m$-bits, where $m \neq n$. For the case where $m < n$, the range of the number has been reduced and an operation can produce an overflow condition. Some processors identify this condition as infinity or NaN. For other processors, especially digital signal processors (DSPs), the value *saturates* or *wraps*. In the case where $m > n$, 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 size of the word and the scaling.

Within this blockset, saturation and wrapping are supported for all fixed-point data types, while guard bits are supported for fractional data types only. Saturation and wrapping are available to you through the **Saturate to max or min when overflows occur** checkbox, and guard bits can be specified through the **Output data type** parameter as shown below.



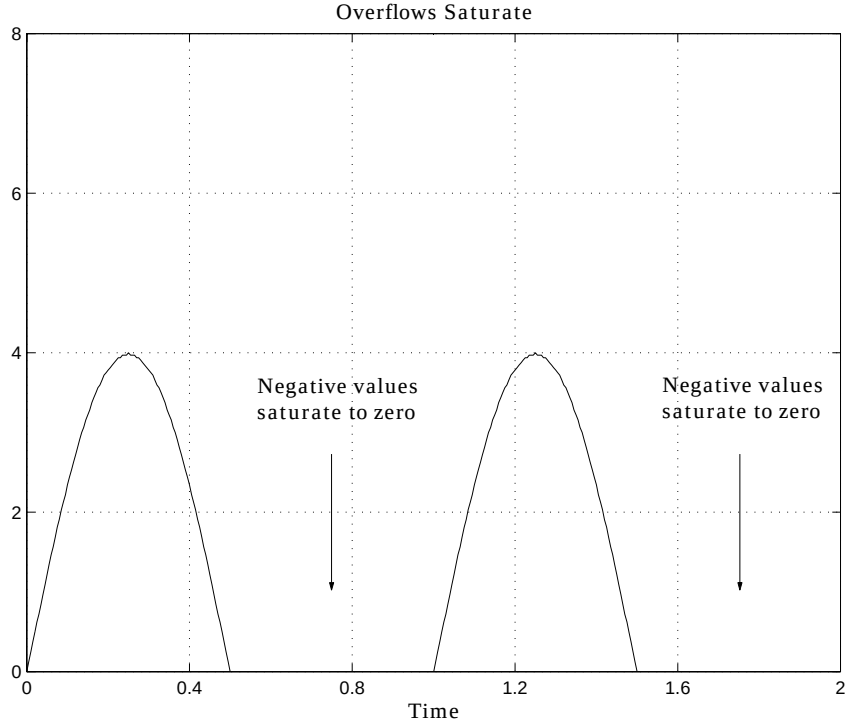

## Saturation and Wrapping

Saturation and wrapping describe a particular way that some processors deal with overflow conditions. For example, Analog Device's ADSP-2100 family of processors supports either of these modes.

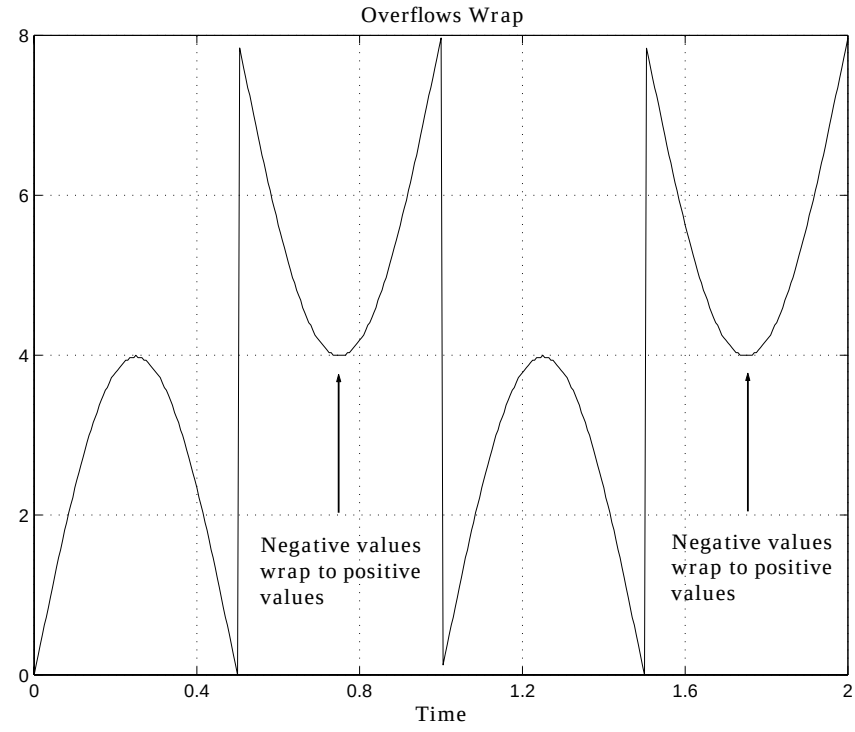
If a register has a saturation mode of operation, then an overflow condition is set to the maximum positive or negative value allowed. Conversely, if a register has a wrapping mode of operation, an overflow condition can be set to any value within the range of the representation.

### Example: Saturation and Wrapping

Consider an 8-bit unsigned word with radix 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 saturate to zero, which is the smallest number representable by the data type. The result when overflows saturate is shown below.



If overflows wrap, all negative values wrap to a positive value. The result when overflows wrap is shown below.




**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. When guard bits are specified for unsigned fractionals, they lie to the left of the default radix point. For signed fractionals, they lie to the left of the radix point and the sign bit.

## 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 presented in "Scaling" on page 3-4, 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

This section presents the relationship between arithmetic operations and fixed-point scaling, and some basic recommendations that may be appropriate for your fixed-point design. For each arithmetic operation:

- The general slope/bias encoding scheme presented in "Scaling" on page 3-4 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, radix 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 (i.e., 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 3-4.

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

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 multiplies 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 multiplies.

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. The maximum precision scaling can be determined if the range of the variable is known. As shown in “Example: Maximizing Precision”

on page 4-10, the range of a fixed-point operation can be determined from $max(\tilde{V}_a)$ and $min(\tilde{V}_a)$. For a summation, the range can be determined from

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

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

The maximum precision slope can now be derived.

$$F_a 2^{E_a} = \frac{max(\tilde{V}_a) - min(\tilde{V}_a)}{2^{ws_a} - 1}$$

$$= \frac{F_b 2^{E_b}(2^{ws_b} - 1) + F_c 2^{E_c}(2^{ws_c} - 1)}{2^{ws_a} - 1}$$

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 value 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

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

$$B_a \approx B_b + B_c$$

### Radix Point-Only Scaling

For radix 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 radix 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 multiply 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.

### Radix Point-Only Scaling

For radix 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 3-4.

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

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 multiply of a constant and two variables, two multiplies 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

The maximum precision scaling can be determined if the range of the variable is known. As shown in “Example: Maximizing Precision” on page 4-10, the range of a fixed-point operation can be determined from $max(\tilde{V}_a)$ and $min(\tilde{V}_a)$.

For multiplication, the range can be determined 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

$$V_{LL} = min(\tilde{V}_b) \cdot min(\tilde{V}_c)$$

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

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

$$V_{HH} = max(\tilde{V}_b) \cdot max(\tilde{V}_c)$$

### Radix Point-Only Scaling

For radix 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 multiply 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 radix point-only format. For each of these constants, the range is the trivial case of only one value. Despite the trivial range, the radix 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 operations involved.

### Inherited Scaling for Maximum Precision

The scaling for maximum precision does not need to be different than 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 will prevent saturation but will cause 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 3-4.

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

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 multiplies 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

The maximum precision scaling can be determined if the range of the variable is known. As shown in "Example: Maximizing Precision" on page 4-10, the range of a fixed-point operation can be determined from $max(\tilde{V}_a)$ and $min(\tilde{V}_a)$. For division, the range can be determined 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

$$V_{LL} = min(\tilde{V}_b) / min(\tilde{V}_c)$$

$$V_{LH} = min(\tilde{V}_b) / max(\tilde{V}_c)$$

$$V_{HL} = max(\tilde{V}_b) / min(\tilde{V}_c)$$

$$V_{HH} = max(\tilde{V}_b) / max(\tilde{V}_c)$$

### Radix Point-Only Scaling

For radix 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.
- Radix point-only scaling guarantees simpler math, but generally sacrifices some precision.
- It is important to note that the previous formulas don't show that
  - Constants and variables are represented with a finite number of bits.
  - Variables are either signed or unsigned.
  - The rounding and overflow handling schemes. These decisions must be made before an actual fixed-point realization is achieved.

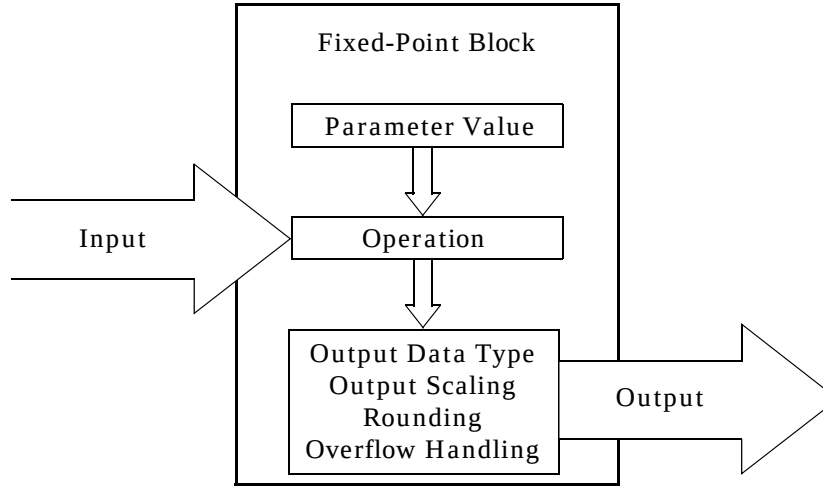
# Parameter and Signal Conversions

The bulk of this chapter, together with Chapter 3, presents how data types, scaling, rounding, overflow handling, and arithmetic operations are incorporated into the Fixed-Point Blockset. With this knowledge, you can control 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 the Fixed-Point Blockset, you must be aware of these three issues:

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

For example, suppose a particular fixed-point 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 diagram illustrates how these issues are related.




Parameter conversions and signal conversions are discussed below. Arithmetic operations are discussed in “Rules for Arithmetic Operations” on page 4-29.

## Parameter Conversions

Block parameters that accept numerical values are always converted from a double to a Fixed-Point Blockset 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 FixPt FIR block converts the initial condition parameter to the input data type, and converts the FIR coefficients parameter to a data type you explicitly specify through the block.

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.

For information about parameter conversions for a specific block, refer to Chapter 9, “Block Reference.”

### Offline Conversions

An offline conversion is a conversion performed by your development platform (e.g., 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 Blockset 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 3-4.

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

Solving for the output data type's stored integer value, $Q_a$

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

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

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. These operations are listed below:

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

# Rules for Arithmetic Operations

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.

This section describes the blockset-specific rules that are followed 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 followed by an example using the rules.

## Computational Units

The core architecture of many processors contains several computational units including arithmetic logic units (ALU's), multiplier/accumulator (MAC's), 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 Blockset 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 3-4.

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

The solution of the resulting equation for the stored integer, $Q_a$, is given on page 4-16. 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 operations are listed below:

**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.

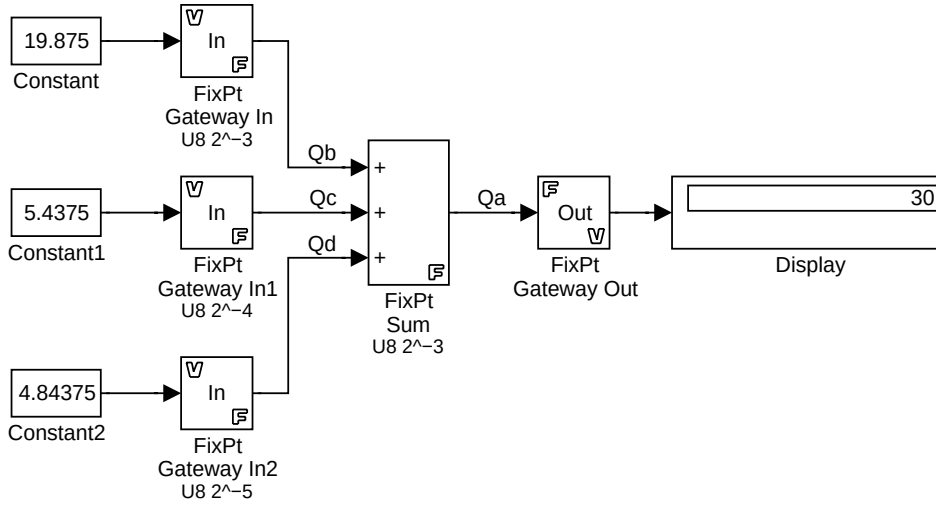
### Streamlining Simulations and Generated Code

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, therefore, be eliminated. Trivial steps in the summation process are eliminated for both simulation and code generation. Exclusive use of radix point-only scaling for both input signals and output signals is a common way to eliminate the occurrence of 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 radix point-only scaling . Additionally, the output is restricted to an 8-bit word with radix point-only scaling of $2^{-3}$.

The summation is shown below for the input values 19.875, 5.4375, and 4.85375.




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

**1** Since 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 radix 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 radix points requires that both the bits and the radix 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 since the bits and radix 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 radix points requires that both the bits and the radix 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 since the bits and radix 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 below, 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 FixPt Sum, FixPt Matrix Gain, and FixPt FIR 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.

For example, the Texas Instruments TMS320C2x family of processors performs two's complement 16-bit by 16-bit multiplication and produces a 32-bit (double bit) result.

## Fixed-Point Blockset 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 3-4.

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

The solution of the resulting equation for the output stored integer, $Q_a$, is given below and on page 4-19.

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

The worst-case implementation of this equation occurs when the slopes and biases of the input and output signals are mismatched. This worst-case implementation is permitted in simulation but is not permitted for code generation since it requires more resources than is considered practical for an embedded system. For code generation and bit-true simulations, all biases must be zero and the fractional slopes must match. When these requirements are met, the implementation reduces to

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

The bit-true implementation of this equation is discussed below.

**Offline Conversions.** As shown in the previous section, no offline conversions are performed.

**Online Conversions and Operations.** The online operations for matched slopes and biases of zero are listed below:

**1** The integer values, $Q_b$ and $Q_c$, are multiplied together.

$$Q_{RawProduct} = Q_b Q_c$$

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

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

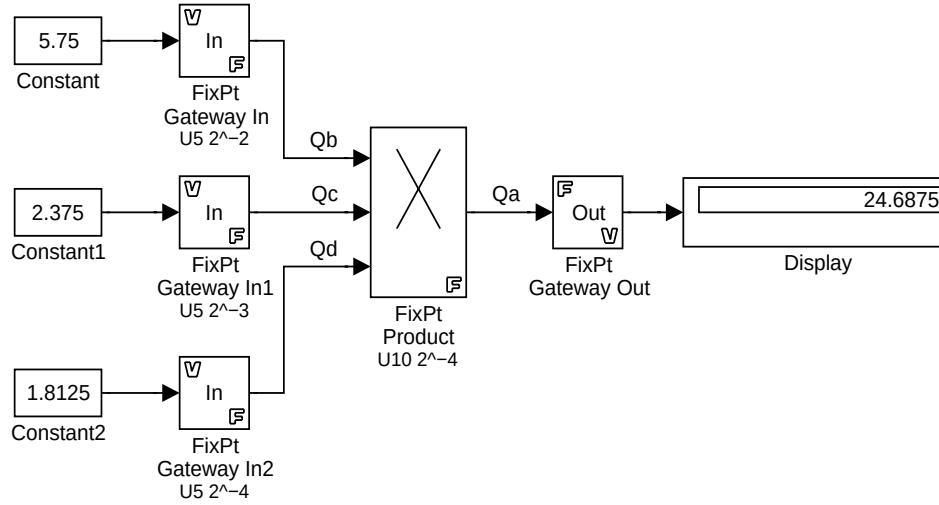
$$Q_a = convert(Q_{RawProduct})$$

This conversion includes any necessary bit shifting, rounding, or overflow handling. Conversions are discussed in "Signal Conversions" on page 4-26.

**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 radix point-only scaling. Additionally, the output is restricted to a 10-bit word with radix point-only scaling of $2^{-4}$. The multiplication is shown below 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 radix point of the product is given by the sum of the radix points of the multiplied numbers.

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

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

Conversions are discussed in “Signal Conversions” on page 4-26. 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.

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

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

**4** The product is converted to the output data type.

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

$$= 011000.1011 = 24.6875$$

Conversions are discussed in “Signal Conversions” on page 4-26. Note that a loss in precision of four 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 FixPt Product, FixPt FIR, FixPt Gain, and FixPt Matrix Gain blocks.

## Division

As with multiplication, division with mismatched scaling is complicated. Mismatched division is permitted for simulation only. For code generation and bit-true simulation, the signals must all have zero biases and matched fractional slopes.

### Fixed-Point Blockset 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 3-4.

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

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 below.

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

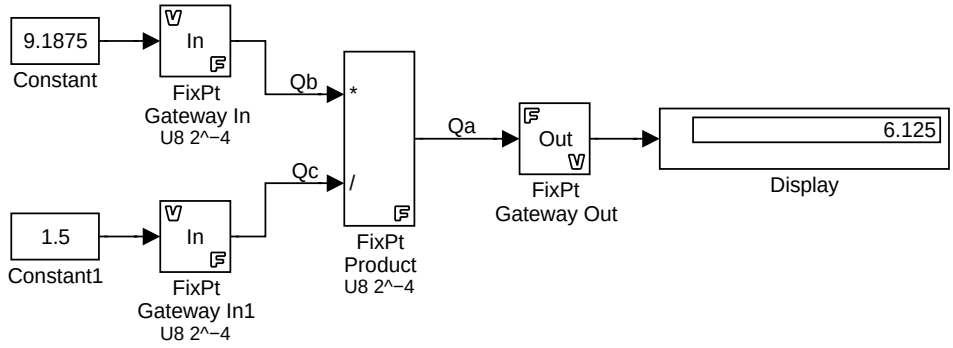
This equation involves an integer division and some bit shifts. If $E_a \geq E_b - E_c$, then any bit shifts are to the right and the implementation is simple. However, if $E_a < E_b - E_c$, then the bit shifts are to the left and the implementation can be more complicated. The essential issue is the 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, some of the fractional bits may be obtained by performing a shift prior to the integer division. In the worst case, it may 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. Division of signals with nonzero biases or mismatched slopes is not supported.

### 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 radix point-only scaling of $2^{-4}$. Additionally, the output is restricted to an 8-bit word with radix point-only scaling of $2^{-4}$.

The division of 9.1875 by 1.5000 is shown below.




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 date 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 below.

| Shift Operation | Binary Value | Decimal Value |
|---|---|---|
| N/A (original number) | 00110101 | 53 |
| Shift left by 2 bits | 11010100 | 212 |
| Shift right by 2 bits | 00001101 | 13 |

You can perform a shift with the Fixed-Point Blockset using either the FixPt Conversion block or the FixPt Gain block. The FixPt Conversion block shifts both the bits and radix point while the FixPt Gain block shifts the bits but not the radix point. These two modes of shifting as well as shifting to the right are discussed below.

**Note:** Performing a "plain" or "raw" machine-level shift such as those given in the example above with the Fixed-Point Blockset is complicated by the available scaling options. Therefore, a single "FixPt Shift" block is not provided. For more information about scaling, refer to "Scaling" on page 3-4.

## Shifting to the Right

Shifts to the right can be classified 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. With the Fixed-Point Blockset, shifting to the right follows these rules:

- For signed numbers, an arithmetic shift right is performed. Therefore, the most significant bit is recycled for each bit shift. For example, given the signed fixed-point number 10110.101, a bit shift two places to the right with the radix point unmoved yields the number 11101.101.
- For unsigned numbers, a logical shift right is performed. Therefore, the most significant bit is a 0 for each bit shift. For example, given the unsigned fixed-point number 10110.101, a bit shift two places to the right with the radix point unmoved yields the number 00101.101.

## Shifting Bits and the Radix Point

With the FixPt Conversion block, you can perform a shift operation on the input by specifying the appropriate radix point-only scaling for the output. This block shifts both the bits and the radix point.

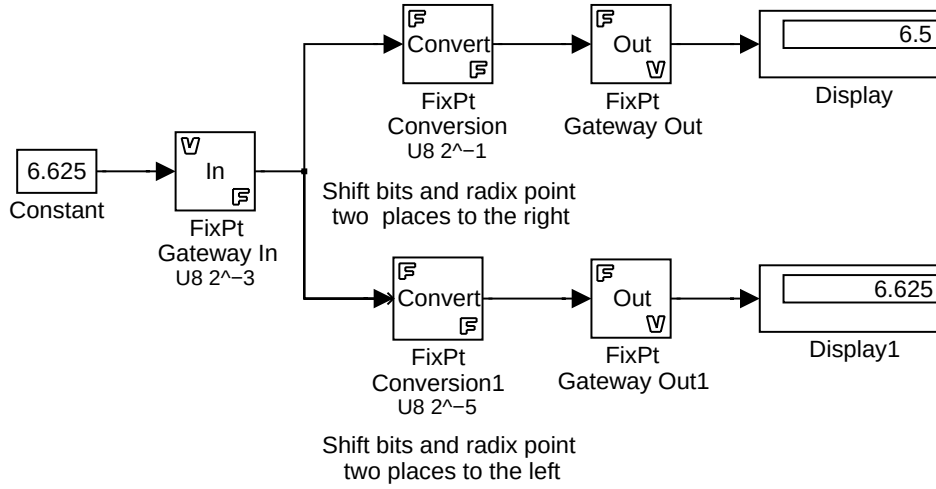
In most cases, you will perform a “plain” or “raw” shift. To perform such a shift using the FixPt Conversion block, the block's dialog box must be configured this way:

- The output data type is identical to the input data type.
- The rounding mode is set to `floor`. Therefore, bits simply fall off the left or fall off the right when a shift occurs.
- Overflows wrap.
- The output scaling is specified to reflect the required shift.

For example, suppose you start with the fixed-point number 00110.101 (a decimal value of 6.625), which is characterized by the blockset as an 8-bit unsigned, generalized fixed-point number with radix point-only scaling of $2^{-3}$. To shift the bits and radix point two places to the right, the input scaling of $2^{-3}$ is multiplied by $2^{2}$, which yields a scaling of $2^{-1}$. To shift the bits and radix point two places to the left, the input scaling of $2^{-3}$ is multiplied by $2^{-2}$, which yields as scaling of $2^{-5}$. This situation is shown below.

| Shift Operation | Block Scaling | Binary Value | Decimal Value |
|---|---|---|---|
| N/A (original number) | $2^{-3}$ | 00110.101 | 6.625 |
| Shift right by 2 bits | $2^{-1}$ | 0000110.1 | 6.5 |
| Shift left by 2 bits | $2^{-5}$ | 110.10100 | 6.625 |

The figure below shows the fixed-point model used to generate the above data.




Refer to Chapter 9, "Block Reference," for more information about the FixPt Conversion block.

## Shifting Bits but Not the Radix Point

With the FixPt Gain block, you can perform a shift operation on the input by specifying the gain as a power of two. This block shifts only the bits and not the radix point.

In most cases, you will perform a plain or raw shift. To perform such a shift using the FixPt Gain block, the block's dialog box must be configured this way:

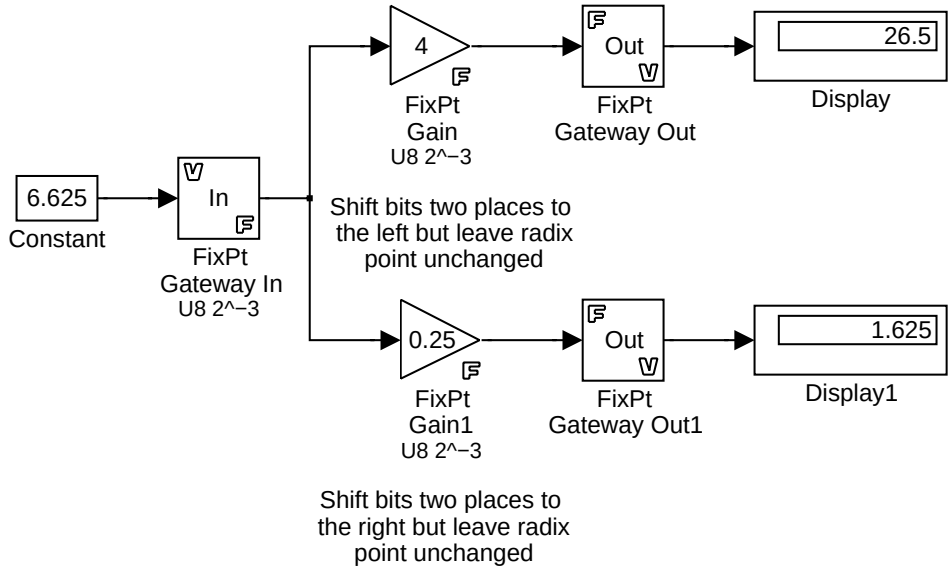
- The output data type is identical to the input data type.
- The rounding mode is set to `Floor`. Therefore, bits simply fall off the left or fall off the right when a shift occurs.
- Overflows wrap.
- The gain is specified as the appropriate power of 2 to reflect the required shift.

For example, suppose you start with the same fixed-point number, 00110.101, defined above. To shift the bits two places to the left, a gain of 4 is specified,

and to shift the bits two places to the right, a gain of 0.25 is specified. This situation is shown below.

| Shift Operation | Gain Value | Binary Value | Decimal Value |
|---|---|---|---|
| N/A (original number) | $2^{-3}$ | 00110.101 | 6.625 |
| Shift left by 2 bits | 4 | 11010.100 | 26.5 |
| Shift right by 2 bits | 0.25 | 00001.101 | 1.625 |

The figure below shows the fixed-point model used to generate the above data.




Refer to Chapter 9, “Block Reference,” for more information about the FixPt Gain block.

# Example: Conversions and Arithmetic Operations

This example uses the FixPt FIR 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 and subtraction, and multiplication are applied. For details about conversions and operations, refer to "Parameter and Signal Conversions" on page 4-25 and "Rules for Arithmetic Operations" on page 4-29.

---

**Note:** If a block can perform all four operations, such as the FixPt FIR block, then the rules for multiplication and division are applied first.

---

Suppose you configure the FixPt FIR block for two outputs (SIMO mode) where the first output is given by

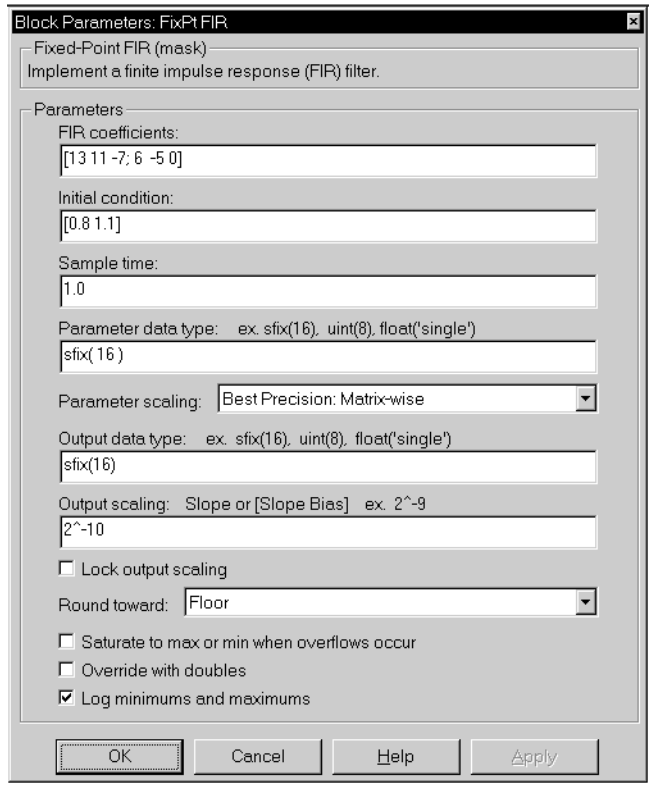
$$y_1(k) = 13 \cdot u(k) + 11 \cdot u(k-1) - 7 \cdot u(k-2)$$

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 radix point-only scaling.

To configure the FixPt FIR block for this situation, you must specify the **FIR coefficient** parameter as `[13 11 -7; 6 -5 0]` and the **Initial condition** parameter as `[0.8 1.1]` as shown below in the dialog box below.




Parameter conversions and block operations are given below in the order in which they are carried out by the FixPt FIR block.

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

   The **Initial condition** parameter is converted from doubles to the input data type offline 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** 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 since the coefficients and inputs are already converted to the **Output data type**.

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

5

# Realization Structures

**Overview** . . . . . 5-2

**Realizations** . . . . . 5-4
Direct Form II . . . . . 5-4
Series Cascade Form . . . . . 5-7
Parallel Form . . . . . 5-9

# Overview

This chapter investigates how you can realize digital filters using the Fixed-Point Blockset.

The Fixed-Point Blockset applies to the control system and signal processing fields, and other fields where algorithms are implemented on fixed-point hardware. In signal processing, a digital filter is a computational algorithm that converts an input number sequence to an output sequence. This algorithm is designed such that the output signal meets frequency-domain and/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 may contain nonlinear functions such as lookup tables in addition to a discrete transfer function. The term *digital filter* will be used when referring to discrete transfer functions.

This blockset does not attempt to standardize on one particular fixed-point digital filter design method. For example, a design can be done in continuous time and an "equivalent" discrete-time digital filter can be obtained using one of many transformation methods. Alternatively, digital filters can be directly designed in discrete time. After the digital filter is obtained, it can be realized for fixed-point hardware using any number of canonical forms (also referred to as the digital filter structure). Typical canonical forms are the direct form, series form, and parallel form, all of which are outlined in this chapter.

For a given digital filter, the canonical forms describe a set of fundamental operations for the processor. Since there are an infinite number of ways to realize a given digital filter, the best realization must be made 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 finite length binary words used to represent the real numbers will introduce errors. Some realizations are more sensitive to these errors than others.

The Fixed-Point Blockset allows you to evaluate various digital filter realization methods in a simulation environment. Following the development cycle outlined on page 1-5, you can fine tune the realizations with the goal of reducing the cost (code and data size) and/or increasing signal quality. After the desired performance has been achieved, you can use the Real-Time Workshop to generate rapid prototyping C code and evaluate its performance with respect to your system's real-time timing constraints. You can alter the model based upon feedback from the rapid prototyping system. When you are satisfied with the performance of the rapid prototyping system, you can modify the code for production using the model and generated code as a specification.

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 Fixed-Point Blockset for this transfer function.

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

# Realizations

This section describes the direct form II, series cascade form, and parallel form realization structures. Each of these realization structures reflects a digital filter design method with specific advantages and disadvantages. Since there is no one universal realization method (in fact there are an infinite number of realization methods), these advantages and disadvantages will determine whether a given structure suits your particular needs. Of course, if none of these structures suits your needs, you can use the Fixed-Point Blockset to build you own structures.

The realization methods presented here are formulated with respect to some criteria such as minimal number of delay elements. Each method is presented as a conceptual block diagram and a Fixed-Point Blockset model that you can use and modify as needed. These models are available to you in the demos directory.

## 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.

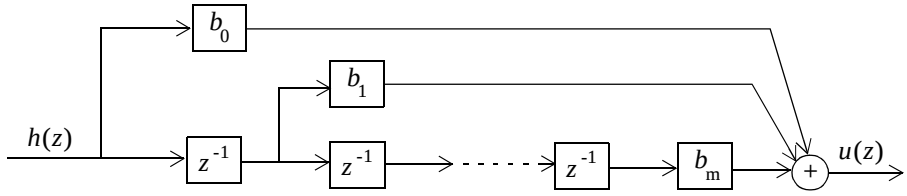
The canonical direct form II is presented as “Standard Programming” in *Discrete-Time Control Systems* by Ogata. It is known as the “Control Canonical Form” in *Digital Control of Dynamic Systems* by Franklin, Powell, and Workman.

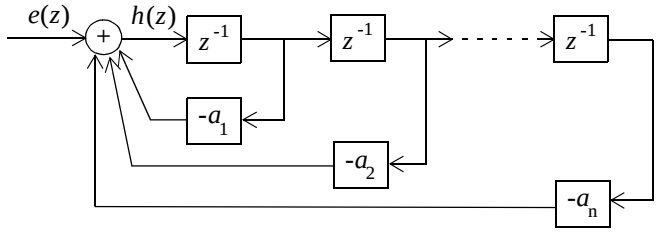
The canonical direct form II realization can be derived by writing the discrete-time transfer function with input $e(z)$ and output $u(z)$ as

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

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

The block diagrams for $u(z)/h(z)$ and $h(z)/e(z)$ are shown below.

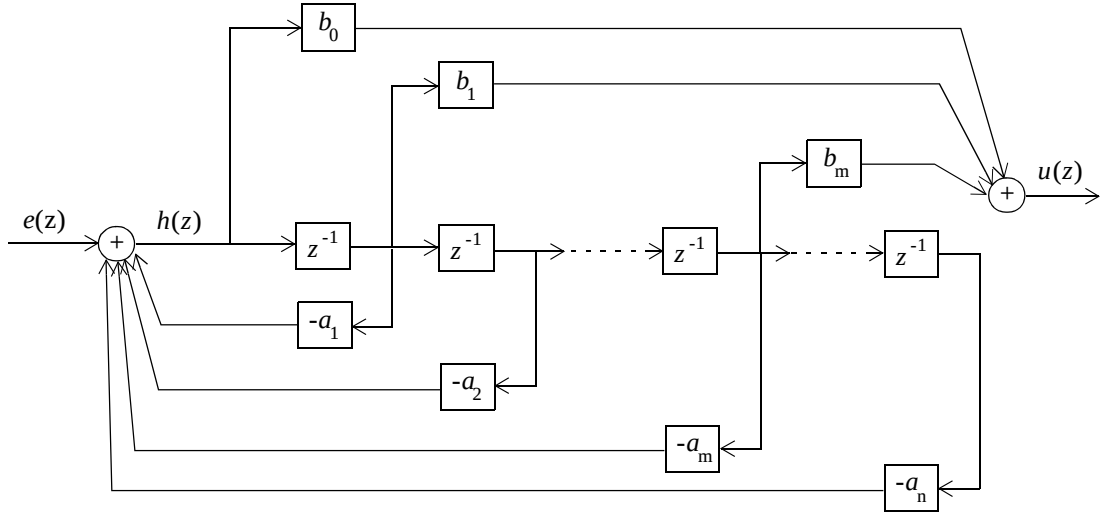



$$\frac{u(z)}{h(z)} = b_0 + b_1 z^{-1} + \ldots + b_m z^{-m}$$

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

Combining these two block diagrams yields the direct form II diagram shown below. Notice that the feed-forward part (top of block diagram) contains the numerator coefficients and the feedback part (bottom of block diagram) contains the denominator coefficients.

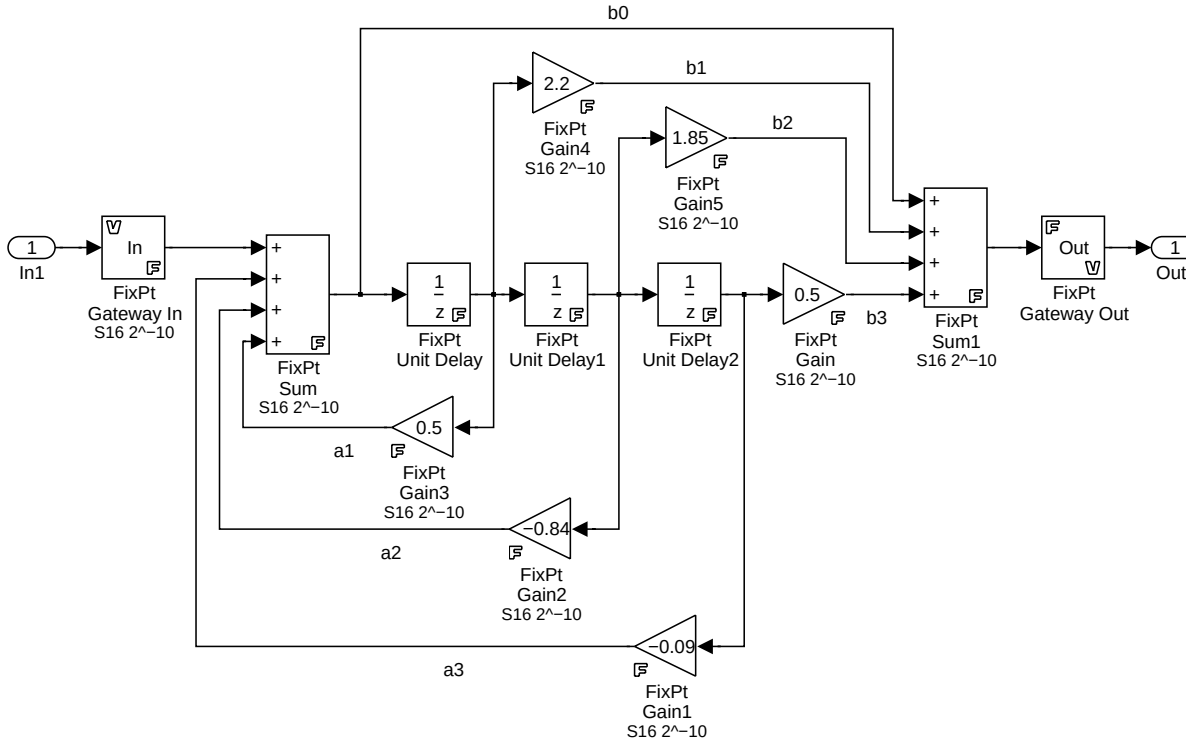



$\frac{u(z)}{e(z)}$

The direct form II realization of the transfer function $H_{ex}(z)$, where

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

using the Fixed-Point Blockset is shown below.



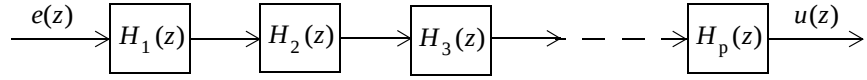

This model, `fxpdemo_direct_form2.mdl`, can be found in the `fxpdemos` directory.

## Series Cascade Form

In the canonical series cascade form, the transfer function, *H(z)*, is written as a product of first-order and/or 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 shown below.



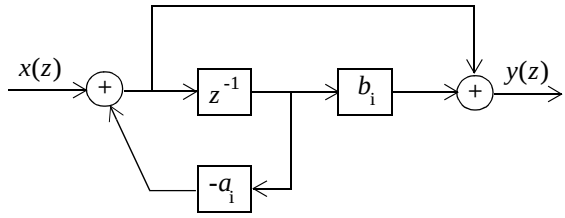

Factoring *H(z)* into $H_i(z)$ where $i = 1,2,3,\ldots,p$ can be done in a number of ways. Using the poles and zeros of *H(z)*, you can obtain $H_i(z)$ by grouping pairs of conjugate complex poles and pairs of conjugate complex zeros to produce second-order transfer functions, or by grouping real poles 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 $H_i(z)$, it is desirable to compare the various groupings to see which produces the best results for the transfer function under consideration.

For example, one factorization of *H(z)* might be

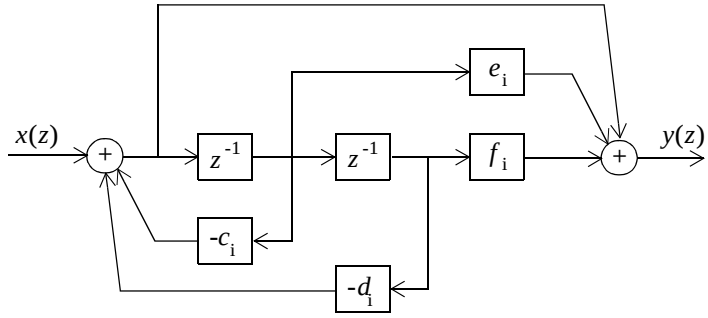
$$\begin{aligned} H(z) &= H_1(z)H_2(z)\ldots H_p(z) \\ &= \prod_{i=1}^{j} \frac{1 + b_i z^{-1}}{1 + a_i z^{-1}} \prod_{i=j+1}^{p} \frac{1 + e_i z^{-1} + f_i z^{-2}}{1 + c_i z^{-1} + d_i z^{-2}} \end{aligned}$$

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

The first and second order diagrams for the series cascade form are given below.




$$\frac{y(z)}{x(z)} = \frac{1 + b_i z^{-1}}{1 + a_i z^{-1}}$$

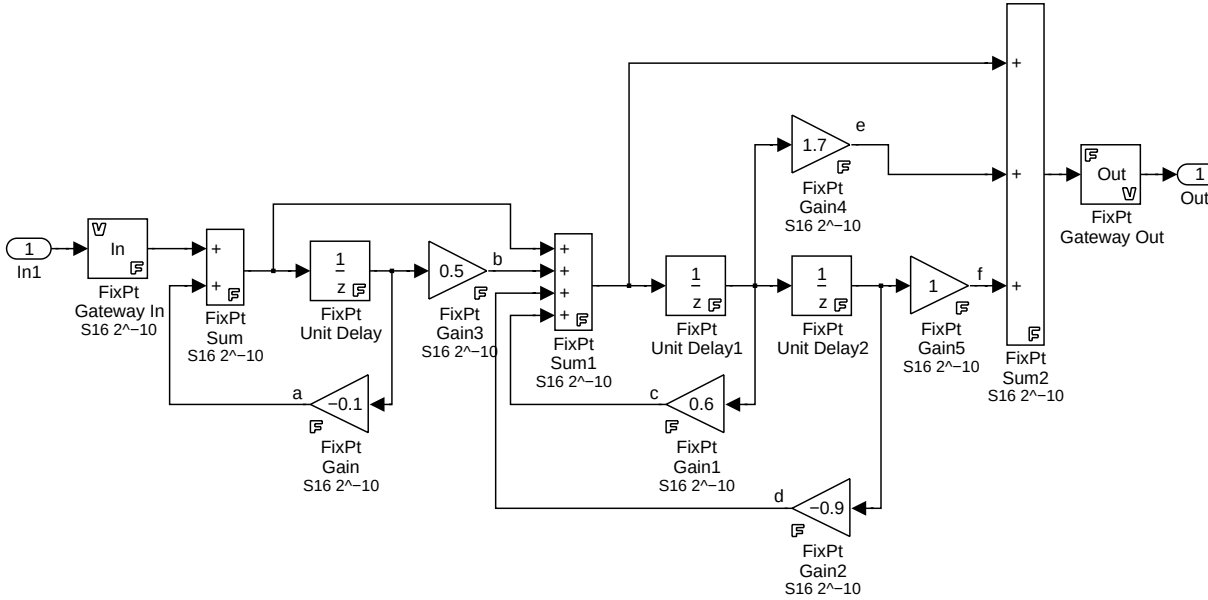



$$\frac{y(z)}{x(z)} = \frac{1 + e_i z^{-1} + f_i z^{-2}}{1 + c_i z^{-1} + d_i z^{-2}}$$

The series cascade form realization of transfer function $H_{ex}(z)$, where

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

using the Fixed-Point Blockset is shown below.



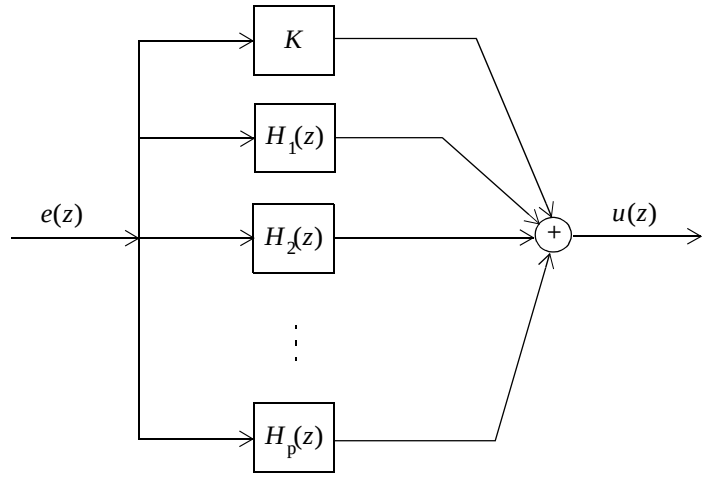

This model, `fxpdemo_series_cascade_form.mdl`, can be found in the `fxpdemos` directory.

## 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/or second-order transfer functions as shown below.

$$H_i(z) = \frac{u(z)}{e(z)} = K + H_1(z) + H_2(z) + \ldots + H_p(z)$$

This expansion, where $K$ is a constant and the $H_i(z)$ are the first and/or second-order transfer functions, is shown below.

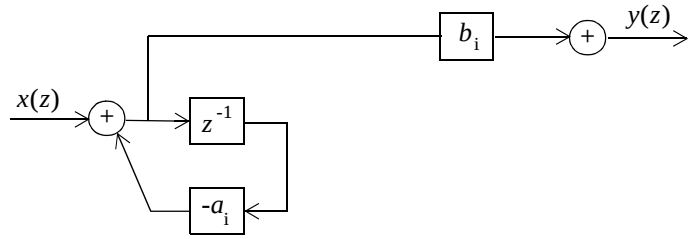



As in the series canonical form, there is no one unique description for the first-order and second-order transfer function. Due to the nature of the FixPt Sum block, the ordering of the individual filters doesn't matter. However, because of the constant $K$, the first-order and second-order transfer functions can be chosen such that their forms are simpler than those for the series cascade form as shown in the preceding section. This is done by expanding $H(z)$ as

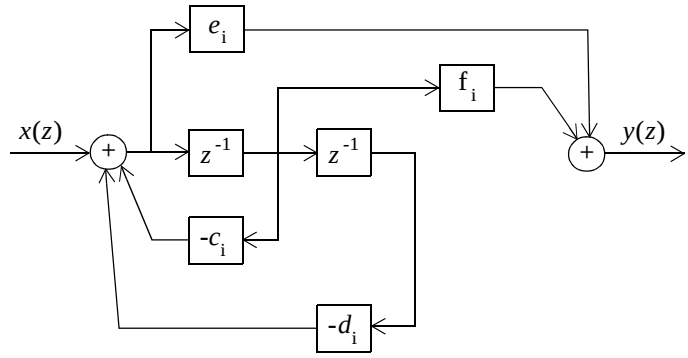
$$H(z) = K + \sum_{i=1}^{j} H_i(z) + \sum_{i=j+1}^{p} H_i(z)$$

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

The first and second order block diagrams for this expansion are given below.




$$\frac{y(z)}{x(z)} = \frac{b_i}{1 + a_i z^{-1}}$$

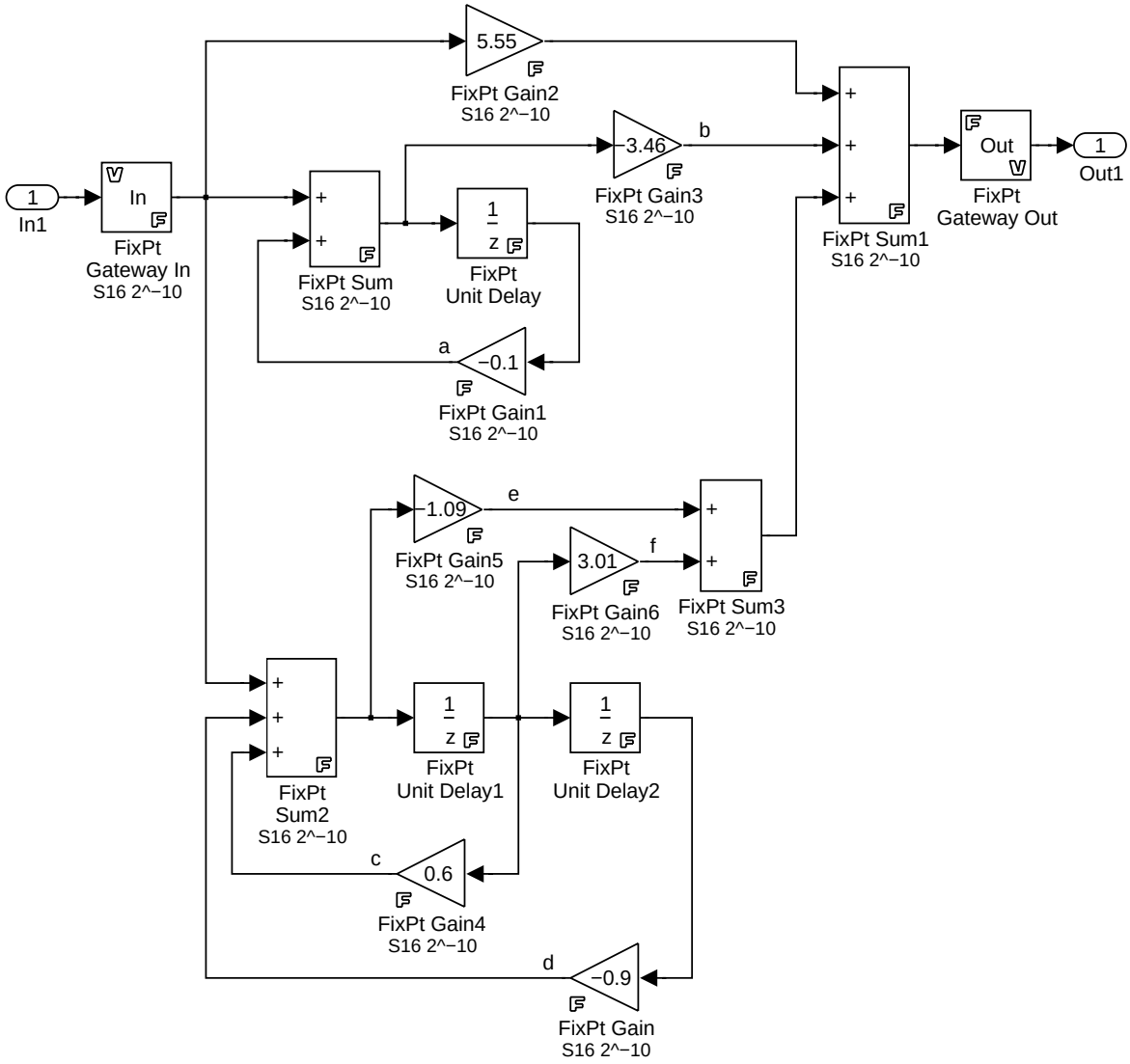



$$\frac{y(z)}{x(z)} = \frac{e_i + f_i z^{-1}}{1 + c_i z^{-1} + d_i z^{-2}}$$

The parallel form realization of the transfer function $H_{ex}(z)$, where

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

using the Fixed-Point Blockset is shown below.




This model, `fxpdemo_parallel_form.mdl`, can be found in the `fxpdemos` directory.

6

# Tutorial: Feedback Controller Simulation

**Overview** . . . . . 6-2

**Simulink Model of a Feedback Design** . . . . . 6-3

**Idealized Description of a Feedback Design** . . . . . 6-6

**Digital Controller Realization** . . . . . 6-7

**Simulation Results** . . . . . 6-9
Simulation 1: Initial Guess at Scaling . . . . . 6-9
Simulation 2: Global Override . . . . . 6-11
Simulation 3: Automatic Scaling . . . . . 6-11
Simulation 4: Individual Override . . . . . 6-16

# Overview

The purpose of this tutorial is to show you how to use the Fixed-Point Blockset to simulate a fixed-point feedback design. In doing so, many of the essential blockset features are demonstrated. These include:

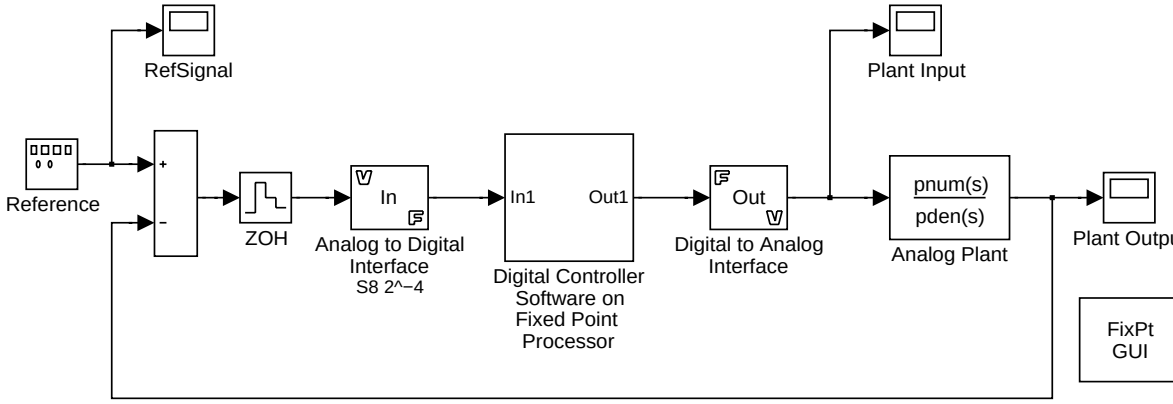
- Output data type selection
- Output scaling
- Logging maximum and minimum simulation results

Although these features are discussed in other chapters, they are demonstrated here as a collective whole. In addition to these main features, this tutorial demonstrates:

- The automatic scaling tool
- The utility of global data type override
- The utility of the **Override with doubles** checkbox for debugging purposes

# Simulink Model of a Feedback Design

The Simulink model of the feedback design, `fxpdemo_feedback.mdl`, can be accessed via the Demos block which is found in the fixed-point library. The demo is titled Automatic Scaling in Feedback Control. Alternatively, you can access the model directly by typing its name at the command line. However, to run the model this way, you must also invoke `preload_feedback.m` which populates the workspace with the required parameter values. The model is shown below.




The model consists of these blocks:

- **Reference**

  Simulink's Signal Generator block generates a continuous-time reference signal. It is configured to output a square wave.

- **ZOH**

  Simulink's Zero-Order Hold block samples and holds the continuous signal. This block is configured so that it quantizes the signal in time by an amount `tsamp` = 0.01 seconds.

- **Analog to Digital Interface**

  The analog to digital (A/D) interface consists of a FixPt Gateway In block that converts a Simulink double to a Fixed-Point Blockset data type. It represents any hardware that digitizes the amplitude of the analog input signal (an A/D hardware). In the real world, its characteristics are fixed.

- **Digital Controller**

  The digital controller is a masked subsystem that represents the software running on the hardware target. It is discussed in detail in “Digital Controller Realization” on page 6-7.

- **Digital to Analog Interface**

  The digital to analog interface consists of a FixPt Gateway Out block that converts a Fixed-Point Blockset data type into a Simulink double. It represents any hardware that converts a digitized signal into an analog signal (a D/A converter). In the real world, its characteristics are fixed.

- **Analog Plant**

  The analog plant is described by a transfer function and is the object controlled by the digital controller. In the real world, its characteristics are fixed.

## Simulation Setup

Setting up the fixed-point feedback controller simulation involves these steps:

**1 Identify all design components**

In the real world, there are design components with fixed characteristics (hardware) and design components with characteristics that can be changed (software). The main components modeled in this feedback design are the analog to digital (A/D) hardware, digital controller, digital to analog (D/A) hardware, and analog plant.

**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 and will not be 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**

This is accomplished with a Bode plot. The evaluation is idealized since all numbers, operations, and states are double precision.

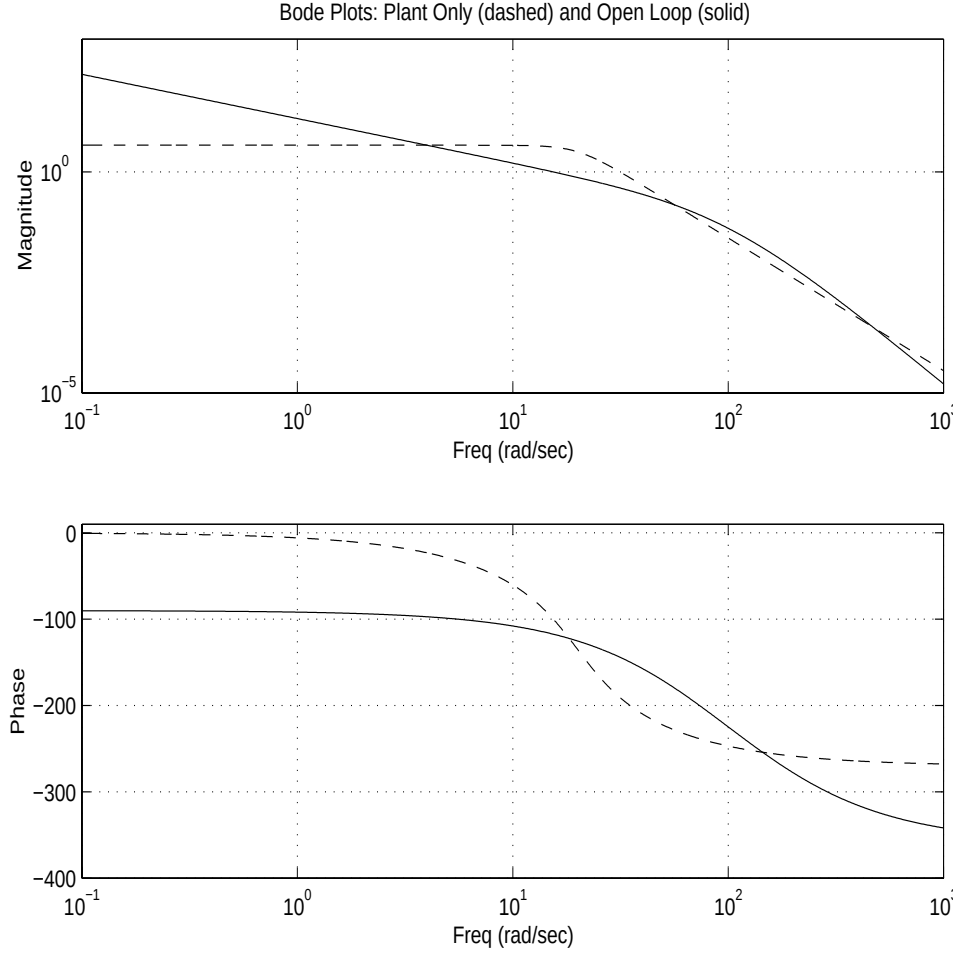
**4 Simulate the system**

The feedback controller design is simulated using Simulink and the Fixed-Point Blockset. Of course, in a simulation environment, all components (software *and* hardware) can be treated as though their characteristics are not fixed.

# Idealized Description of a Feedback Design

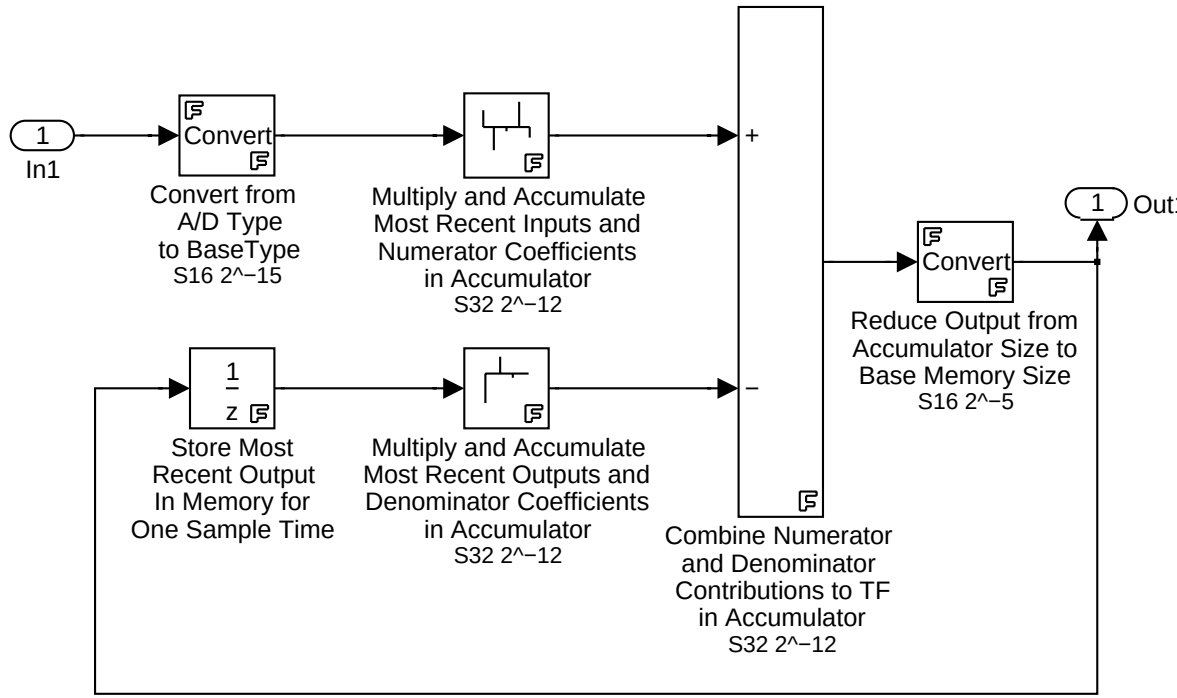
The open loop (controller + plant) and plant-only Bode plots shown below are automatically generated by the Automatic Scaling in Feedback Control demo (or by invoking `preload_feedback.m`). The open loop Bode plot results from a digital controller described in the idealized world of continuous time as well as double precision coefficients, storage of states, and math operations.

The plant and controller design criteria are not important for the purposes of this tutorial.



# Digital Controller Realization

The digital controller is implemented using a fixed-point direct form realization. The 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. The realization is shown below.




The realization consists of these blocks:

- **FixPt Conversion**

  These blocks represent RAM. The first FixPt Conversion block 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). The second FixPt Conversion block 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.

- **FixPt FIR**

  These blocks represent a weighted sum carried out in the CPU target. The word size and precision reflect those of the accumulator. The first FixPt FIR block multiplies and accumulates the most recent inputs with the FIR numerator coefficients. The second FixPt FIR block multiples and accumulates the most recent delayed inputs with the FIR denominator coefficients.

- **FixPt Sum**

  This block represents the accumulator in the CPU. Its word size and precision are twice that of the RAM (double bits).

- **FixPt Unit Delay**

  This block delays the feedback signal in memory by one sample period.

### 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 for the previous time step.
- $y(k)$ represents the current output, and $y(k-1)$ represents the output for the previous time step.
- $b_i$ represent the FIR numerator coefficients.
- $a_i$ represent 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 inputs and denominator coefficients in the accumulator. Since 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).

# Simulation Results

Using Simulink and the Fixed-Point Blockset, you can easily transition from a digital controller described in the ideal world of double precision numbers to one realized in the world of fixed-point numbers. The simulation approach used in this tutorial follows these steps:

1. Take an initial guess at the scaling. For this tutorial, an initial "proof of concept" simulation using a reasonable guess at the fixed-point word size and scaling was the first step in simulating the digital controller. This step is included only to illustrate how difficult it is to guess the best scaling.

2. Perform a global override of the fixed-point data types and scaling using double precision numbers. The maximum and minimum simulation values for each digital controller block are logged to the workspace.

3. Use the automatic scaling M-file script. This script uses the doubles simulation values previously logged to the MATLAB workspace, and changes the scaling for each block that does not have its scaling fixed.

4. Perform a simulation on the "fixed" hardware block by overriding the data type with doubles. This simulation will determine whether the A/D hardware warrants modification or replacement.

The four simulation trials are described below. The quality of the simulation results is determined by examining the input and output of the analog plant.

---

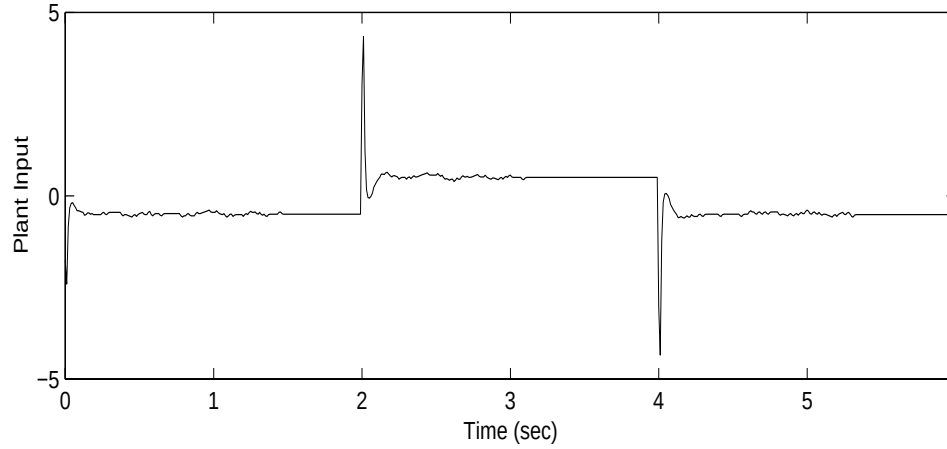
**Note:** The four steps described above can be performed with the FixPt GUI block provided with the model.

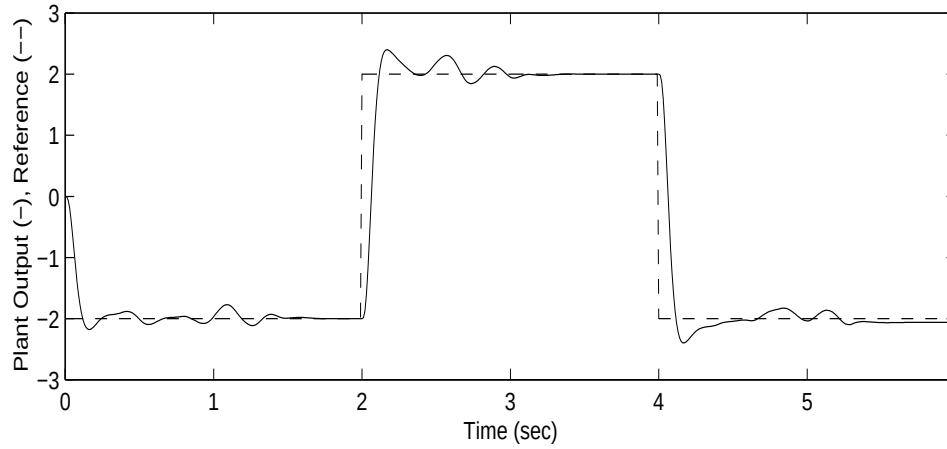
---

## Simulation 1: Initial Guess at Scaling

The first simulation uses guesses for the scaling. These initial scaling values are shown in the previous figure of the feedback system and digital controller in "Digital Controller Realization" on page 6-7. In general, you won't need to perform this step, and this simulation is included to illustrate the difficulty of guessing at scaling.

The input and output of the analog plant are shown below.




The Bode plot design sought to produce a well behaved linear response for the closed loop system. Clearly, the response is nonlinear. The nonlinear features are due to significant quantization effects. An important part of fixed-point design is finding scalings that reduce quantization effects to acceptable levels.

## Simulation 2: Global Override

Prior to using the automatic scaling tool, a global override with doubles of the fixed-point data type is performed for every block that does not have its **Lock output scaling** checkbox checked. Using this feature, accurate simulation limits can be obtained. Global override with doubles is accomplished with the `FixUseDbl` variable.

```
global FixUseDbl
FixUseDbl = 1
```

Maximum and minimum simulation values must be logged for all blocks that are to be scaled. This is accomplished by checking the **Log minimums and maximums** checkbox.

The simulation is run and maximum/minimum values are recorded. These values, displayed using the `showfixptsimranges` script, are given below.

**Table 6-1: Digital Controller Global Override Simulation Results**

| Block | Block Description | Min | Max |
|---|---|---|---|
| FixPt Conversion | Convert from the A/D data type to base data type | -2.0000 | 4.0000 |
| FixPt FIR | Multiply and accumulate most recent inputs and numerator coefficients in accumulator | -3.4465 | 3.5440 |
| FixPt FIR | Multiply and accumulate most recent outputs and denominator coefficients in accumulator | -8.5165 | 4.7503 |
| FixPt Sum | Combine numerator and denominator contributions to the transfer function and accumulate | -3.3674 | 4.3269 |
| FixPt Conversion | Reduce output from accumulator data type to base data type | -2.4135 | 4.3269 |

## Simulation 3: Automatic Scaling

Using the automatic scaling tool, 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 tool can make achieving this goal tedious and time consuming.

Automatic scaling is performed for the fixed-point digital controller block. This block is a masked subsystem representing software running on the target, and requires optimization. Automatic scaling consists of these steps:

1. Global override with doubles is turned off.

   ```
   FixUseDbl = 0
   ```

2. The scaling is set so the largest simulation value seen is 20% smaller than the maximum value allowed.

   ```
   RangeFactor = 1.20
   ```

   `RangeFactor` simply multiplies the "raw" simulation values by the specified value. Setting `RangeFactor` to a value greater than one guarantees the simulation covers the largest possible range, although it does not necessarily mean the resolution improves. Since there is always some uncertainty when representing a real-world value with a fixed-point number with only a few simulations, using `RangeFactor` is recommended.

3. The `autofixexp` M-file script is run. This script automatically changes the scaling on all fixed-point blocks that do not have their scaling locked, and that have their output data type specified as a generalized fixed-point number. It uses the minimum and maximum data logged from the last simulation run. The scaling changes such that the precision is maximized while the full range of simulation values are spanned for each block.

The automatic scaling results for the digital controller are given below. These are the raw results before `RangeFactor` is applied.

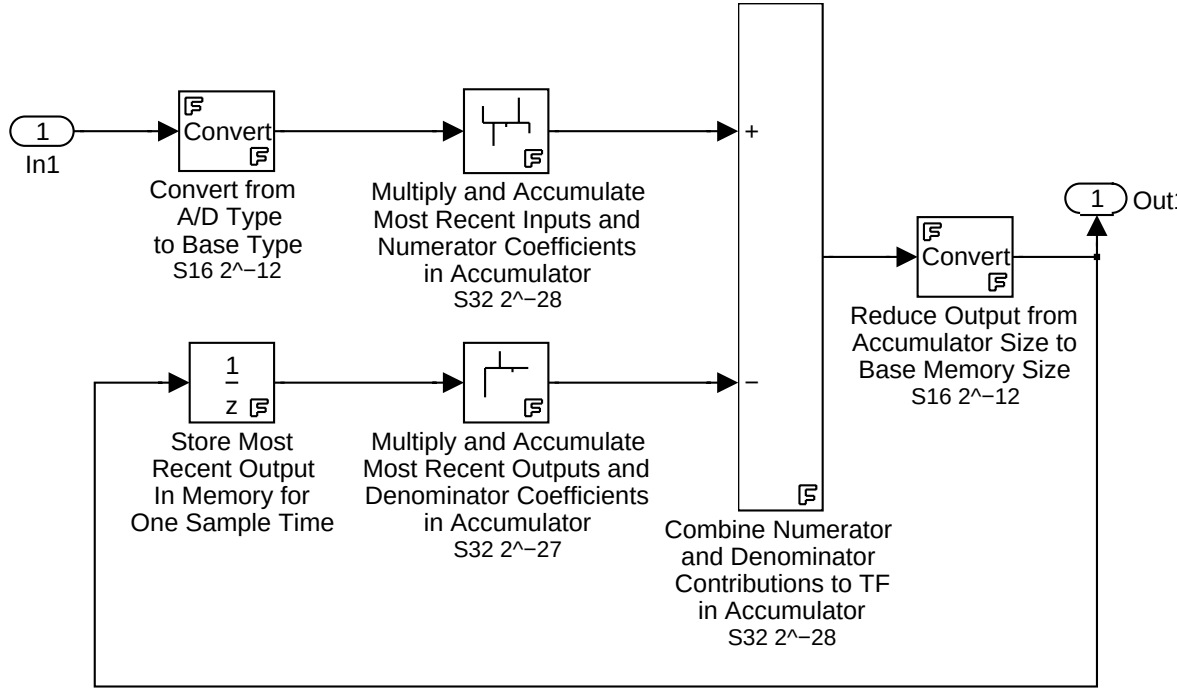
**Table 6-2: Digital Controller Automatic Scaling Results**

| Block | Block Description | Min | Max |
|---|---|---|---|
| FixPt Conversion | Convert from the A/D data type to base data type | -2.0000 | 4.0000 |
| FixPt FIR | Multiply and accumulate most recent inputs and numerator coefficients in accumulator | -3.4466 | 3.5440 |
| FixPt FIR | Multiply and accumulate most recent outputs and denominator coefficients in accumulator | -8.5516 | 4.7504 |

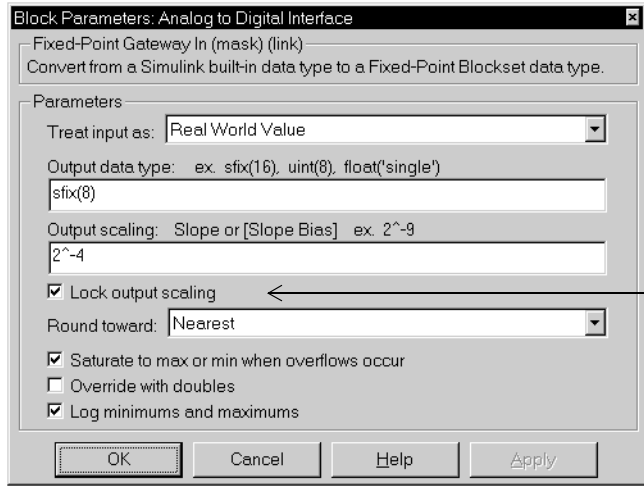
**Table 6-2: Digital Controller Automatic Scaling Results (Continued)**

| Block | Block Description | Min | Max |
|---|---|---|---|
| FixPt Sum | Combine numerator and denominator contributions to the transfer function and accumulate | -3.3674 | 4.3270 |
| FixPt Conversion | Reduce output from accumulator size to base memory size | -2.4135 | 4.3270 |

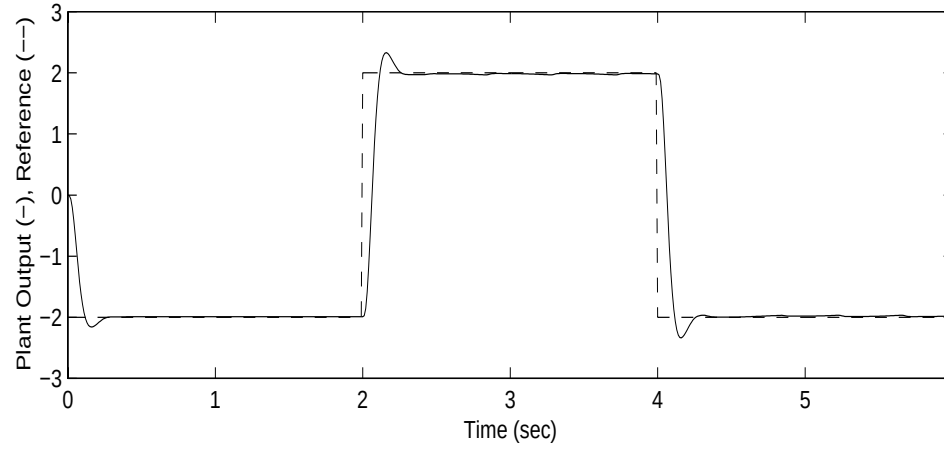
Each block is scaled based on its own maximum and minimum values obtained from the previous simulation using double precision numbers. As shown below, the block icons display the new scaling for each block that had its scaling changed. This scaling is based on the raw simulation values multiplied by `RangeFactor`.

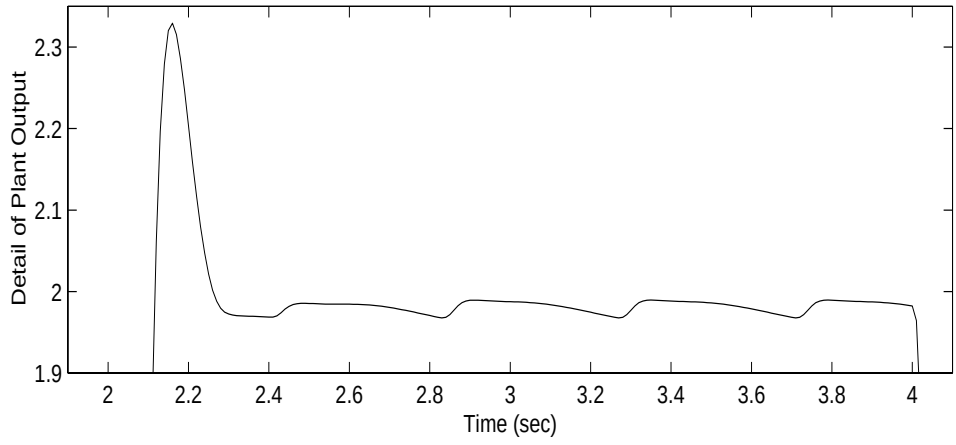


The scaling of the Combined A/D Hardware block did not change because it was locked as shown below.



The simulation results are shown below. It is important to note that a steady-state has been achieved, and a small limit cycle is present in the steady state due to poor A/D design.



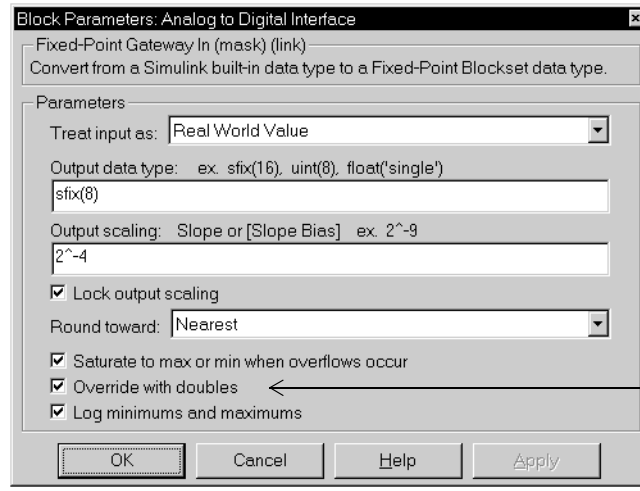




Throughout this simulation, radix point-only scaling is used. However, if variables use slope/bias scaling, then some operations will have to use these values. In the simulation environment (the desktop machine) these values can be represented with IEEE doubles. However, when implemented on a target,

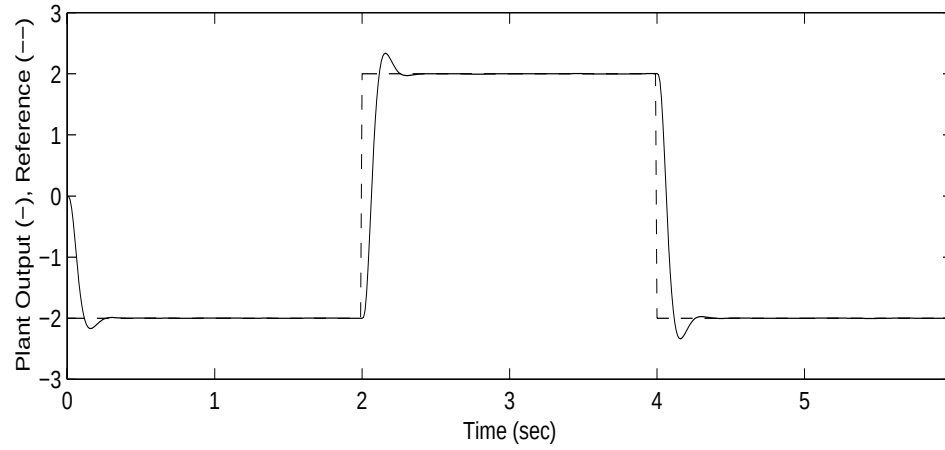
the values should be represented in some native form (e.g., a 16-bit fixed-point number). By forcing the simulation environment to use a specified number of bits, any numerical errors that would appear on the target will be seen in the simulation environment.

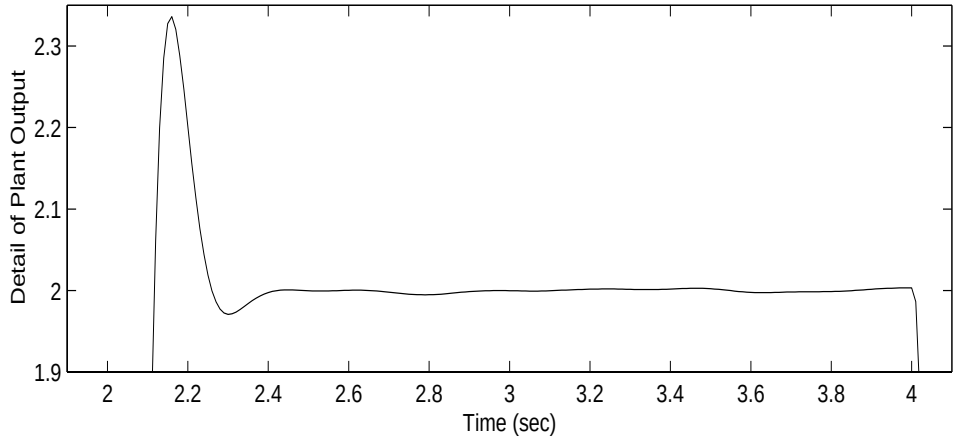
## Simulation 4: Individual Override

The previous simulation optimized results for the fixed-point digital controller. In this simulation, the Analog to Digital Interface block is configured so it feeds the digital controller with doubles. This represents overriding the A/D hardware constraints, and is accomplished by checking the **Override with doubles** checkbox as shown below.



The simulation results are shown below. The limit cycle is no longer present in the steady state—confirmation of a poor A/D design. This means you should replace the appropriate hardware or amplify the signal.



7

# Building Systems and Filters


**Overview** . . . . . 7-2
Realizations and Data Types . . . . . 7-3
Realizations and Scaling . . . . . 7-3

**Targeting an Embedded Processor** . . . . . 7-4
Size Assumptions . . . . . 7-4
Operation Assumptions . . . . . 7-4
Design Rules . . . . . 7-5

**Integrator Realizations** . . . . . 7-7
Trapezoidal Integration . . . . . 7-7
Backward Integration . . . . . 7-9
Forward Integration . . . . . 7-10

**Derivative Realizations** . . . . . 7-12
Filtered Derivative . . . . . 7-12
Unfiltered Derivative . . . . . 7-14

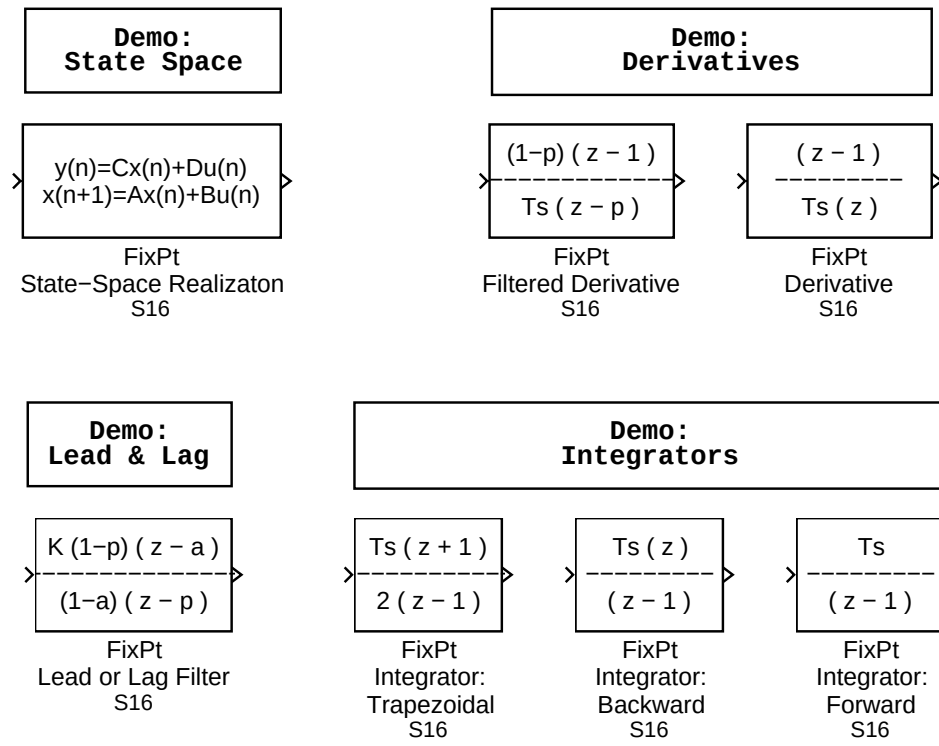
**Lead Filter or Lag Filter Realization** . . . . . 7-18

**State-Space Realization** . . . . . 7-21

# Overview

The Fixed-Point Blockset provides several fixed-point filter and system realizations. These realizations are intended to be used as design templates so you can easily see how to build filters and systems that suit your particular needs. Realizations are provided for state-space, integrator, derivative, and lead or lag systems. For more information about realization structures, refer to Chapter 5, or the texts included in Appendix B.

To display the filters and systems, you can type `fixptsys` at the MATLAB prompt or you can access them through the fixed-point library's Filters & Systems: Examples block. The filters and systems are shown below.



For each filter or system realization, you can:

- Run the demo. Fixed-point results are compared to those obtained from Simulink built-in blocks with identical input.
- Modify the realization. The realizations can be configured or modified to suit your particular design needs.

This chapter presents a few realizations out of many possibilities. These realizations illustrate several important design rules that you need to be aware of when modeling dynamic systems with fixed-point math.

## 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 to each other.

In the more realistic world of double-precision numbers, calculations, and storage of states, small nonlinearities are introduced due to the finite precision and range of floating-point data types. Therefore, each realization of a given system will produce 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 7-4.

## Realizations and Scaling

As with all Fixed-Point Blockset models, you must select a scaling that gives the best precision, range, and performance for your specific fixed-point design.

The scaling for each filter and system demo is based on the default parameters. If these parameters are changed (e.g., the magnitude of the input signal is increased), or if you are creating a new realization, you must define an appropriate scaling. For each system or filter, the scaling can be adjusted manually with the dialog box, or automatically as illustrated in “Simulation Results” on page 6-9.

# Targeting an Embedded Processor

This section describes issues that often arise in targeting a fixed-point design for use on an embedded processor. Rather than describe a specific microprocessor (micro) or digital signal processor (DSP), this section describes some general assumptions about integer sizes and operations available on embedded processors. These assumptions lead to design issues and design rules that may 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 due to 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 multiply 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 need to be aware of when modeling dynamic systems with fixed-point math are given below.

### 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, a multiplication of accumulator-type inputs could be handled by combining several instructions. However, this can lead to large, slow embedded code.

Blocks to convert inputs from the accumulator-type to the base-type can be inserted prior to multiply or gain blocks if needed.

### Design Rule 2: Delays Should Use the Base Data Type

There are two general reasons that a unit delay should use only base-type numbers. First, 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 second reason is that 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 memory that is shared and reused. 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 round-off or overflow. For example, a common filter operation is a weighted sum of several variables. Multiplying a variable by a weight will naturally produce a product of the accumulator type. Before summing, each product could 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 will generally be 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.

# Integrator Realizations

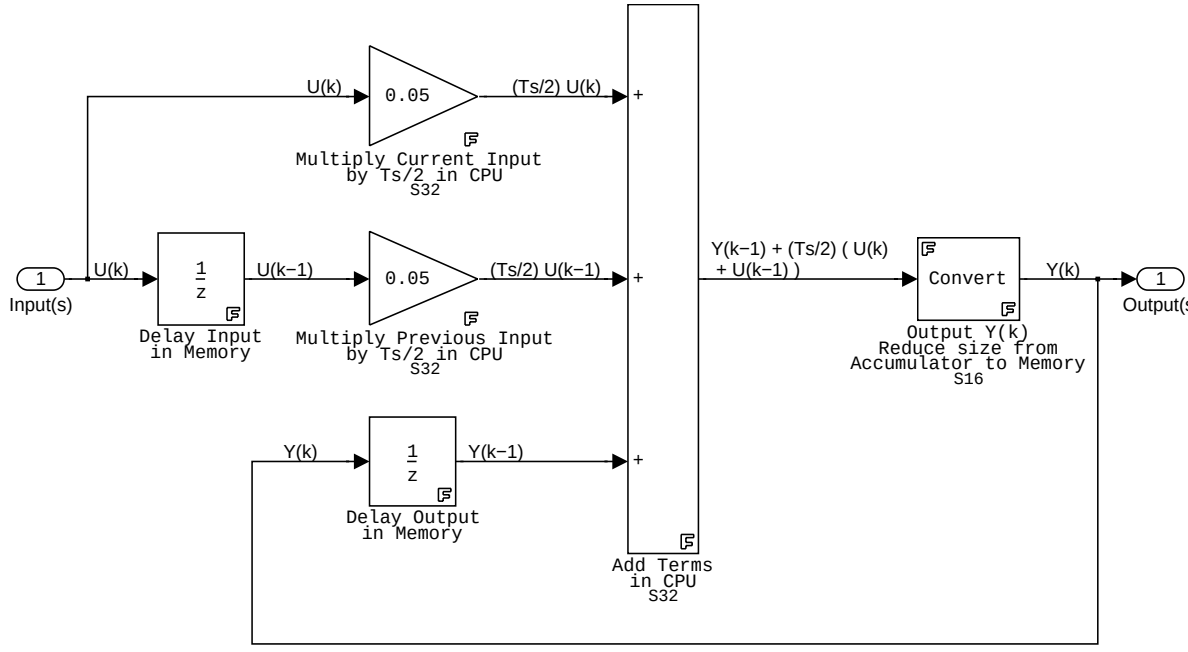
This section presents realizations for trapezoidal, backward, and forward integration. Discussed are the transfer function and difference equation, block parameters, and a review of the model design to reinforce the design rules presented on page 7-5.

## Trapezoidal Integration

The FixPt Integrator: Trapezoidal realization is a masked subsystem that performs discrete-time integration using the Trapezoidal method. For this method, integration is approximated by the *z*-domain transfer function

$$\frac{Ts(z+1)}{2(z-1)}$$

where *Ts* is the sampling period. The realization is shown below.



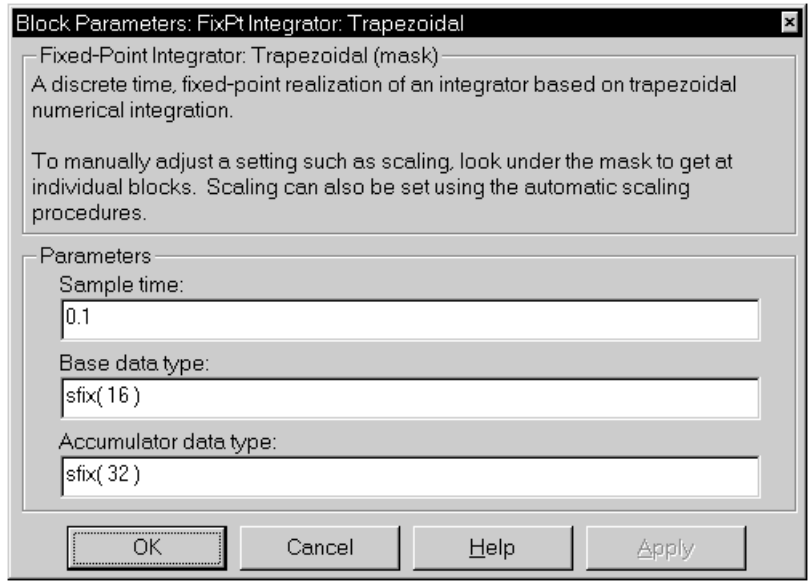
As shown in the figure, the transfer function yields the difference equation

$$y(k) = y(k-1) + \frac{Ts}{2}(u(k) + u(k-1))$$

where $k$ is the time step, $y(k)$ is the current output, $y(k-1)$ is the output from the previous time step, $u(k)$ is the current input, and $u(k-1)$ is the input from the previous time step.

## Parameters and Dialog Box

The dialog box and parameter descriptions for the trapezoidal integrator realization are given below.




**Sample time**

The time interval, *Ts*, between samples

**Base data type**

The processor's base data type

**Accumulator data type**

The processor's accumulator data type

### Model Design Review

A brief review of the model design is given below. The design criteria reflect the rules presented on page 7-5.

- The gains involve multiplications which are a size-growing operation. In most cases, it is desirable for gain and input to use the word size given by the **Base data type** or smaller. The output can be left at the **Accumulator data type** for extra precision in subsequent operations. Alternatively, if the output were stored to RAM, or used by a size-growing operation, it could be reduced to the **Base data type**.
- The FixPt Sum block converts inputs to the output data type before performing the actual addition. Given this order of operation, using the **Accumulator data type** often gives better precision.
- The FixPt Conversion block forces the output to the **Base data type** before storage in RAM (i.e., before input to the unit delay). Casting the output in the feedforward part of the realization prevents subsequent operations from being burdened with a large data type.

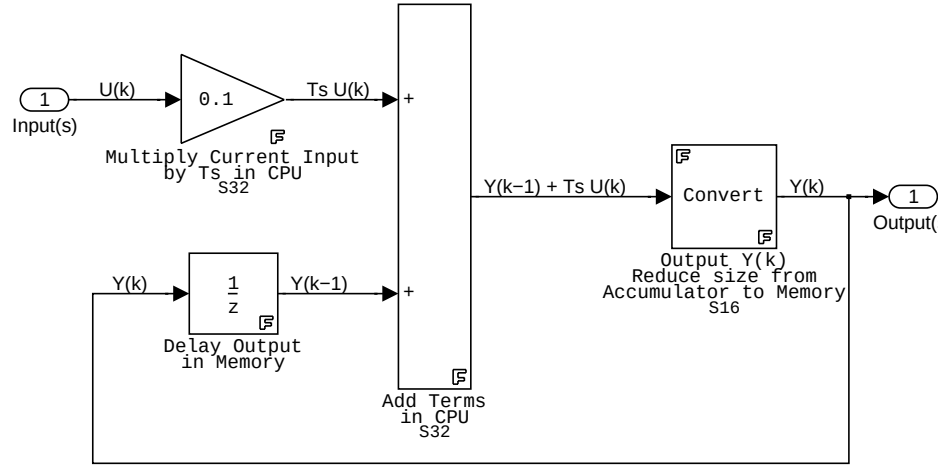
## Backward Integration

The FixPt Integrator: Backward realization is a masked subsystem that performs discrete-time integration using the Backward Euler method. The Backward Euler method is also known as the Backward Rectangular method or right-hand approximation. For this method, integration is approximated by the *z*-domain transfer function

$$\frac{Ts(z)}{(z-1)}$$

where $Ts$ is the sampling period.

The realization is given below.




As shown in the figure, the transfer function yields the difference equation

$$y(k) = y(k-1) + Ts(u(k))$$

where $k$ is the time step, $y(k)$ is the current output, $y(k-1)$ is the output from the previous time step, and $u(k)$ is the current input.

### Parameters and Dialog Box

The parameters and dialog box for the backward integrator realization are the same as those for the trapezoidal integrator realization, and are given on page 7-8.

### Model Design Review

The model design issues are the same as those for the trapezoidal integrator as described on page 7-9.

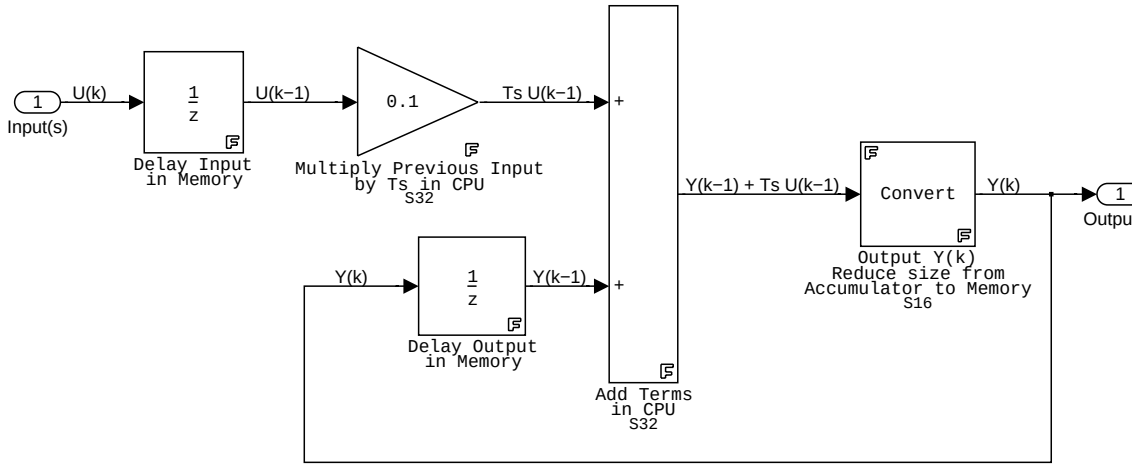
## Forward Integration

The FixPt Integrator: Forward realization is a masked subsystem that performs discrete-time integration using the Forward Euler method. The Forward Euler method is also known as the Forward Rectangular method or

left-hand approximation. For this method, integration is approximated by the *z*-domain transfer function

$$\frac{Ts}{(z-1)}$$

where *Ts* is the sampling period. The realization is given below.




As shown in the figure, the transfer function yields the difference equation

$$y(k) = y(k-1) + Ts(u(k-1))$$

where $k$ is the time step, $y(k)$ is the current output, $y(k-1)$ is the output from the previous time step, and $u(k-1)$ is the input from the previous time step.

## Parameters and Dialog Box

The parameters and dialog box for the forward integrator realization are the same as those for the trapezoidal integrator realization and are given on page 7-8.

## Model Design Review

The model design issues are the same as those for the trapezoidal integrator as described on page 7-9.

# Derivative Realizations

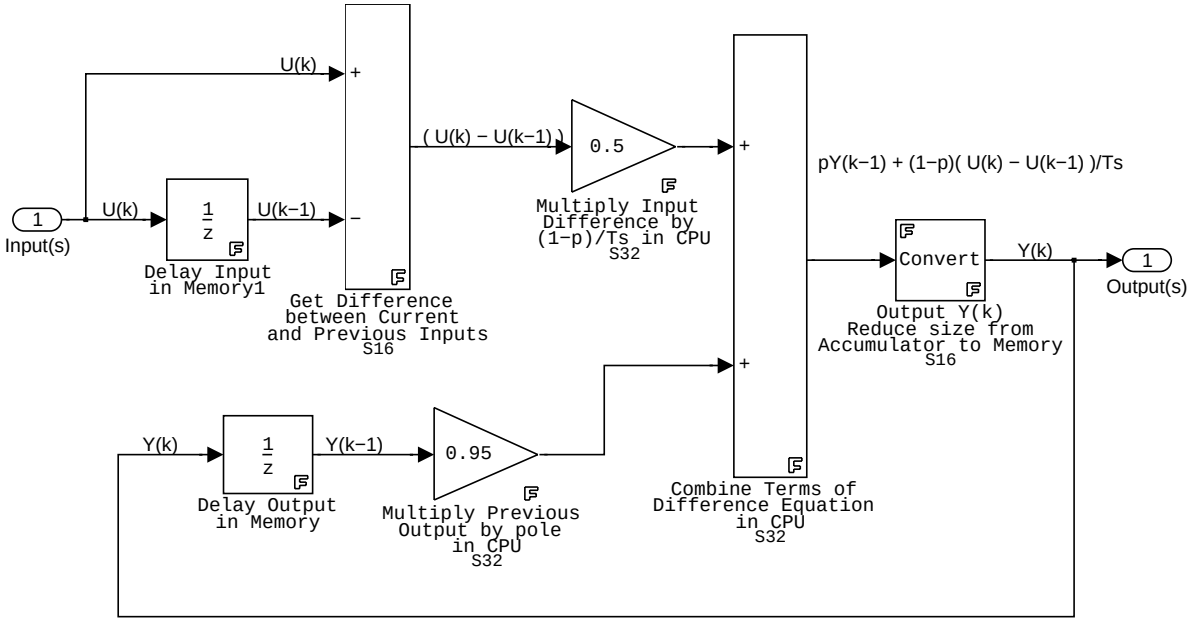
This section presents realizations for filtered and unfiltered derivatives. Discussed are the transfer function and difference equation, block parameters, and a review of the model design to reinforce the design rules presented on page 7-5.

## Filtered Derivative

The FixPt Filtered Derivative realization is a masked subsystem that performs discrete-time filtered differentiation. For this method, differentiation is approximated by the *z*-domain transfer function

$$\frac{(1-p)(z-1)}{Ts(z-p)}$$

where *Ts* is the sampling period and *p* is a pole on the unit circle. The realization is shown below.



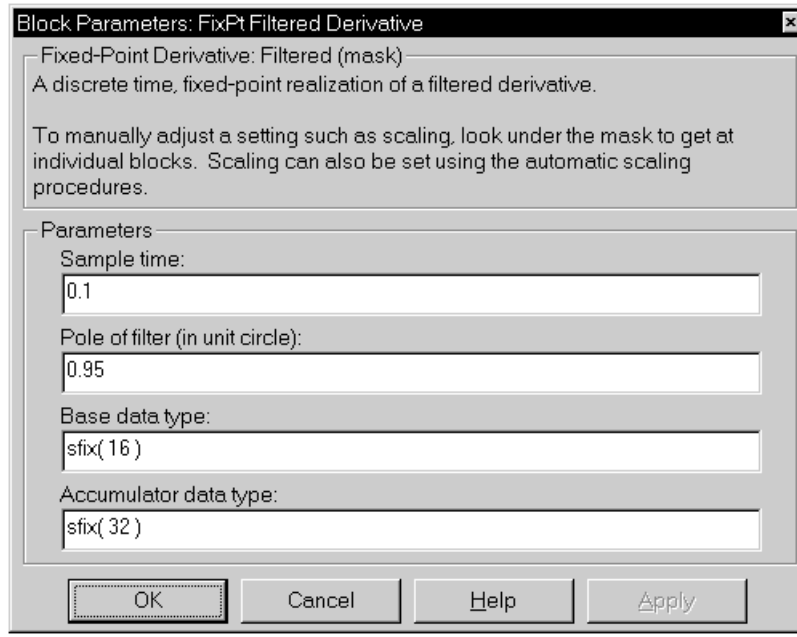
As shown in the figure, the transfer function yields the difference equation

$$y(k) = p(y(k-1)) + \frac{1}{Ts}(1-p)(u(k)-u(k-1))$$

where $k$ is the time step, $y(k)$ is the current output, $y(k-1)$ is the output from the previous time step, $u(k)$ is the current input, and $u(k-1)$ is the input from the previous time step.

## Parameters and Dialog Box

The dialog box and parameter descriptions for the filtered derivative realization are given below.




**Sample time**

The time interval, *Ts*, between samples

**Pole of filter**

The pole, *p*, is defined in the *z* plane so poles inside the unit circle are stable

**Base data type**

The processor's base data type

**Accumulator data type**

The processor's accumulator data type

### Model Design Review

A brief review of the model design is given below. The design criteria reflect the rules presented on page 7-5.

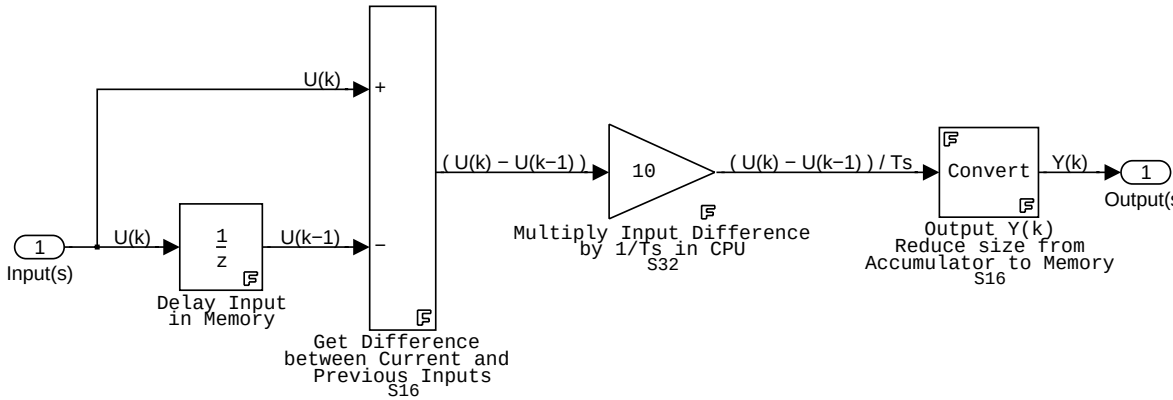
- Using the **Accumulator data type** for the first FixPt Sum block would rarely be advantageous. Both inputs are given by the **Base data Type** with identical scaling so using the same data type for the output makes sense. Also, the subsequent block is a gain, and its input should be the **Base data type** or smaller. The input values to this block should be close so the subtraction can be safely carried out using the **Base data type**.
- The gains involve multiplication which is a size-growing operation. In most cases, it is desirable for gain and input to use the word size given by the **Base data type** or smaller. The output can be left at the **Accumulator data type** for extra precision in subsequent operations. Alternatively, if the output were stored to RAM, or used by a size-growing operation, it could be reduced to the **Base data type**.
- The second FixPt Sum block converts inputs to the output data type before performing the actual addition. Given this order of operation, using **Accumulator data type** often gives better precision.
- The FixPt Conversion block forces the output to the **Base data type** before storage in RAM (i.e., before input to the unit delay). Converting the output in the feedforward part of the realization prevents subsequent operations from being burdened with a large data type.

## Unfiltered Derivative

The FixPt Derivative realization is a masked subsystem that performs discrete-time differentiation. For the this method, differentiation is approximated by the *z*-domain transfer function

$$\frac{(z-1)}{Ts(z)}$$

where *Ts* is the sampling period. The realization is shown below.



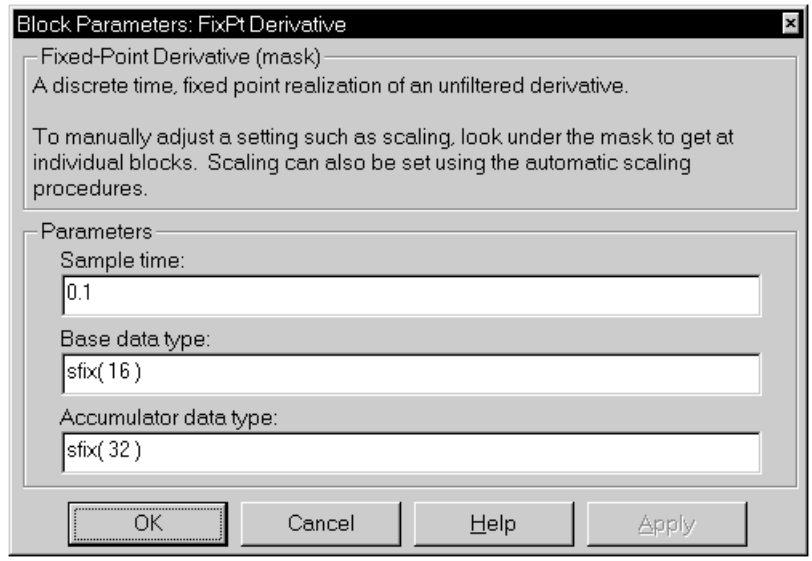

As shown in the figure, the transfer function yields the difference equation

$$y(k) = \frac{1}{Ts}(u(k) - u(k-1))$$

where $k$ is the sample time, $y(k)$ is the current output, $u(k)$ is the current input, and $u(k-1)$ is the input from the previous time step.

## Parameters and Dialog Box

The dialog box and parameter descriptions for the unfiltered derivative realization are given below.




**Sample time**

The time interval, *Ts*, between samples

**Base data type**

The processor's base data type

**Accumulator data type**

The processor's accumulator data type

## Model Design Review

A brief review of the model design is given below. The design criteria reflect the rules presented on page 7-5.

- Using the **Accumulator data type** for the FixPt Sum block would rarely be advantageous. Both inputs are given by the **Base data type** with identical scaling so using the same data type for the output makes sense. Also, the subsequent block is a gain; and its input should be the **Base data type** or

smaller. The input values to this block should be close so the subtraction can be safely carried out using the **Base data type**.

- The gain involves multiplication which is a size-growing operation. In most cases, it is desirable for gain and input to use the word size given by the **Base data type** or smaller. The output can be left at the **Accumulator data type** for extra precision in subsequent operations. Alternatively, if the output were stored to RAM, or used by a size-growing operation, it could be reduced to the **Base data type**.
- The FixPt Conversion casts the output to the **Base data type** before storage in RAM (i.e., before input to the unit delay).

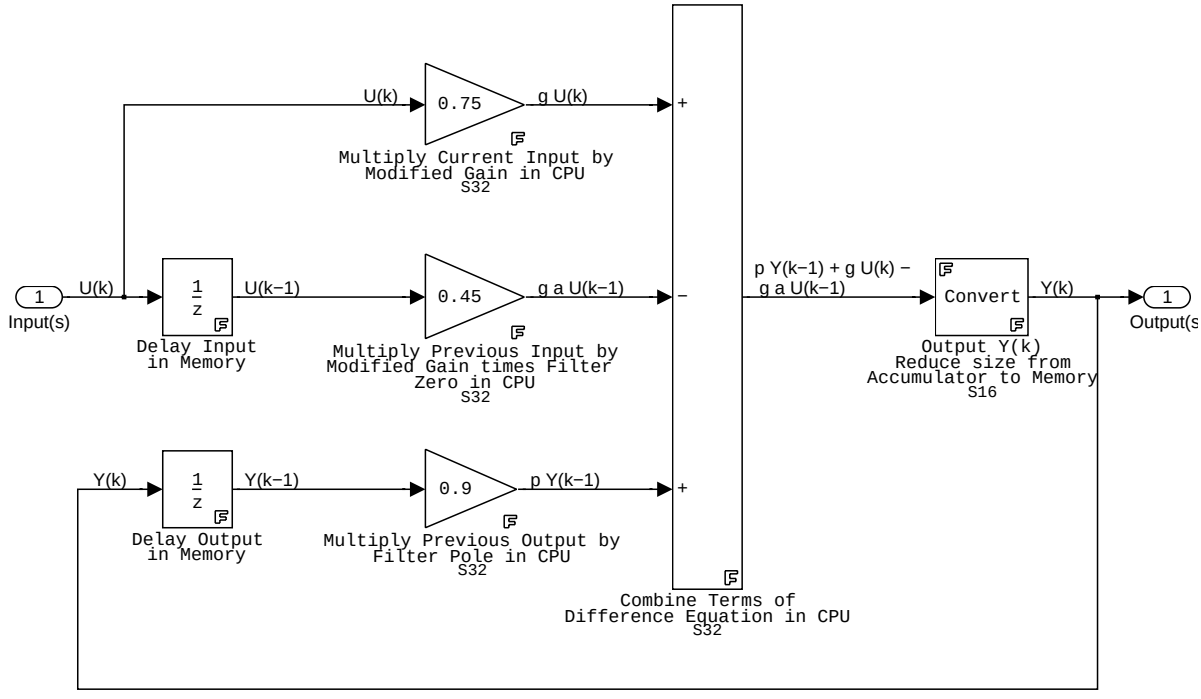
# Lead Filter or Lag Filter Realization

This section presents the realization for a lead filter or lag filter. Discussed are the transfer function and difference equation, block parameters, and a review of the model design to reinforce the design rules presented on page 7-5.

The FixPt Lead or Lag Filter is approximated by the *z*-domain transfer function

$$\frac{K(1-p)(z-a)}{(1-a)(z-p)}$$

where $K$ is the DC gain, $a$ is a zero on the unit circle, and $p$ is a pole on the unit circle. The realization is given below.



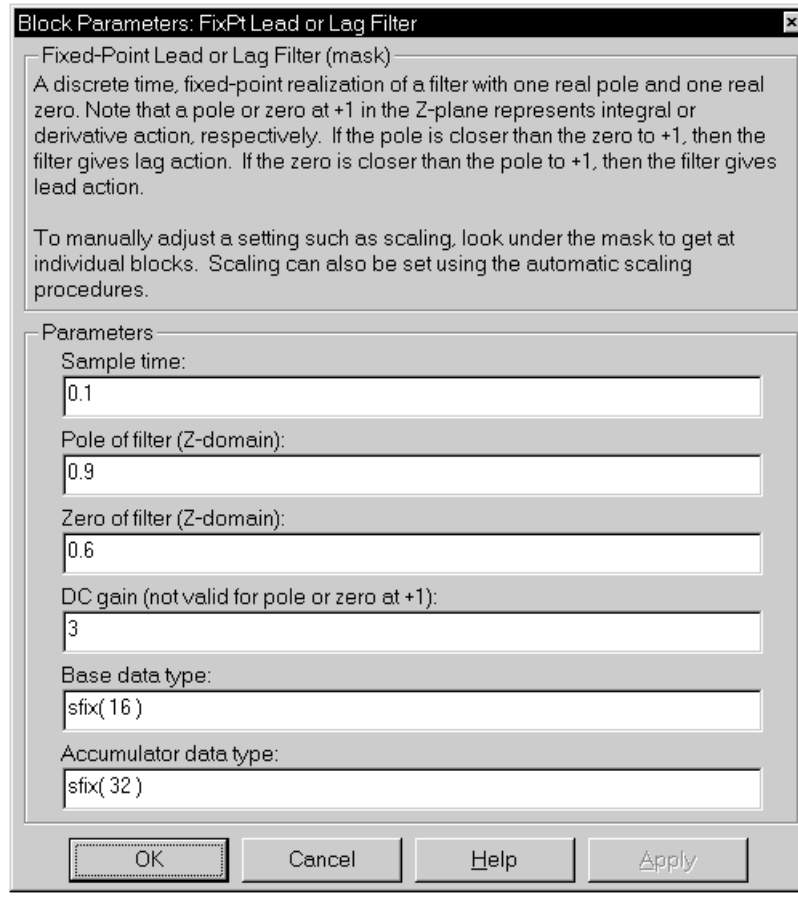
As shown in the figure, the transfer function yields the difference equation

$$y(k) = p(y(k-1)) + g(u(k) - a(u(k-1)))$$

where $k$ is the time step, $g = K(1 - p)$ is the modified gain, $y(k)$ is the current output, $y(k - 1)$ is the output from the previous time step, $u(k)$ is the current input, and $u(k - 1)$ is the input from the previous time step.

## Parameters and Dialog Box

The dialog box and parameter descriptions for the lead or lag filter realization are given below.



**Sample time**

The time interval, *Ts*, between samples

**Pole of filter**

The pole, *p*, defined in the *z*-plane. A pole at +1 represents integral action

**Zero of filter**

The pole, *a*, defined in the *z*-plane. A pole at +1 represents derivative action

**DC gain**

The constant gain, *K*

**Base data type**

The processor's base data type

**Accumulator data type**

The processor's accumulator data type

## Model Design Review

A brief review of the model design is given below. The design criteria reflect the rules presented on page 7-5.

- The gains involve multiplications which are a size-growing operation. In most cases, it is desirable for gain and input to use the word size given by the **Base data type** or smaller. The output can be left at the **Accumulator data type** for extra precision in subsequent operations. Alternatively, if the output were stored to RAM, or used by a size-growing operation, it could be reduced to the **Base data type**.
- The FixPt Sum block converts inputs to the output data type before performing the actual addition. Given this order of operation, using **Accumulator data type** often gives better precision.
- The FixPt Conversion block forces the output to the **Base data type** before storage in RAM (i.e., before input to the unit delay). Converting the output in the feedforward part of the realization prevents subsequent operations from being burdened with a large data type.

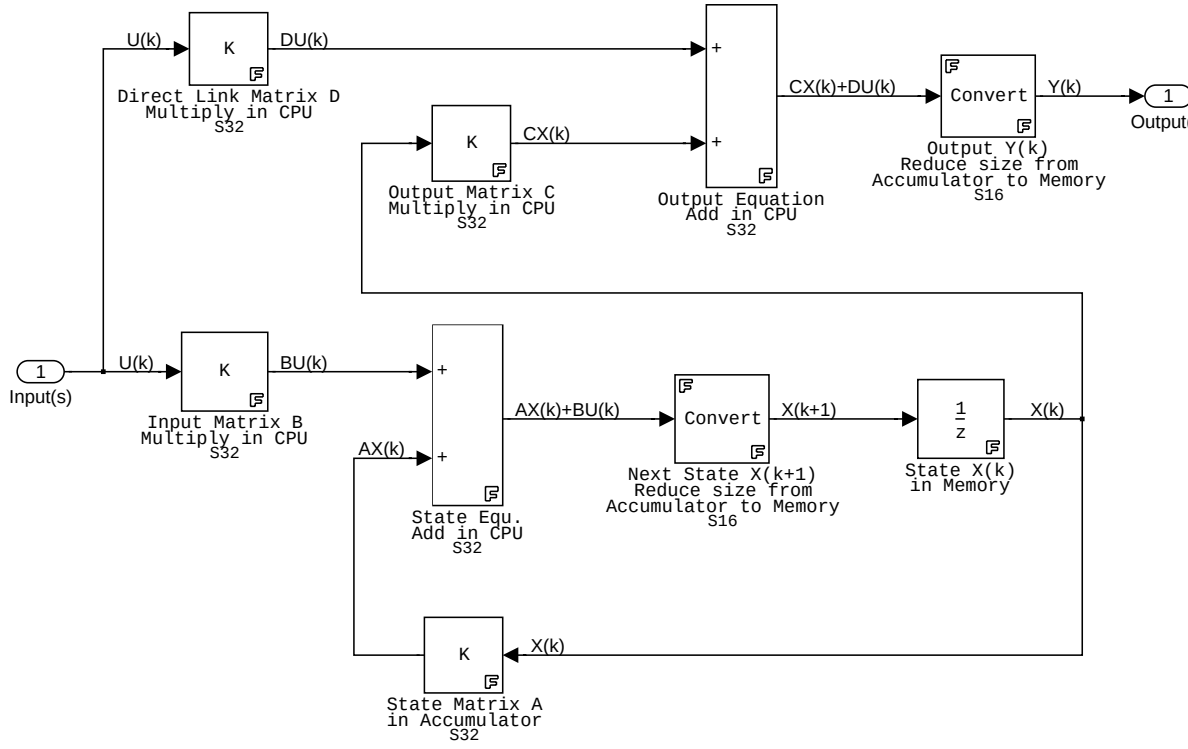
# State-Space Realization

This section presents a fixed-point state-space realization. Discussed are the difference equation, block parameters, and a review of the model design to reinforce the design rules presented on page 7-5.

The FixPt State-Space Realization block is a masked subsystem that implements the system described by

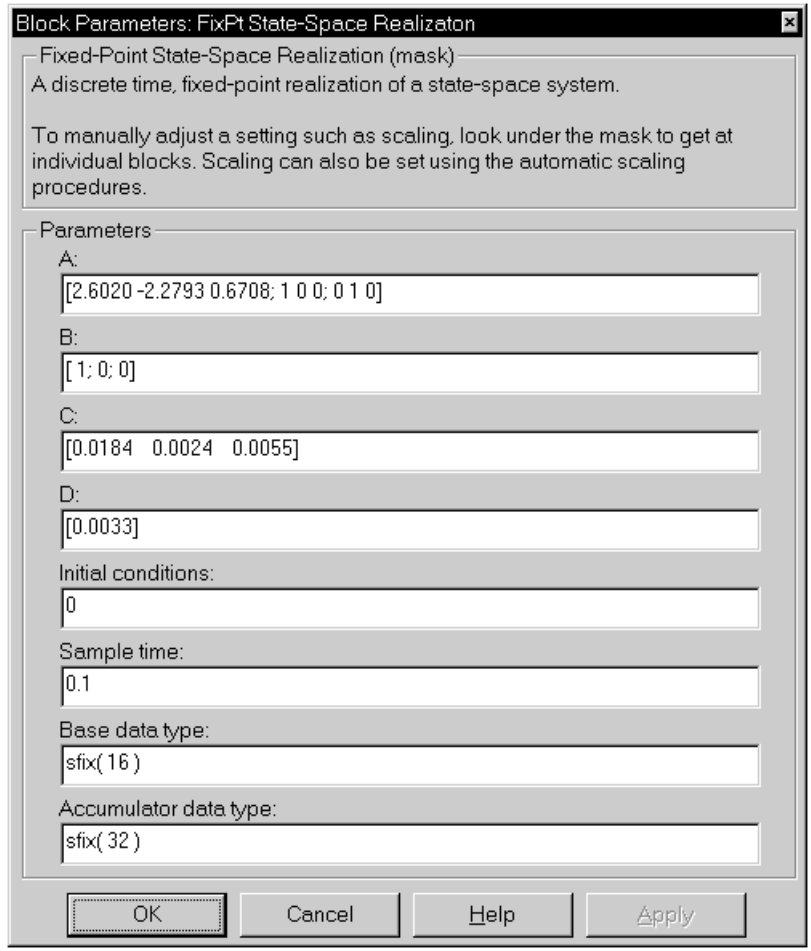
$$x(k+1) = Ax(k) + Bu(k)$$
$$y(k) = Cx(k) + Du(k)$$

where $k$ is the time step, $u(k)$ is the current input, $x(k)$ is the current state, $x(k+1)$ is the state from the next time step, $y(k)$ is the current output, and $A$, $B$, $C$, and $D$ are all coefficient matrices. The realization is shown below.



## Parameters and Dialog Box

The dialog box and parameter descriptions for the state space realization are given below.




**A**

An $n \times n$ matrix where $n$ is the number of states

**B**

An $n \times m$ matrix where $m$ is the number of inputs

**C**

An $r \times n$ matrix where r is the number of outputs

**D**

An $r \times m$ matrix

**Initial conditions**

The initial values for all times preceding the current time

**Sample time**

The time interval, *Ts*, between samples

**Base data type**

The processor's base data type

**Accumulator data type**

The processor's accumulator data type

The advantage of using the state-space realization is that you can build high order systems quickly. The disadvantage is that you can't individually scale the elements on vector signal lines. For example, even if the i-th state, $x_i$, is large and the j-th state, $x_j$, is small, you must use the same scaling for both. Matrix gain coefficients can be individually scaled but this may not suffice.

The solution to this problem is to use a new realization with more blocks and fewer elements on each signal line. For maximum control of scaling, you should use a diagram that has only scalars on each line.

## Model Design Review

A brief review of the model design is given below. The design criteria reflect the rules presented on page 7-5.

- The matrix gains involve a multiplication which is a size-growing operation. In most cases, it is desirable for gain and input to use the word size given by the **Base data type** or smaller. The output can be left at the **Accumulator data type** for extra precision in subsequent operations. Alternatively, if the

output were stored to RAM, or used by a size-growing operation, it could be reduced to the **Base data Type**.

- The FixPt Sum blocks converts inputs to the output data type before performing the actual addition. Given this order of operation, using the **Accumulator data type** often gives better precision.
- The FixPt Conversion blocks force the output to the **Base data type** before storage in RAM (i.e., before input to the unit delay). Converting the output in the feedforward part of the realization prevents subsequent operations from being burdened with a large data type.

8

# Command Reference

**Overview** . . . . . 8-2

# Overview

This chapter contains reference pages for the Fixed-Point Blockset M-file commands. Many of these commands generally do not need to be called from the MATLAB command line. Instead, they are called by specifying certain parameter values via block dialog boxes. These commands are listed below.

| Command | Description |
|---|---|
| `autofixexp` | Automatically change the scaling for each fixed-point block that does not have its scaling locked. |
| `float` | Create a MATLAB structure describing a floating-point data type. |
| `fpupdate` | Update obsolete fixed-point blocks from previous Fixed-Point Blockset releases to current fixed-point blocks. |
| `fxptdlg` | Invoke the automatic scaling graphical user interface (GUI). |
| `sfix` | Create a MATLAB structure describing a signed generalized fixed-point data type. |
| `sfrac` | Create a MATLAB structure describing a signed fractional data type. |
| `showfixptsimranges` | Display the logged maximum and minimum values from the last simulation. |
| `sint` | Create a MATLAB structure describing a signed integer data type. |
| `ufix` | Create a MATLAB structure describing an unsigned generalized fixed-point data type. |
| `ufrac` | Create a MATLAB structure describing an unsigned fractional data type. |
| `uint` | Create a MATLAB structure describing an unsigned integer data type. |

## Purpose

Automatically change the scaling for each fixed-point block that does not have its scaling locked.

## Syntax

```
autofixexp
```

## Description

The `autofixexp` script automatically changes the scaling for each block that does not have its scaling locked. This tool uses the maximum and minimum data obtained from the last simulation run to log data to the workspace. The scaling is changed such that the simulation range is covered and the precision is maximized. The script follows these steps:

1. The global variable `FixPtTempGlobal` is created to "steal" parameters (such as data type) from variables not known in the base workspace. For example, assume the FixPt Sum block has its output data type specified as `DerivedVar`. `DerivedVar` is derived in the mask initialization based on mask parameters and the block is under a mask.

   The value of the parameter `DerivedVar` is retrieved by temporarily replacing `DerivedVar` with `stealparameter(DerivedVar)` in the block dialog. A model update is then forced. When `stealparameter(DerivedVar)` is evaluated, it returns the value of `DerivedVar` without modification and stores the value in `FixPtTempGlobal`. The stolen value is immediately used by this procedure and is not needed again. Therefore, the procedure can move from one block to the next using the same global variable.

2. The `RangeFactor` variable allows you to specify a range differing from that defined by the maximum and minimum values logged in `FixPtSimRanges`. For example, a `RangeFactor` value of 1.55 specifies that a range *at least* 55 percent larger is desired. A value of 0.85 specifies that a range *up to* 15 percent smaller is acceptable.

   You should be aware that the scaling is not exact for the radix point-only case since 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 logged value is 5 and the minimum logged value is -0.5, then any `RangeFactor` from 4/5 to slightly under 8/5 would produce the same radix point since these limits are less than a factor of two from each other. The radix point selected will produce a range from -8 to +8 (minus a bit).

3. The global variable `FixPtSimRanges` is retrieved from the workspace. This is the variable that holds the maximum and minimum simulation values.
4. The workspace is searched for the variables `SlopeBits` and `BiasBits`, which specify the number of bits to use in representing slopes and biases. If these variables are not found, then they are automatically created with default values of 7 and 8, respectively.
5. All blocks that logged maximum and minimum simulation data are processed.
6. All blocks that do not have their scaling locked are automatically scaled. If the data type class is `FIX`, then radix point-only scaling is performed. If the data type class is `INT`, then slope/bias scaling is performed. To find out a data type's class, refer to its reference page in this chapter.

## Example

To see how `autofixexp` is used, refer to "Simulink Model of a Feedback Design" on page 6-3.

## See Also

`fxptdlg`, `showfixptsimranges`

## Purpose

Create a MATLAB structure describing a floating-point data type.

## Syntax

```
a = float('single')
a = float('double')
a = float(TotalBits, ExpBits)
```

## Description

`float('single')` returns a MATLAB structure that describes the data type of an IEEE single (32 total bits, 8 exponent bits).

`float('double')` returns a MATLAB structure that describes the data type of an IEEE double (64 total bits, 11 exponent bits).

`float(TotalBits, ExpBits)` returns a MATLAB structure that describes a nonstandard floating-point data type that mimics the IEEE style. That is, the 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's and NaN's.

`float` is automatically called when a floating point number is specified in a block dialog box.

---

**Note:** Unlike fixed-point numbers, floating point numbers are not subject to any specified scaling.

---

## Example

Define a nonstandard, IEEE-style, floating-point data type with 31 total bits (including the hidden leading one) and 9 exponent bits.

```
a = float(31,9)
a =
        Class: 'FLOAT'
    MantBits: 21
     ExpBits: 9
```

## See Also

`sfix`, `sfrac`, `sint`, `ufix`, `ufrac`, `uint`

# fpupdate

## Purpose

Update obsolete fixed-point blocks from previous Fixed-Point Blockset releases to current fixed-point blocks.

## Syntax

```
fpupdate('model')
fpupdate('model', blkprompt)
fpupdate('model', blkprompt, varprompt)
fpupdate('model', blkprompt, varprompt, muxprompt)
fpupdate('model', blkprompt, varprompt, muxprompt, message)
```

## Description

`fpupdate('model')` replaces all obsolete fixed-point blocks contained in `model` with current fixed-point blocks. The model must be opened prior to calling `fpupdate`.

`fpupdate('model', blkprompt)` prompts you for replacement of obsolete blocks. If `prompt` is 0 (the default), you will not be prompted. If `prompt` is 1, you will have these three options:

- `y` (default) replaces the block.
- `n` does not replace the block.
- `a` replaces all blocks without further prompting.

`fpupdate('model', blkprompt, varprompt)` gives you the option of updating variables which appear in each block's dialog box with their actual numerical values. Note that such an update is possible only if the variables can be evaluated in the MATLAB workspace. If `varprompt` is 1 (the default), you are prompted for each variable found in the block diagram. If `varprompt` is 0, all variables are automatically updated without prompting.

`fpupdate('model', blkprompt, varprompt, muxprompt)` allows you to update the input size parameters of the Mux and Demux blocks found in `model`. The input sizes of these blocks may need to be updated to account for the mismatch between the old and new fixed-point data representations. In the old representation, each number had a width of 2. In the new representation, each number has a width of 1. To update Mux and Demux blocks that have only fixed-point inputs, the vector that specifies the input size should be divided by 2. If `muxprompt` is 1 (the default), each Mux and Demux block found in `model` is updated. If `muxprompt` is 0, the Mux and Demux blocks are automatically updated without prompting.

`fpupdate('model', blkprompt, varprompt, muxprompt, message)` allows you to show or suppress any warning or update messages generated during the update process. If `message` is 1 (the default), all messages are displayed. If `message` is 0, all messages are suppressed.

`fpupdate` calls `addterms` to terminate any unconnected input or output ports by attaching Ground or Terminator blocks, respectively.

## Example

To see how `fpupdate` works, convert the obsolete model `fixpoint/obsolete/fpex1.mdl`.

```
fpex1
fpupdate('fpex1')
```

## Purpose

Invoke the fixed-point graphical user interface (GUI).

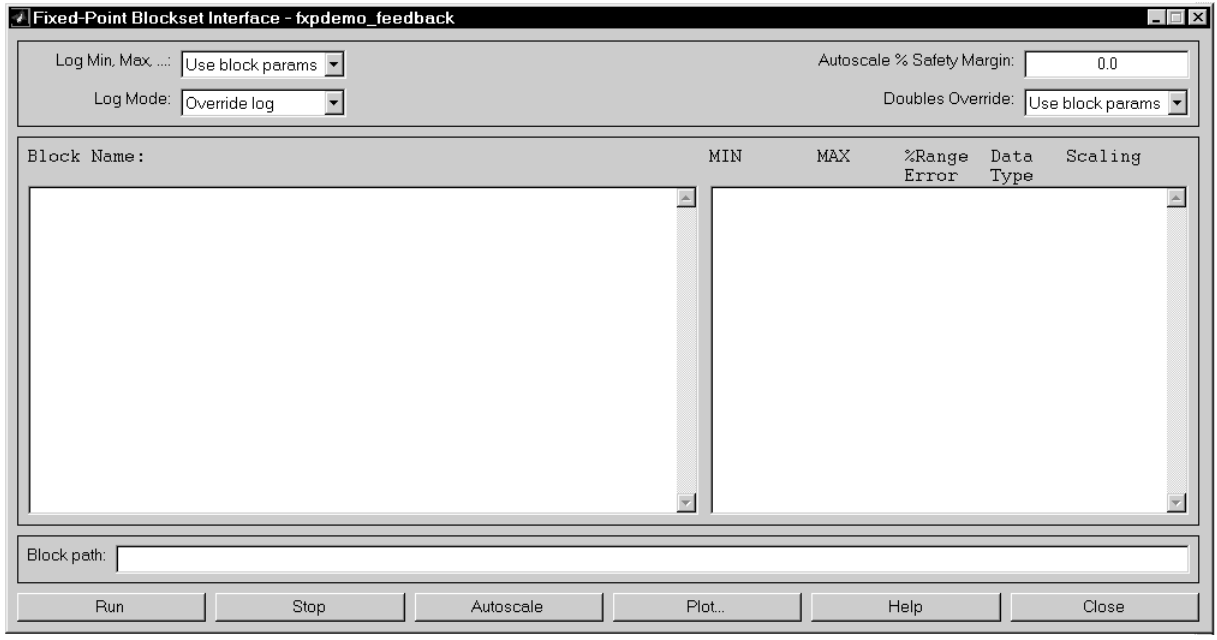
## Syntax

```
fxptdlg('model')
```

## Description

`fxptdlg('model')` opens the fixed-point graphical user interface (GUI) for the fixed-point MDL-file `model`. The interface provides convenient access to the global overrides and min/max logging settings, the logged min/max data, the automatic scaling tool, and the signal comparison tool. A GUI can be opened for each unique MDL-file, and controls only that model. The GUI can also be invoked through the fixed-point library's Fixed-Point GUI block.

For each block in the model that logs data, the GUI displays the block descriptive name, the minimum simulation value, the maximum simulation value, the maximum absolute error as a percentage of range, the data type, and the scaling. You can display a block's dialog box by double-clicking on the appropriate GUI entry. The fixed-point GUI is shown below



The maximum absolute error as a percentage of range is a useful diagnostic parameter particularly for large models. A value of 100% or more typically indicates an overflow condition for that block although some other significant error such as saturation is also possible.

The `Log Min/Max` pull-down menu controls which blocks log data. `All` logs min/max data for all blocks, `None` doesn't log any min/max data, and `Use block params` logs min/max data for all blocks that have the **Log minimums and maximums** checkbox checked.

The `Log Mode` pull-down menu controls how the log file is updated when multiple simulations are run. `Override log` updates all logged values for each simulation run. `Merge log` keeps the highest and lowest logged values across multiple simulations.

Overriding the output data type with doubles is controlled with the `Doubles Override` pull-down menu. `All` overrides the output data type for all blocks, `None` doesn't override the output data type for any block, and `Use block params` overrides the output data type for blocks that have the **Override with doubles** checkbox checked.

`Autoscale % Safety Margin` sets the value for `RangeFactor` as a percent safety margin. For example, a `RangeFactor` of 1.20 is specified as 20. Before automatic scaling is performed, the simulation must be run to collect min/max data. For more information about `RangeFactor`, refer to page 8-3.

The fixed-point GUI contains six buttons: **Run**, **Stop**, **Autoscale**, **Plot**, **Help**, and **Close**. The **Run** button runs the model and updates the GUI display with the latest min/max values. The **Stop** button stops the simulation from running. The **Autoscale** button invokes the automatic scaling tool. The **Plot** button invokes the signal comparison tool, which displays any To Workspace, Outport, or Scope blocks found in the model. The **Help** button displays the HTML-based help. The **Close** button closes the fixed-point GUI.

The raw signal data is plotted with the **Plot**>**Plot Signals** button. Raw signal data is generated when the global override switch is off. Doubles are plotted with the **Plot**>**Plot Doubles** button. Doubles are generated when the global override switch is on. Both raw signal data and doubles can be plotted with the **Plot**>**Plot Both** button. Note that the doubles override does not overwrite the raw data.

`Block path` displays the path for each selected block.

## Example

The Fixed-Point GUI block is included with all demos. To learn how to use the GUI, select the Automatic Scaling in Feedback Control demo, and follow the steps outlined in “Simulation Results” in Chapter 6.

## See Also

`autofixexp`, `showfixptsimranges`

## Purpose

Create a MATLAB structure describing a signed generalized fixed-point data type.

## Syntax

```
a = sfix(TotalBits)
```

## Description

`sfix(TotalBits)` returns a MATLAB structure that describes the data type of a signed generalized fixed-point number with a word size given by `TotalBits`.

`sfix` is automatically called when a signed generalized fixed-point data type is specified in a block dialog box.

---

**Note:** A default radix point is not included in this data type description. Instead, the scaling must be explicitly defined in the block dialog box.

---

## Example

Define a 16-bit signed generalized fixed-point data type.

```
a = sfix(16)
a =
       Class: 'FIX'
    IsSigned: 1
    MantBits: 16
```

## See Also

`float`, `sfrac`, `sint`, `ufix`, `ufrac`, `uint`

# sfrac

## Purpose

Create a MATLAB structure describing a signed fractional data type.

## Syntax

```
a = sfrac(TotalBits)
a = sfrac(TotalBits, GuardBits)
```

## Description

`sfrac(TotalBits)` returns a MATLAB structure that describes the data type of a signed fractional number with a word size given by `TotalBits`.

`sfrac(TotalBits, GuardBits)` returns a MATLAB structure that describes the data type of a signed fractional number. The total word size is given by `TotalBits` with `GuardBits` bits located to the left of the sign bit.

`sfrac` is automatically called when a signed fractional data type is specified in a block dialog box.

The default radix point for this data type is assumed to lie immediately to the right of the sign bit. If guard bits are specified, they lie to the left of the radix point in addition to the sign bit.

## Example

Define an 8-bit signed fractional data type with 4 guard bits. Note that the range of this number is $-2^4 = -16$ to $(1 - 2^{(1-8)}) \cdot 2^4 = 15.875$.

```
a = sfrac(8,4)
a =
        Class: 'FRAC'
     IsSigned: 1
     MantBits: 8
    GuardBits: 4
```

## See Also

`float`, `sfix`, `sint`, `ufix`, `ufrac`, `uint`

## Purpose

Display the logged maximum and minimum values from the last fixed-point simulation.

## Description

The `showfixptsimranges` script displays the logged maximum and minimum values from the last fixed-point simulation. Data is logged only from blocks where the **Log minimums and maximums** checkbox is checked.

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

## Example

To see how `showfixptsimranges` is used, refer to "Simulink Model of a Feedback Design" in Chapter 6.

## See Also

`autofixexp`, `fxptdlg`

# sint

## Purpose

Create a MATLAB structure describing a signed integer data type.

## Syntax

```
a = sint(TotalBits)
```

## Description

`sint(TotalBits)` returns a MATLAB structure that describes the data type of a signed integer with a word size given by `TotalBits`.

`sint` is automatically called when a signed integer is specified in a block dialog box.

The default radix point for this data type is assumed to lie to the right of all bits.

## Example

Define a 16-bit signed integer data type.

```
a = sint(16)
a =
       Class: 'INT'
    IsSigned: 1
    MantBits: 16
```

## See Also

`float`, `sfix`, `sfrac`, `ufix`, `ufrac`, `uint`

## Purpose

Create a MATLAB structure describing an unsigned generalized fixed-point data type.

## Syntax

```
a = ufix(TotalBits)
```

## Description

`ufix(TotalBits)` returns a MATLAB structure that describes the data type of an unsigned generalized fixed-point data type with a word size given by `TotalBits`.

`ufix` is automatically called when an unsigned generalized fixed-point data type is specified in a block dialog box.

---

**Note:** The default radix point is not included in this data type description. Instead, the scaling must be explicitly defined in the block dialog box.

---

## Example

Define a 16-bit unsigned generalized fixed-point data type.

```
a = ufix(16)
a =
       Class: 'FIX'
    IsSigned: 0
    MantBits: 16
```

## See Also

`float`, `sfix`, `sfrac`, `sint`, `ufrac`, `uint`

# ufrac

## Purpose

Create a MATLAB structure describing an unsigned fractional data type.

## Syntax

```
a = ufrac(TotalBits)
a = ufrac(TotalBits, GuardBits)
```

## Description

`ufrac(TotalBits)` returns a MATLAB structure that describes the data type of an unsigned fractional number with a word size given by `TotalBits`.

`ufrac(TotalBits, GuardBits)` returns a MATLAB structure that describes the data type of an unsigned fractional number. The total word size is given by `TotalBits` with `GuardBits` bits located to the left of the radix point.

`ufrac` is automatically called when an unsigned fractional data type is specified in a block dialog box.

The default radix point for this data type is assumed to lie immediately to the left of all bits. If guard bits are specified, then they lie to the left the default radix point.

## Example

Define an 8-bit unsigned fractional data type with 4 guard bits. Note that the range of this number is from 0 to $(1 - 2^{-8}) \cdot 2^{4} = 15.9375$.

```
a = ufrac(8,4)
a =
         Class: 'FRAC'
     IsSigned: 0
     MantBits: 8
    GuardBits: 4
```

## See Also

`float`, `sfix`, `sfrac`, `sint`, `ufix`, `uint`

## Purpose

Create a MATLAB structure describing an unsigned integer data type.

## Syntax

```
a = uint(TotalBits)
```

## Description

`uint(TotalBits)` returns a MATLAB structure that describes the data type of an unsigned integer with a word size given by `TotalBits`.

`uint` is automatically called when an unsigned integer is specified in a block dialog box.

The default radix point for this data type is assumed to lie to the right of all bits.

## Example

Define a 16-bit unsigned integer.

```
a = uint(16)
a =
       Class: 'INT'
    IsSigned: 0
    MantBits: 16
```

## See Also

`float`, `sfix`, `sfrac`, `sint`, `ufix`, `ufrac`

9

# Block Reference

**The Block Reference Page** . . . . . 9-2

**The Fixed-Point Blockset Library** . . . . . 9-10

# The Block Reference Page

Fixed-Point Blockset blocks appear in alphabetical order and contain some or all of this information:

- The block name and icon
- The purpose of the block
- A description of the block including:
  - A detailed description of the parameters
  - A description of the block icon
- The block parameters and dialog box including a brief description of each parameter
- The rules for some or all of these topics, as they apply to the block:
  - Converting block parameters from double precision numbers to Fixed-Point Blockset data types
  - Converting the input data type(s) to the output data type
  - Performing block operations between inputs and parameters
- The block characteristics, including some or all of these, as they apply to the block:
  - Input Port(s) – the data type(s) accepted by the block and whether the inputs can be a scalar or vector
  - Output Port – the data type(s) produced by the block and whether the outputs can be a scalar or vector
  - Direct Feedthrough – whether the block or any of its ports has direct feedthrough
  - Sample Time – how the block's sample time is determined, whether by the block itself or inherited from the block that drives it or is driven by it
  - Scalar Expansion – whether or not scalars are expanded to vectors
  - States – the number of discrete states
  - Vectorized – whether or not the block accepts and/or generates vector signals

In some cases, an example using the block (or a specific feature of the block) is included in the reference page description as well.

# The Block Dialog Box

Fixed-Point Blockset blocks are configured with a parameter dialog box. The parameter dialog box provides you with:

- The name and type of the block at the top of the dialog box
- A brief description of the block's behavior below the title
- Zero or more editable parameter fields, checkboxes, or pull-down menus below the description. The parameter values should be a MATLAB expression.
- A row of four buttons labeled **OK**, **Cancel**, **Help,** and **Apply** at the bottom of the dialog box. The **OK** button sets the current parameter values and closes the dialog box. The **Cancel** button reverts all the parameter values back to their values at the time the window was opened, losing any changes made since the window was opened. The **Help** button displays the on-line HTML-based reference information. The **Apply** button sets the current parameter values and but does not close the dialog box.

Simulink stores the strings entered in these fields and passes them to MATLAB for evaluation when a simulation is started. If MATLAB variables are used, the simulation uses the values that exist in the workspace at the start of the simulation. These variables are not necessarily the same as when the variables are entered into the dialog box fields. If a simulation is running when a parameter is changed, MATLAB evaluates the parameter as soon as the **OK** or **Apply** button is pressed.

# Common Block Features

Many Fixed-Point Blockset blocks display the same parameters through their dialog boxes and display similar information through their block icon. For convenience, all the common block features are described in this section.

## Block Parameters

The common block parameters are associated with these blockset features:

- Parameter and output data type selection
- Scaling
- Rounding
- Overflow handling
- Overriding the output data type with doubles
- Logging simulation results

Each of these features is described below.

### Data Type Selection

The parameter and output data type and word size are specified with the **Parameter data type** and **Output data type** parameters, respectively. The supported output data types are shown below.

**Table 9-1: Output Data Types and Default Scaling**

| Data Type | Description | Default Scaling |
|---|---|---|
| `float` | Floating-point number | None |
| `ufix` | Unsigned generalized fixed-point number | None |
| `sfix` | Signed generalized fixed-point number | None |
| `uint` | Unsigned integer | Right of the least significant bit |
| `sint` | Signed integer | Right of the least significant bit |

**Table 9-1: Output Data Types and Default Scaling**

| Data Type | Description | Default Scaling |
|---|---|---|
| `ufrac` | Unsigned fractional number | Left of the most significant bit |
| `sfrac` | Signed fractional number | Right of the sign bit |

The word size in bits of fixed-point data types is given as an argument to the data type. For example, `sfix(16)` specifies a 16-bit signed generalized fixed-point number. Word sizes from 1 to 128 bits are supported.

Floating-point data types are IEEE-style and are specified as `float('single')` for single precision numbers and `float('double')` for double precision numbers. Nonstandard IEEE-style numbers are specified as `float(x,y)` where `x` is the total number of physical bits and `y` is the number of exponent bits.

---

**Note:** A default radix point is not included with the generalized fixed-point data type. Instead, the scaling must be explicitly specified as described below.

---

For more information about supported data types and their default scaling, refer to Chapter 3, “Data Types and Scaling.”

## Scaling

If the parameter or output data type is specified as generalized fixed-point, then you must select a scaling mode with the **Parameter scaling** and **Output scaling** parameters, respectively. The scaling modes are:

- **Radix Point-Only Scaling**

  This mode specifies radix point-only scaling for the output. For example, a scaling given by `2^–10` (or `pow2(–10)`) places the radix point at a location 10 places to the left of the least significant bit.

- **Slope/Bias Scaling**

  This mode specifies slope/bias scaling for the output. For example, a scaling given by `[5/9 10]` specifies a slope of 5/9 and a bias of 10. When using this mode, you must specify a positive slope.

Default scaling is used for all other fixed-point data types.

### Lock Output Scaling

If the **Lock output scaling** checkbox is checked, then the automatic scaling tool will not change the **Output scaling** parameter value. Otherwise, the automatic scaling tool is free to adjust the scaling.

### Rounding

You can choose the rounding mode for the block operation with the **Round toward** pull-down menu. The available rounding modes are shown below.

**Table 9-2: Rounding Modes**

| Rounding Mode | Description |
|---|---|
| `zero` | Round the result towards zero |
| `nearest` | Round the result towards the nearest representable number (the exact midpoint is rounded towards positive infinity) |
| `ceiling` | Round the result towards positive infinity |
| `floor` | Round the result towards negative infinity |

### Overflow Handling

Overflow handling for fixed-point numbers is specified with the **Saturate to max or min when overflows occur** checkbox. If checked, fixed-point overflow results saturate. Otherwise, overflow results wrap. Whenever a result saturates, a warning is displayed.

### Overriding with Doubles

If the **Override with doubles** checkbox is checked, then the **Parameter data type** and **Output data type** selection is ignored. Instead, parameters and outputs are represented using double precision floating-point numbers. Also, any calculations will be performed using floating-point arithmetic.

An exception to above rule is when parameters or output contain a bias. In this case, the bias is not ignored in subsequent fixed-point operations.

If the parameter and output data types are both floating-point, the **Override with doubles** checkbox is not available.

### Logging Simulation Results

The minimum and maximum values produced by the simulation are logged if the **Log minimums and maximums** checkbox is checked. The logged values are stored in the `FixPtSimRanges` cell array in the MATLAB workspace. You can access these values with the `showfixptsimranges` script or the fixed-point GUI.

In addition to logging the minimum and maximum simulation values, the maximum absolute error for each block is also logged. If the maximum absolute error exceeds 50% of the output range, then a warning, an error, or nothing is issued depending on the setting of the **Integer Overflow** mode under **Simulation**>**Parameters**>**Diagnostics**. The maximum absolute error represents error due to online calculations within a block. The maximum absolute error excludes effects due to quantization of the parameters and inputs.

The automatic scaling tool will not adjust blocks that have not logged min/max information.

## Block Icon Display

Many blockset icons look like those of built-in Simulink blocks. For this reason, all block icons have an “F” (for “Fixed-Point”) associated with them. An “F” in the lower right (upper left) corner of the icon means the block output (input) is a Fixed-Point Blockset data type. Some blocks have additional labels, which are described in their reference pages.

For blocks that allow you to choose the output data type, you can control the information displayed below the icon with the global variable `FixDispPref`. The icon display choices are explained below.

**Table 9-3: Icon Display Information**

| FixDispPref Value | Explanation |
|---|---|
| 0 | Display no information |
| 1 | Display data type, radix, fractional slope, and bias information |

**Table 9-3: Icon Display Information**

| FixDispPref Value | Explanation |
|---|---|
| 2 | Display data type and slope/bias information |
| 3 | Display data type, and maximum and minimum scaled values |

When a scaling property takes a trivial value, that information is not displayed. This occurs when:

- The bias or fixed power of two exponent = 0
- The slope or fractional slope = 1

## Fixed-Point Output

If a block is configured for fixed-point output and `FixDispPref` = 1 or 2, then the block icon displays the data type format. Scaling information is displayed for all fixed-point data types. Also, if the output has a non-zero bias and the **Override with doubles** checkbox is checked, then the bias is still displayed.

For example, suppose the output data type is a 16-bit signed generalized fixed point number with slope/bias scaling specified as `[2.5 10]`. If `FixDispPref` = 1, then the data type information is displayed as

```
S16 2^1
Fslope 1.25
Bias 10
```

If `FixDispPref` = 2, then the data type information is displayed as

```
S16
Slope 2.5
Bias 10
```

If `FixDispPref` = 3, then the data type information is displayed as

```
S16
Min -81910
Max 81927.5
```

### Floating-Point Output

If the block is configured for standard floating-point output, and `FixDispPref` = 1, 2, or 3, the block icon displays either `single` or `double`. Nonstandard floating-point output is indicated by the number of mantissa and exponent bits. For example, `float(16,8)` is displayed as `7 MantBits` and `8 ExpBits`.

If the block is configured for fixed-point output, has a non-zero bias, and the **Override with doubles** checkbox is checked, then the bias will still be displayed.

# The Fixed-Point Blockset Library

The Fixed-Point Blockset library contains the blocks described below.

**Table 9-4: Fixed-Point Blocks**

| Block Name | Purpose |
|---|---|
| FixPt Constant | Generate a constant value |
| FixPt Conversion | Convert from one Fixed-Point Blockset data type to another |
| FixPt Conversion Inherited | Convert second input to the data type of the first input |
| FixPt FIR | Implement a fixed-point finite impulse response (FIR) filter |
| FixPt Gain | Multiply the input by a constant |
| FixPt Gateway In | Convert a Simulink data type to a Fixed-Point Blockset data type |
| FixPt Gateway Out | Convert a Fixed-Point Blockset data type to a Simulink data type. |
| FixPt Logical Operator | Perform the specified logical operation on the inputs |
| FixPt Look-Up Table | Approximate a one-dimensional function using a selected look-up method |
| FixPt Look-Up Table (2-D) | Approximate a two-dimensional function using a selected look-up method |
| FixPt Matrix Gain | Multiply the input by a constant matrix |
| FixPt Product | Multiply or divide inputs |
| FixPt Relational Operator | Perform the specified relational operation on the inputs |
| FixPt Relay | Switch output between two constants |
| FixPt Saturation | Bound the range of the input |
| FixPt Sum | Add or subtract inputs |
| FixPt Switch | Switch output between input one or input three based upon the value of input two |

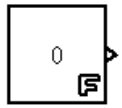
**Table 9-4: Fixed-Point Blocks (Continued)**

| Block Name | Purpose |
|---|---|
| FixPt Unit Delay | Delay a signal one sample period |
| FixPt Zero-Order Hold | Implement a zero-order hold of one sample period |

## Purpose

Generate a constant value.

## Description




The FixPt Constant block is a masked subsystem that generates a constant value.

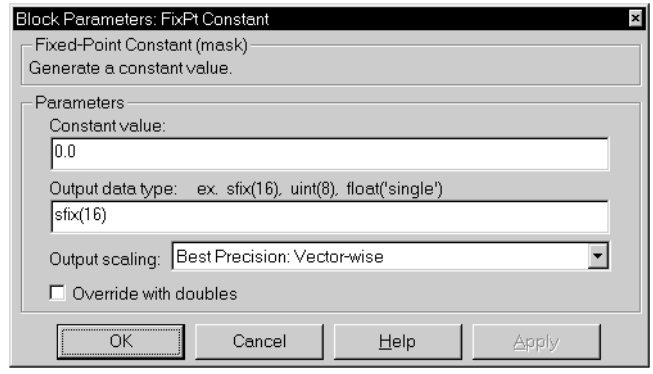
Constants are specified with the **Constant value** parameter. A constant can be a scalar or a vector. The **Output scaling** parameter supports these scaling modes when **Output data type** is a generalized fixed-point data type:

- `Use Specified Scaling` — This is the default mode; it uses the scaling specified for the **Output scaling** parameter.
- `Best Precision: Vector-wise` — This mode produces a common radix point for each element of the **Constant value** vector based on the best precision for the largest value of the vector.

For a description of all other parameters, refer to “Block Parameters” on page 9-4.

If the block is configured to generate a single constant value, the block icon displays the constant. If the block is configured to generate a vector of constants, the block icon displays `C`. For a description of how the block icon displays the output data type, refer to “Block Icon Display” on page 9-7.

## Parameters and Dialog Box




### Constant value

Constant value output by the block. It can be a scalar or vector. All members of the output vector must be the same data type.

**Output data type**

Any data type supported by the Fixed-Point Blockset.

**Output scaling**

Radix point-only or slope/bias scaling. Additionally, if **Constant value** is specified as a vector, it can be scaled using the constant vector scaling modes for maximizing precision. These scaling modes are only available for generalized fixed-point data types.

**Override with doubles**

If checked, **Output data type** is overridden with doubles.

## Conversions

The **Constant value** parameter is converted from a double to the specified output data type offline using round-to-nearest and saturation. Refer to "Parameter Conversions" on page 4-26 for more information about parameter conversions.

## Characteristics

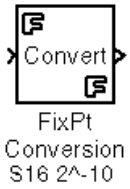
| | |
|---|---|
| Output Port | Any data type supported by the blockset |
| Direct Feedthrough | No |
| Sample Time | Inherited |
| Scalar Expansion | No — the output is always the same size as **Constant value** |
| States | 0 |
| Vectorized | Yes |

# FixPt Conversion

## Purpose

Convert from one Fixed-Point Blockset data type to another.

## Description

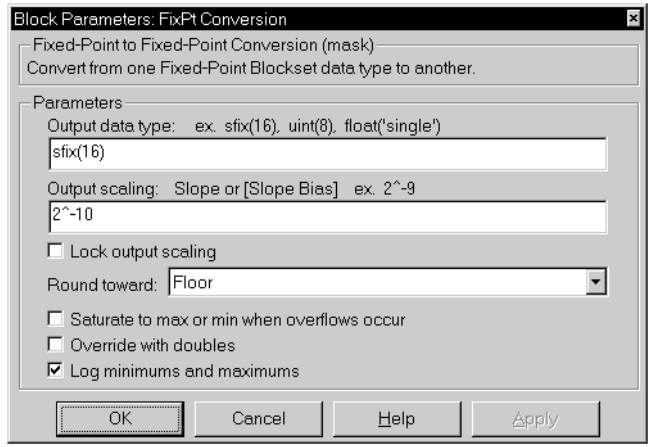



The FixPt Conversion block is a masked S-function that converts from one Fixed-Point Blockset data type to another.

For a description of all parameters, refer to “Block Parameters” on page 9-4. For a description of how the block icon displays the output data type, refer to “Block Icon Display” on page 9-7.

For more information about converting from one Fixed-Point Blockset data type to another, refer to “Signal Conversions” on page 4-26.

## Parameters and Dialog Box




### Output data type

Any data type supported by the Fixed-Point Blockset.

### Output scaling

Radix point-only or slope/bias scaling. These scaling modes are only available for generalized fixed-point data types.

### Lock output scaling

If checked, **Output scaling** is locked. This feature is only available for generalized fixed-point output.

### Round toward

Rounding mode for the fixed-point output.

**Saturate to max or min when overflows occur**

If checked, fixed-point overflows saturate. Otherwise, they wrap.

**Override with doubles**

If checked, the **Output data type** is overridden with doubles.

**Log minimums and maximums**

If checked, minimum and maximum simulation values are logged to the workspace.

## Characteristics

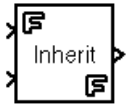
| | |
|---|---|
| Input Ports | Any data type supported by the blockset |
| Output Port | Any data type supported by the blockset |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | N/A |
| States | 0 |
| Vectorized | Yes |

# FixPt Conversion Inherited

## Purpose

Convert input two to the Fixed-Point Blockset data type of input one.

## Description

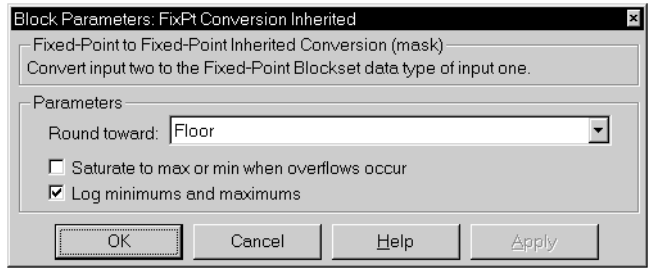



The FixPt Conversion Inherited block is a masked S-function that forces dissimilar data types to be the same. The first input is used as the reference data type and the second input is converted to the reference type. Either input will be scalar expanded such that the output has the same width as the widest input.

For a description of all parameters, refer to “Block Parameters” on page 9-4.

For more information about converting from one Fixed-Point Blockset data type to another, refer to “Signal Conversions” on page 4-26.

## Parameters and Dialog Box




**Round toward**

Rounding mode for the fixed-point output.

**Saturate to max or min when overflows occur**

If checked, fixed-point overflows saturate. Otherwise, they wrap.

**Log minimums and maximums**

If checked, minimum and maximum simulation values are logged to the workspace.

## Characteristics

| | |
|---|---|
| Input Ports | Any data type supported by the blockset |
| Output Port | Any data type supported by the blockset |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |

| | |
|---|---|
| Scalar Expansion | Yes |
| States | 0 |

# FixPt FIR

## Purpose

Implement a fixed-point finite impulse response (FIR) filter.

## Description


The FixPt FIR block is a masked S-function that samples and holds the *n* most recent inputs, multiplies each input by a specified value (its FIR coefficient), and stacks them in a vector. This block supports both single-input/single-output (SISO) and single-input/multi-output (SIMO) modes.

For the SISO mode, the **FIR coefficients** parameter is specified as a row vector. For the SIMO mode, the **FIR coefficients** are specified as a matrix where each row corresponds to a separate output.

The **Initial condition** parameter provides the initial values for all times preceding the start time in the FIR realization. The time interval between samples is specified with the **Sample time** parameter.

The **Parameter data type** parameter can have the same values as the **Output data type** parameter as discussed in "Block Parameters" on page 9-4. Additionally, if **Parameter data type** is a generalized fixed-point number, then you can select from these scaling modes:

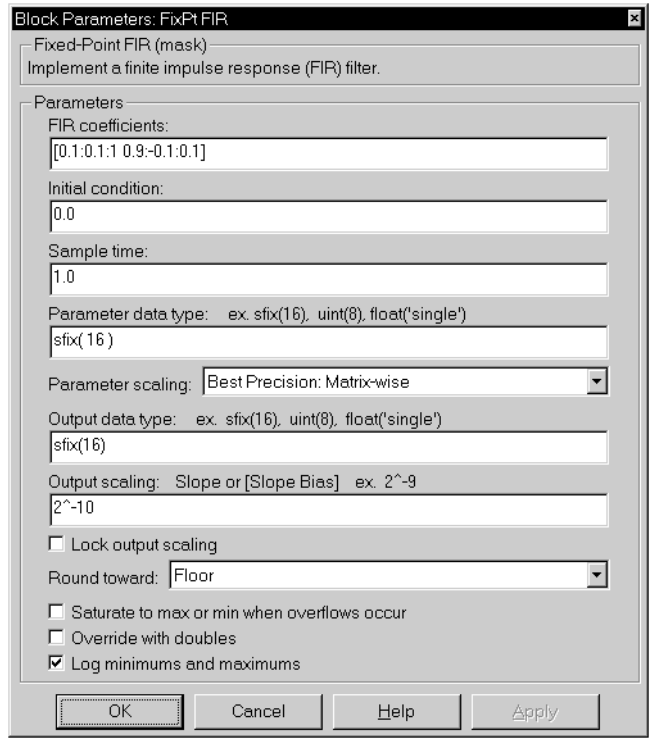
- `Use Specified Scaling` — This is the default mode; it uses the scaling from the **Parameter scaling** parameter.
- `Best Precision: Element-wise` — This mode produces radix points such that the precision is maximized for each element of the **Gain matrix value** matrix.
- `Best Precision: Row-wise` — This mode produces a common radix point for each element of a **Gain matrix value** row based on the best precision for the largest value of that row.
- `Best Precision: Column-wise` — This mode produces a common radix point for each element of a **Gain matrix value** column based on the best precision for the largest value of that column.
- `Best Precision: Matrix-wise` — This mode produces a common radix point for each element of the **Gain matrix value** matrix based on the best precision for the largest value of the matrix.

If the FIR coefficients are specified as a row vector, then scaling element-wise and column-wise produce the same result, while scaling matrix-wise and row-wise produce the same result.

For a description of all other parameters, refer to “Block Parameters” on page 9-4.

The block icon displays a graph of the FIR coefficients unless the SIMO mode is used. When a coefficient is changed on the block’s dialog box, the graph is automatically redrawn when you press the **Apply** or **OK** button. For a description of how the block icon displays the output data type, refer to “Block Icon Display” on page 9-7.

## Parameters and Dialog Box




### FIR coefficients

FIR coefficients. One row per output.

### Initial condition

Initial values for all times preceding the start time.

**Sample time**

Sample time.

**Parameter data type**

Any data type supported by the Fixed-Point Blockset.

**Parameter scaling**

Radix point-only or slope/bias scaling. Additionally, the **FIR coefficients** vector or matrix can be scaled using the constant vector or constant matrix scaling modes for maximizing precision. These scaling modes are only available for generalized fixed-point data types.

**Output data type**

Any data type supported by the Fixed-Point Blockset.

**Output scaling**

Radix point-only or slope/bias scaling. These scaling modes are only available for generalized fixed-point data types.

**Lock output scaling**

If checked, **Output scaling** is locked. This feature is only available for generalized fixed-point output.

**Round toward**

Rounding mode for the fixed-point output.

**Saturate to max or min when overflows occur**

If checked, fixed-point overflows saturate. Otherwise, they wrap.

**Override with doubles**

If checked, the **Output data type** is overridden with doubles.

**Log minimums and maximums**

If checked, minimum and maximum simulation values are logged to the workspace.

## Conversions and Operations

The **FIR coefficients** parameter is converted from doubles to the specified data type offline using round-to-nearest and saturation.The **Initial condition** parameter is converted from doubles to the input data type offline using round-to-nearest and saturation. Refer to “Parameter Conversions” on page 4-26 for more information about parameter conversions.

The FixPt FIR block first multiplies its inputs by the **FIR coefficients** parameter, converts those results to the output data type using the specified rounding and overflow modes, and then carries out the summation. Refer to "Rules for Arithmetic Operations" on page 4-29 for more information about the rules this block adheres to when performing operations.

## Characteristics

| | |
|---|---|
| Input Ports | Any data type supported by the blockset — it must be a scalar |
| Output Port | Any data type supported by the blockset |
| Direct Feedthrough | Yes |
| Sample Time | Specified as a parameter |
| Scalar Expansion | Of initial conditions |
| States | One less than the columns in the **FIR coefficients** vector or matrix |
| Vectorized | No |

## Example

Suppose you want to configure this block for two outputs (SIMO mode) where the first output is given by

$$y_1(k) = a_1 \cdot u(k) + b_1 \cdot u(k-1) + c_1 \cdot u(k-2)$$

the second output is given by

$$y_2(k) = a_2 \cdot u(k) + b_2 \cdot u(k-1)$$

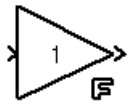
and the initial values of $u(k-1)$ and $u(k-2)$ are given by $ic_1$ and $ic_2$, respectively. To configure the FixPt FIR block for this situation, you must specify the **FIR coefficient** parameter as `[a1 b1 c1; a2 b2 c2]` where `c2 = 0`, and the **Initial condition** parameter as `[ic1 ic2]`.

# FixPt Gain

## Purpose

Multiply the input by a constant.

## Description




The FixPt Gain block is a masked S-function that multiplies the input by a constant value (referred to as the gain). To multiply the input by a constant matrix, see `FixPt Matrix Gain` on page 9-42.

The gain is specified with the **Gain value** parameter. The gain can be a scalar or a vector.

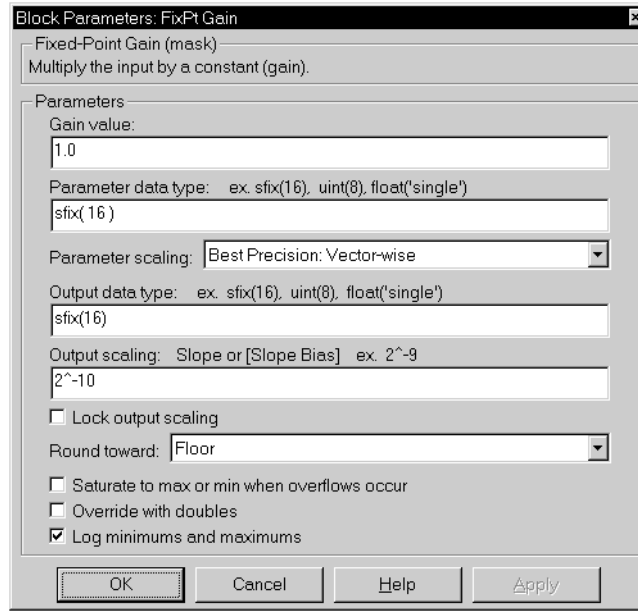
The **Parameter data type** parameter can have the same values as the **Output data type** parameter as discussed in "Block Parameters" on page 9-4. Additionally, if the **Parameter data type** is a generalized fixed-point number, then you can select from these scaling modes:

- `Use Specified Scaling` — This is the default mode; it uses the scaling from the **Parameter scaling** parameter.
- `Best Precision: Element-wise` — This mode produces radix points such that the precision is maximized for each element of the **Gain value** vector.
- `Best Precision: Vector-wise` — This mode produces a common radix point for each element of the **Gain value** vector based on the best precision for the largest value of the vector.

For a description of all other parameters, refer to "Block Parameters" on page 9-4.

If the block is configured for a single gain value, the block icon displays that value. If the block is configured to generate a vector of gains, the block icon displays `K`. For a description of how the block icon displays the output data type, refer to "Block Icon Display" on page 9-7.

## Parameters and Dialog Box




**Gain value**

Specify as a vector or scalar.

**Parameter data type**

Any data type supported by the Fixed-Point Blockset.

**Parameter scaling**

Radix point-only or slope/bias scaling. Additionally, if **Gain value** is specified as a vector, it can be scaled using the constant vector scaling modes for maximizing precision. These scaling modes are only available for generalized fixed-point data types.

**Output data type**

Any data type supported by the Fixed-Point Blockset.

**Output scaling**

Radix point-only or slope/bias scaling. These scaling modes are only available for generalized fixed-point data types.

**Lock output scaling**

If checked, **Output scaling** is locked. This feature is only available for generalized fixed-point output.

**Round toward**

Rounding mode for the fixed-point output.

**Saturate to max or min when overflows occur**

If checked, fixed-point overflows saturate. Otherwise, they wrap.

**Override with doubles**

If checked, the **Output data type** is overridden with doubles.

**Log minimums and maximums**

If checked, minimum and maximum simulation values are logged to the workspace.

## Conversions and Operations

The **Gain value** parameter is converted from doubles to the specified data type offline using round-to-nearest and saturation. Refer to "Parameter Conversions" on page 4-26 for more information about parameter conversions.

The FixPt Gain block first multiples its inputs by the **Gain value** parameter, and then converts those results to the output data type using the specified rounding and overflow modes. Refer to "Rules for Arithmetic Operations" on page 4-29 for more information about the rules this block adheres to when performing operations.

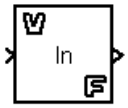
## Characteristics

| | |
|---|---|
| Input Ports | Any data type supported by the blockset |
| Output Port | Any data type supported by the blockset |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | Of inputs and gain |
| States | 0 |
| Vectorized | Yes |

## Purpose

Convert a Simulink data type to a Fixed-Point Blockset data type.

## Description



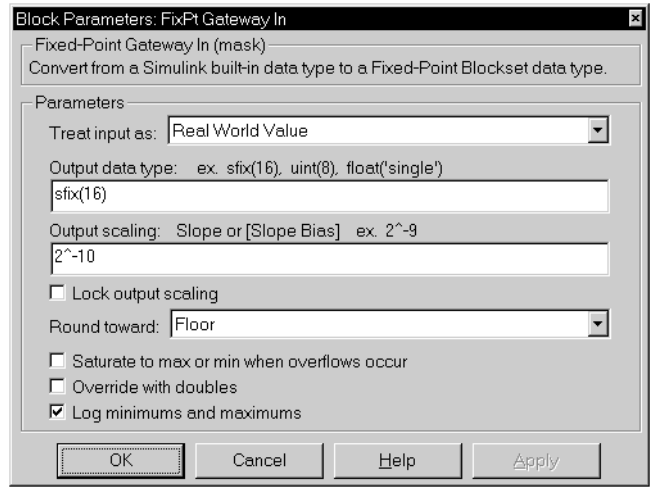

The FixPt Gateway In block is a masked S-function that converts a built-in Simulink data type to a Fixed-Point Blockset data type. Simulink's complex data type is not supported.

The **Treat input as** parameter controls how the input is specified. In terms of the general encoding scheme described in "Scaling" on page 3-4, `Real World Value` specifies the input as $V = SQ + B$ where $S$ is the slope and $B$ is the bias, while `Stored Integer` specifies the input as an integer, $Q$.

For a description of all other parameters, refer to "Block Parameters" on page 9-4.

If the block input is treated as a real-world value, then a "V" appears in the upper left corner of the icon. If the block input is treated as an integer, then an "I" appears in the upper left corner of the icon. For a description of how the block icon displays the output data type, refer to "Block Icon Display" on page 9-7.

## Parameters and Dialog Box




### Treat input as

Store the input as a real world value or as an integer.

**Output data type**

Any data type supported by the Fixed-Point Blockset.

**Output scaling**

Radix point-only or slope/bias scaling. These scaling modes are only available for generalized fixed-point data types.

**Lock output scaling**

If checked, **Output scaling** is locked. This feature is only available for generalized fixed-point output.

**Round toward**

Rounding mode for fixed-point output.

**Saturate to max or min when overflows occur**

If checked, fixed-point overflows saturate. Otherwise, they wrap.

**Override with doubles**

If checked, **Output data type** is overridden with doubles.

**Log minimums and maximums**

If checked, minimum and maximum simulation values are logged to the workspace.

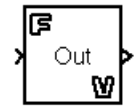
## Characteristics

| | |
|---|---|
| Input Port | Any built-in Simulink data type except complex |
| Output Port | Any data type supported by the blockset |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | No |
| States | 0 |
| Vectorized | Yes |

## Purpose

Convert a Fixed-Point Blockset data type to a Simulink data type.

## Description




The FixPt Gateway Out block is a masked S-function that converts any data type supported by the Fixed-Point Blockset to a Simulink data type.

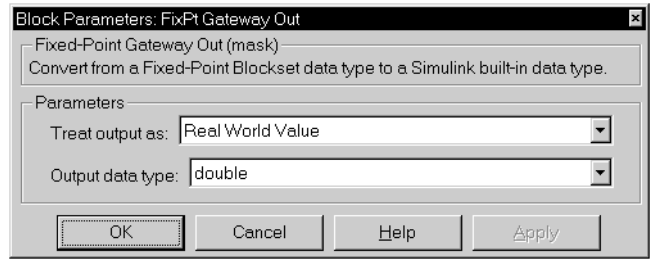
The **Treat output as** parameter controls how the output is specified. In terms of the general encoding scheme described in "Scaling" on page 3-4, `Real World Value` specifies the output as $V = SQ + B$ where $S$ is the slope and $B$ is the bias, while `Stored Integer` specifies the output as an integer, $Q$. Outputting numbers as `Stored Integer` may be useful in these circumstances:

- If generating code for a fixed-point processor, the resulting code is purely integer code and does not use floating-point operations.
- If you want to partition your model based on hardware characteristics. For example, part of your model may involve simulating hardware that produces integers as output.

The **Output data type** parameter specifies the Simulink data type. `auto` indicates the Fixed-Point Blockset data type is converted to whatever data type Simulink back-propagates. Simulink's complex data type is not supported.

If the block output is treated as a real-world value, then a "V" appears in the lower right corner of the icon. If the block output is treated as an integer, then an "I" appears in the lower right corner of the icon.

## Parameters and Dialog Box




**Treat output as**

Store the output as a real world value or as an integer.

**Output data type**

Any data type supported by Simulink except the complex data type.

# FixPt Gateway Out

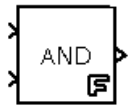
## Characteristics

| | |
|---|---|
| Input Ports | Any data type supported by the blockset |
| Output Port | Any built-in Simulink data type except complex |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | N/A |
| States | 0 |
| Vectorized | Yes |

## Purpose

Perform the specified logical operation on the inputs.

## Description




FixPt
Logical
Operator
U8

The FixPt Logical Operation block is a masked S-function that performs the specified logic on its inputs. An input value is TRUE (1) if it is nonzero and FALSE (0) if it is zero.

The logic connecting the inputs is selected with the **Operator** pull-down menu. The supported logical operations are given below.

| Logical Operator | Description |
|---|---|
| AND | TRUE if all inputs are TRUE |
| OR | TRUE if at least one input is TRUE |
| NAND | TRUE if at least one input is FALSE |
| NOR | TRUE when no inputs are TRUE |
| XOR | TRUE if an odd number of inputs are TRUE |
| NOT | TRUE if the input is FALSE |

The number of input ports is specified with the **Number of input ports** parameter. The output type is specified with the **Logical output data type** parameter. An output value is 1 if TRUE and 0 if FALSE.

---

**Note:** The **Logical output data type** selected should represent zero exactly. Data types that satisfy this condition include signed and unsigned integers and any floating-point data type.

---

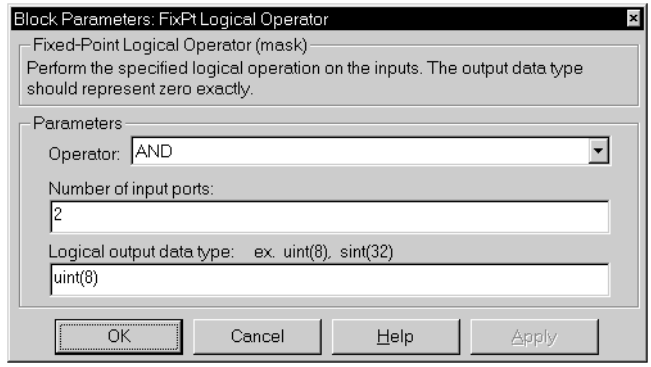
The size of the output depends on the number of inputs, their vector size, and the selected operator:

- For two or more inputs, the block performs the operation between all of the inputs. If the inputs are vectors, the operation is performed between corresponding elements of the vectors to produce a vector output.
- For a single vector input, the block applies the operation (except the NOT operator) to all elements of that vector. The NOT operator accepts only one input, which can be a scalar or vector. If the input is a vector, the output is a vector of the same size containing the logical complements of the elements of the input vector.

When configured as a multi-input XOR gate, this block performs an addition-modulo-two operation as mandated by the IEEE Standard for Logic Elements.

The block icon displays the logic to be performed on the inputs. For a description of how the block icon displays the output data type, refer to “Block Icon Display” on page 9-7.

## Parameters and Dialog Box




**Operator**

Logical operator used to connect the inputs.

**Number of input ports**

Number of inputs.

**Logical output data type**

Output data type. You should only use data types that represent zero exactly.

## Characteristics

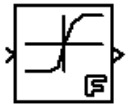
| | |
|---|---|
| Input Port(s) | Any data type supported by the blockset |
| Output Port | Any data type supported by the blockset that can exactly represent zero |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | Of inputs |
| States | 0 |
| Vectorized | Yes |

# FixPt Look-Up Table

## Purpose

Approximate a one-dimensional function using a selected look-up method.

## Description




The FixPt Look-Up Table block is a masked S-function that computes an approximation to some function `y=f(x)` given `x`, `y` data vectors. The look-up method can use interpolation, extrapolation, or the original values of the input. The length of the `x` and `y` data vectors provided to this block must match. Also, the `x` data vector must be strictly monotonically increasing after conversion to the input's fixed-point data type. Note that due to quantization, the `x` data vector may be strictly monotonic in doubles format, but not so after conversion to a fixed-point data type.

To map two inputs to an output, see `FixPt Look-Up Table (2D)` on page 9-37.

You define the table by specifying the **Vector of input values** parameter as a $1 \times n$ vector and the **Vector of output values** parameter as a $1 \times n$ vector.

The block generates output based on the input values using one of these methods selected from the **Method** parameter list:

- `Interpolation-Extrapolation` — This is the default method; it performs linear interpolation and extrapolation of the inputs.
  - If a value matches the block's input, the output is the corresponding element in the output vector.
  - If no value matches the block's input, then the block performs linear interpolation between the two appropriate elements of the table to determine an output value. If the block input is less than the first or greater than the last input vector element, then the block extrapolates using the first two or last two points.
- `Interpolation-Use End Values` — This method performs linear interpolation as described above but does not extrapolate outside the end points of the input vector. Instead, the end-point values are used.
- `Use Input Nearest` — This method does not interpolate or extrapolate. Instead, the element in `x` nearest the current input is found. The corresponding element in `y` is then used as the output.
- `Use Input Below` — This method does not interpolate or extrapolate. Instead, the element in `x` nearest and below the current input is found. The

corresponding element in y is then used as the output. If there is no element in x below the current input, then the nearest element is found.

- `Use Input Above` — This method does not interpolate or extrapolate. Instead, the element in x nearest and above the current input is found. The corresponding element in y is then used as the output. If there is no element in x above the current input, then the nearest element is found.

For a description of all other parameters, refer to "Block Parameters" on page 9-4.

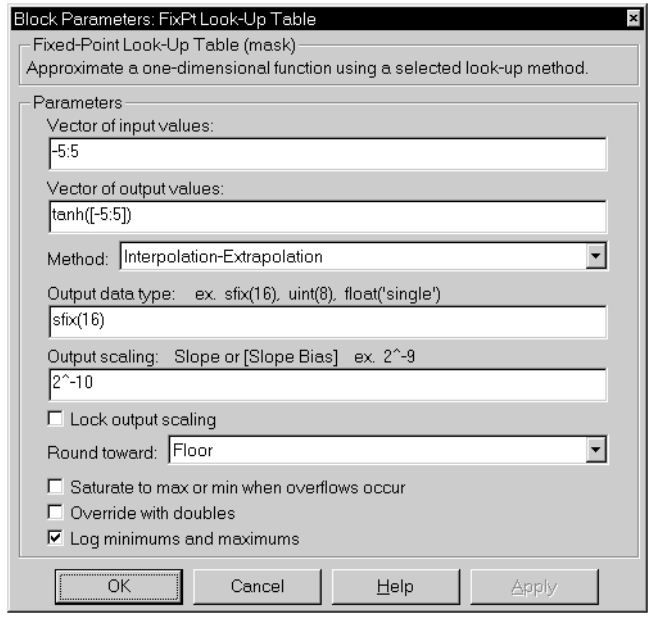
The block icon displays a graph of the input vector versus the output vector. When a parameter is changed on the block's dialog box, the graph is automatically redrawn when you press the **Apply** or **OK** button. For a description of how the block icon displays the output data type, refer to "Block Icon Display" on page 9-7.

---

**Note:** To avoid parameter saturation errors, the automatic scaling script `autofixexp` employs a special rule for the FixPt Look-Up Table block. `autofixexp` modifies the scaling by using the output look-up values in addition to the logged minimum and maximum simulation values. This prevents the data from being saturated to different values. For this block, the look-up values are given by the **Vector of output values** parameter (the `YDataPoints` variable).

---

## Parameters and Dialog Box




**Vector of input values**

The vector of input values must be the same size as the output vector and strictly monotonically increasing.

**Vector of output values**

The vector of output values must be the same size as the input vector.

**Method**

Look-up method.

**Output data type**

Any data type supported by the Fixed-Point Blockset.

**Output scaling**

Radix point-only or slope/bias scaling. These scaling modes are only available for generalized fixed-point data types.

**Lock output scaling**

If checked, **Output scaling** is locked. This feature is only available for generalized fixed-point output.

**Round toward**

Rounding mode for the fixed-point output.

**Saturate to max or min when overflows occur**

If checked, fixed-point overflows saturate. Otherwise, they wrap.

**Override with doubles**

If checked, the **Output data type** is overridden with doubles.

**Log minimums and maximums**

If checked, minimum and maximum simulation values are logged to the workspace.

## Conversions

The **Vector of input values** parameter is converted from doubles to the input data type. The **Vector of output values** parameter is converted from doubles to the output data type. Both conversion are performed offline using round-to-nearest and saturation. Refer to “Parameter Conversions” on page 4-26 for more information about parameter conversions.

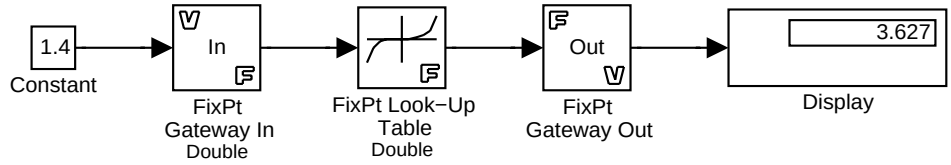
## Characteristics

| | |
|---|---|
| Input Port(s) | Any data type supported by the blockset |
| Output Port | Any data type supported by the blockset |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | No |
| States | 0 |
| Vectorized | Yes |

## Example

This example illustrates the look-up methods supported by the FixPt Look-Up Table block.

Suppose you have a vector of input values given by `[-5:5]` and a vector of output values given by `sinh([-5:5])`. Using the model shown below,




these results were generated.

| Look-Up Method | Input | Output | Comment |
|---|---|---|---|
| `Interpolation-Extrapolation` | `1.4` | `2.153` | N/A |
| | `5.2` | `83.59` | N/A |
| `Interpolation-Use End Values` | `1.4` | `2.153` | N/A |
| | `5.2` | `74.2` | The value for `sinh(5.0)` was used. |
| `Use Input Nearest` | `1.4` | `1.175` | The value for `sinh(1.0)` was used. |
| `Use Input Below` | `1.4` | `1.175` | The value for `sinh(1.0)` was used. |
| | `-5.2` | `-74.2` | The value for `sinh(-5.0)` was used. |
| `Use Input Above` | `1.4` | `3.627` | The value for `sinh(2.0)` was used. |
| | `5.2` | `74.2` | The value for `sinh(5.0)` was used. |

## Purpose

Approximate a two-dimensional function using a selected look-up method.

## Description


The FixPt Look-Up Table (2-D) block is a masked S-function that computes an approximation to some function `z=f(x,y)` given `x`, `y`, `z` data points. The look-up method can use interpolation, extrapolation, or the original values of the inputs. Also, the `x` and `y` data vectors must be strictly monotonically increasing. (see FixPt Look-Up Table description).

The **Row** parameter is a $1 \times m$ vector of `x` data points, the **Col** parameter is a $1 \times n$ vector of `y` data points, and the **Table** parameter is an $m \times n$ matrix of `z` data points. Both the row and column vectors must be strictly monotonically increasing.

The block generates output based on the input values using one of these methods selected from the **Method** parameter list:

- `Interpolation-Extrapolation` — This is the default method; it performs linear interpolation and extrapolation of the inputs.
  - If the inputs match row and column parameter values, the output is the value at the intersection of the row and column.
  - If the inputs do not match row and column parameter values, then the block generates output by linearly interpolating between the appropriate row and column values. If either or both block inputs are less than the first or greater than the last row or column values, the block extrapolates from the first two or last two points.
- `Interpolation-Use End Values` — This method performs linear interpolation as described above but does not extrapolate outside the end points of the input vector. Instead, the end-point values are used.
- `Use Input Nearest` — This method does not interpolate or extrapolate. Instead, the elements in `x` and `y` nearest the current inputs are found. The corresponding element in `z` is then used as the output.
- `Use Input Below` — This method does not interpolate or extrapolate. Instead, the elements in `x` and `y` nearest and below the current inputs are found. The corresponding element in `z` is then used as the output.If there are no elements in `x` or `y` below the current inputs, then the nearest elements are found.
- `Use Input Above` — This method does not interpolate or extrapolate. Instead, the elements in `x` and `y` nearest and above the current inputs are

found. The corresponding element in `z` is then used as the output. If there are no elements in `x` or `y` above the current inputs, then the nearest elements are found.

For a description of all other parameters, refer to "Block Parameters" on page 9-4.

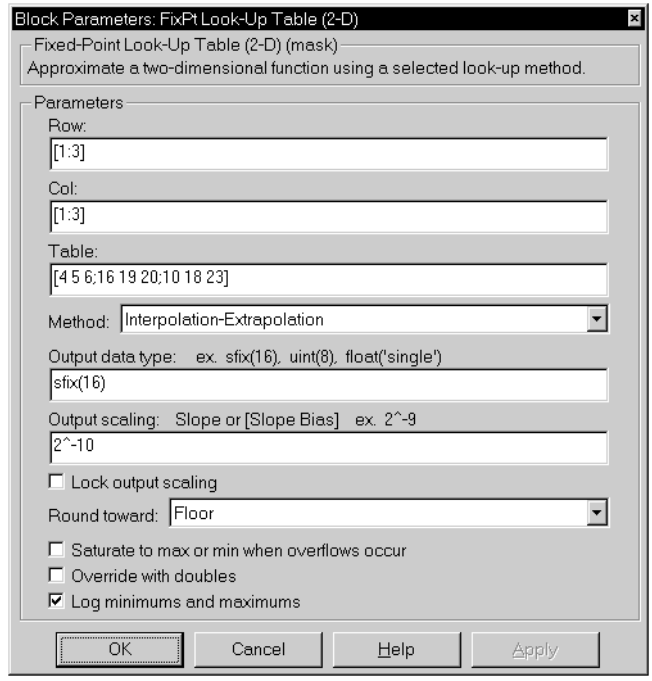
The block icon displays *m* graphs based on the contents of the **Table** parameter. When a parameter is changed on the block's dialog box, the graphs are automatically redrawn when you press the **Apply** or **OK** button. For a description of how the block icon displays the output data type, refer to "Block Icon Display" on page 9-7.

---

**Note:** To avoid parameter saturation errors, the automatic scaling script `autofixexp` employs a special rule for the FixPt Look-Up Table (2D) block. `autofixexp` modifies the scaling by using the output look-up values in addition to the logged minimum and maximum simulation values. This prevents the data from being saturated to different values. For this block, the look-up values are given by the **Table** parameter (the `TableDataPoints` variable).

---

## Parameters and Dialog Box




**Row**

Input row vector. It must be strictly monotonically increasing.

**Col**

Input column vector. It must be strictly monotonically increasing.

**Table**

Output vector. It must match the size defined by the **Row** and **Col** parameters.

**Method**

Look-up method.

**Output data type**

Any data type supported by the Fixed-Point Blockset.

**Output scaling**

Radix point-only or slope/bias scaling. These scaling modes are only available for generalized fixed-point data types.

**Lock output scaling**

If checked, **Output scaling** is locked. This feature is only available for generalized fixed-point output.

**Round toward**

Rounding mode for the fixed-point output.

**Saturate to max or min when overflows occur**

If checked, fixed-point overflows saturate. Otherwise, they wrap.

**Override with doubles**

If checked, the **Output data type** is overridden with doubles.

**Log minimums and maximums**

If checked, minimum and maximum simulation values are logged to the workspace.

## Conversions

The **Row** parameter is converted from doubles to the first input's data type. The **Column** parameter is converted from doubles to the second input's data type. The **Table** parameter is converted from doubles to the output data type. All conversion are performed offline using round-to-nearest and saturation. Refer to "Parameter Conversions" on page 4-26 for more information about parameter conversions.

## Characteristics

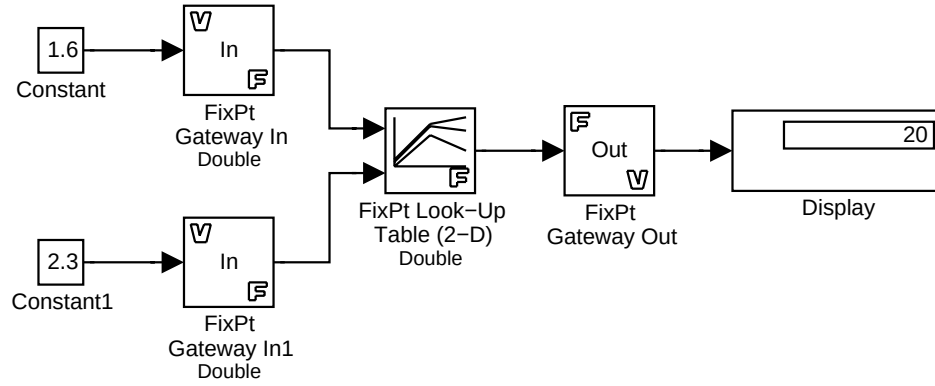
| | |
|---|---|
| Input Ports | Any data type supported by the blockset |
| Output Port | Any data type supported by the blockset |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | Of one input if the other is a vector |
| States | 0 |
| Vectorized | Yes |

## Example

This example illustrates the look-up methods supported by the FixPt Look-Up Table (2D) block.

Suppose you have input row and column vectors given by `[1:3]` and a look-up table given by `[4 5 6; 16 19 20; 10 18 23]`.

Using the model shown below,




these results were generated.

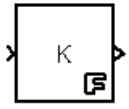
| Look-Up Method | Input [x y] | Output | Comment |
|---|---|---|---|
| Interpolation-Extrapolation | [1.6 2.5] | 13.9 | N/A |
| | [1.6 4.0] | 15.4 | N/A |
| Interpolation-Use End Values | [1.6 2.5] | 13.9 | N/A |
| | [1.6 4.0] | 14.4 | The value for [1.6 3] was used. |
| Use Input Nearest | [1.6 2.3] | 19 | The value for [2 2] was used. |
| Use Input Below | [1.6 2.3] | 5 | The value for [1 2] was used. |
| | [1.6 0.5] | 4 | The value for [1 1] was used. |
| Use Input Above | [1.6 2.3] | 20 | The value for [2 3] was used. |
| | [1.6 3.5] | 20 | The value for [2 3] was used. |

# FixPt Matrix Gain

## Purpose

Multiply the input by a constant matrix.

## Description




The FixPt Matrix Gain block is a masked S-function that multiplies the input by a constant matrix (referred to as the matrix gain). The block generates its output by multiplying the input by a specified matrix

$$y = Ku$$

where $K$ is the matrix gain and $u$ is the input. If the matrix has `m` rows and `n` columns, then the input to this block should be a vector of length `n`. The output is a vector of length `m`.

The gain is specified with the **Gain matrix value** parameter.

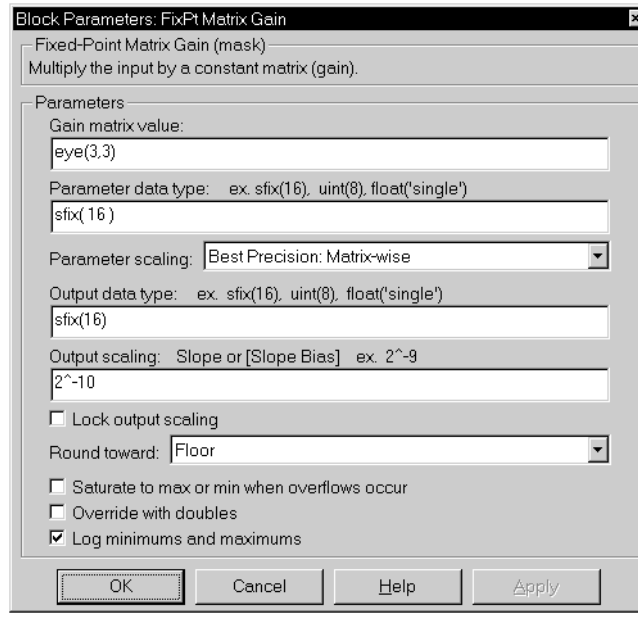
The **Parameter data type** parameter can have the same values as the **Output data type** parameter as discussed in "Block Parameters" on page 9-4. Additionally, if **Parameter data type** is a generalized fixed-point number, then you can select from these scaling modes:

- `Use Specified Scaling` —This is the default mode; it uses the scaling from the **Parameter scaling** parameter.
- Use `Best Precision: Element-wise` — This mode produces radix points such that the precision is maximized for each element of the **Gain matrix value** matrix.
- Use `Best Precision: Row-wise` — This mode produces a common radix point for each element of a **Gain matrix value** row based on the best precision for the largest value of that row.
- Use `Best Precision: Column-wise` — This mode produces a common radix point for each element of a **Gain matrix value** column based on the best precision for the largest value of that column.
- Use `Best Precision: Matrix-wise` — This mode produces a common radix point for each element of the **Gain matrix value** matrix based on the best precision for the largest value of the matrix.

For a description of all other parameters, refer to "Block Parameters" on page 9-4.

The block icon always displays a `K`. For a description of how the block icon displays the output data type, refer to "Block Icon Display" on page 9-7.

## Parameters and Dialog Box




**Gain matrix value**

Specify as a scalar or vector.

**Parameter data type**

Any data type supported by the Fixed-Point Blockset.

**Parameter scaling**

Radix point-only or slope/bias scaling. Additionally, the gain can be scaled using the constant matrix scaling modes for maximizing precision. These scaling modes are only available for generalized fixed-point data types.

**Output data type**

Any data type supported by the Fixed-Point Blockset.

**Output scaling**

Radix point-only or slope/bias scaling. These scaling modes are only available for generalized fixed-point data types.

**Lock output scaling**

If checked, **Output scaling** is locked.This feature is only available for generalized fixed-point output.

**Round toward**

Rounding mode for the fixed-point output.

**Saturate to max or min when overflows occur**

If checked, fixed-point overflows saturate. Otherwise, they wrap.

**Override with doubles**

If checked, the **Output data type** is overridden with doubles.

**Log minimums and maximums**

If checked, minimum and maximum simulation values are logged to the workspace.

## Conversions and Operations

The **Gain matrix value** parameter is converted from doubles to the specified data type offline using round-to-nearest and saturation. Refer to “Parameter Conversions” on page 4-26 for more information about parameter conversions.

The FixPt Matrix Gain block first multiples its inputs by the **Gain matrix value** parameter, converts those results to the output data type using the specified rounding and overflow modes, and then performs the summation. Refer to “Rules for Arithmetic Operations” on page 4-29 for more information about the rules this block adheres to when performing operations.

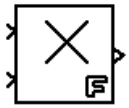
## Characteristics

| | |
|---|---|
| Input Ports | Any data type supported by the blockset |
| Output Port | Any data type supported by the blockset |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | No |
| States | 0 |
| Vectorized | Yes |

## Purpose

Multiply or divide inputs.

## Description




The FixPt Product block is a masked S-function that performs multiplication or division of its inputs.

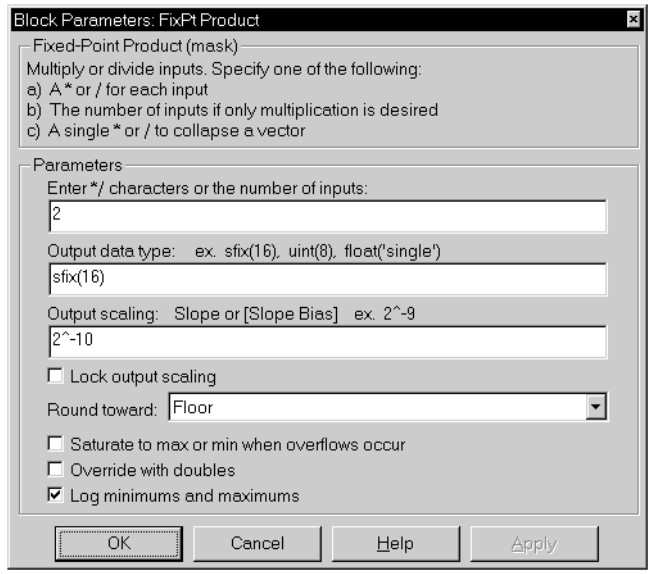
The operations are specified with the **Enter */ characters or the number of inputs** parameter. The multiply-divide characters indicate which operations are to be performed:

- If there are two or more inputs, then the number of multiply-divide characters must equal the number of inputs. For example, '*/*' requires three inputs and configures the block to divide the first input by the second and multiply the third. If the first character is '/', then the first input is inverted.
- If only multiplication of inputs is required, then a numeric parameter value equal to the number of inputs can be supplied instead of multiply-divide characters.
- If only one vector is input, then a single '*' or '/' will collapse the vector using the specified operation.

For a description of all other parameters, refer to "Block Parameters" on page 9-4.

The block icon indicates which operations are performed on the inputs. For a description of how the block icon displays the output data type, refer to "Block Icon Display" on page 9-7.

## Parameters and Dialog Box




**Enter */ characters or the number of inputs**

Enter as many multiply or divide characters as there are inputs. For multiplication only, you can enter the number of inputs since this is the default operation.

**Output data type**

Any data type supported by the Fixed-Point Blockset.

**Output scaling**

Radix point-only or slope/bias scaling. These scaling modes are only available for generalized fixed-point data types.

**Lock output scaling**

If checked, **Output scaling** is locked. This feature is only available for generalized fixed-point output.

**Round toward**

Rounding mode for the fixed-point output.

**Saturate to max or min when overflows occur**

If checked, fixed-point overflows saturate. Otherwise, they wrap.

**Override with doubles**

If checked, the **Output data type** is overridden with doubles.

**Log minimums and maximums**

If checked, minimum and maximum simulation values are logged to the workspace.

## Operations

The FixPt Product block first performs the specified multiply or divide operations on the inputs, and then converts the results to the output data type using the specified rounding and overflow modes. Refer to "Rules for Arithmetic Operations" on page 4-29 for more information about the rules this block adheres to when performing operations.

## Characteristics

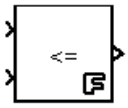
| | |
|---|---|
| Input Ports | Any data type supported by the blockset |
| Output Port | Any data type supported by the blockset |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | Yes |
| States | 0 |
| Vectorized | Yes |

# FixPt Relational Operator

## Purpose

Perform the specified relational operation on the inputs.

## Description




FixPt Relational Operator U8

The FixPt Relational Operator block is a masked S-function that performs a comparison of its two inputs. The first input is converted the data type of the second input prior to comparison.

The operator connecting the two inputs is selected with the **Operator** parameter list. The supported relational operators are given below.

| Relational Operator | Description |
|---|---|
| == | TRUE if the first input is equal to the second input |
| != | TRUE if the first input is not equal to the second input |
| < | TRUE if the first input is less than the second input |
| <= | TRUE if the first input is less than or equal to the second input |
| >= | TRUE if the first input is greater than or equal to the second input |
| > | TRUE if the first input is greater than the second input |

The output is specified with the **Logical output data type** parameter. The output equals 1 for TRUE and 0 for FALSE.

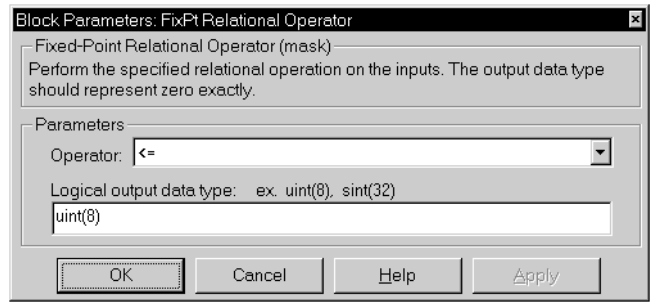
---

**Note:** The **Logical output data type** selected should represent zero exactly. Data types that satisfy this condition include signed and unsigned integers and any floating-point data type.

---

The block icon displays the logic to be performed on the inputs. For a description of how the block icon displays the output data type, refer to “Block Icon Display” on page 9-7.

## Parameters and Dialog Box




**Operator**

Relational operator used to compare the two inputs.

**Logical output data type**

Output data type. You should only use data types that can represent zero exactly.

## Conversions

The input with the smallest positive range is converted to the data type of the other input offline using round-to-nearest and saturation. This conversion is performed prior to comparison. Refer to “Parameter Conversions” on page 4-26 for more information about parameter conversions.

## Characteristics

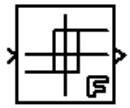
| | |
|---|---|
| Input Port | Any data type supported by the blockset |
| Output Port | Any data type supported by the blockset that can exactly represent zero |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | Of inputs |
| States | 0 |
| Vectorized | Yes |

# FixPt Relay

## Purpose

Switch output between two constants.

## Description




The FixPt Relay block is a masked S-function that allows the output to switch between two specified values. When the relay is on, it remains on until the input drops below the value of the **Switch off point** parameter. When the relay is off, it remains off until the input exceeds the value of the **Switch on point** parameter. The block accepts one input and generates one output.

Specifying a **Switch on point** value greater than the **Switch off point** value models hysteresis, whereas specifying equal values models a switch with a threshold at that value. The **Switch on point** value must be greater than or equal to the **Switch off point**.

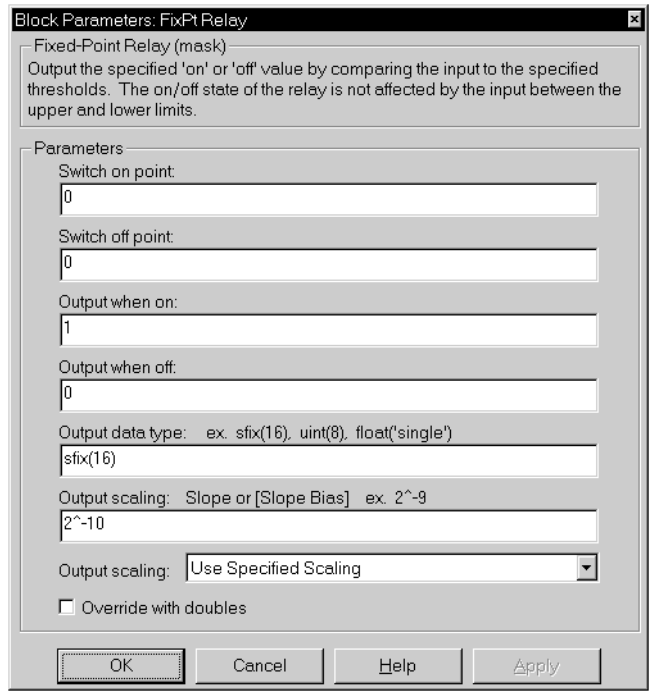
The **Output scaling** parameter supports these scaling modes when **Output data type** is a generalized fixed-point data type:

- `Use Specified Scaling` — This is the default mode; it uses the scaling specified for the **Output scaling** parameter.
- `Best Precision: Vector-wise` — This mode produces the best precision based on the **Output when on** and **Output when off** parameters.

For a description of all other parameters, refer to “Block Parameters” on page 9-4.

For a description of how the block icon displays the output data type, refer to “Block Icon Display” on page 9-7.

## Parameters and Dialog Box




**Switch on point**

The on threshold for the relay.

**Switch off point**

The off threshold for the relay.

**Output when on**

The output when the relay is on.

**Output when off**

The output when the relay is off.

**Output data type**

Any data type supported by the Fixed-Point Blockset.

**Output scaling**

Radix point-only or slope/bias scaling. Additionally, the **Output when on** and **Output when off** parameters can be scaled using the constant vector

scaling mode for maximizing precision.These scaling modes are only available for generalized fixed-point data types.

**Override with doubles**

If checked, the **Output data type** is overridden with doubles.

## Conversions

Both the **Switch on point** and **Switch off point** parameters are converted to the input data type offline using round-to-nearest and saturation.

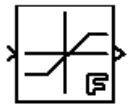
## Characteristics

| | |
|---|---|
| Input Port | Any data type supported by the blockset |
| Output Port | Any data type supported by the blockset |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | Yes |
| States | 0 |
| Vectorized | Yes |

## Purpose

Bound the range of the input.

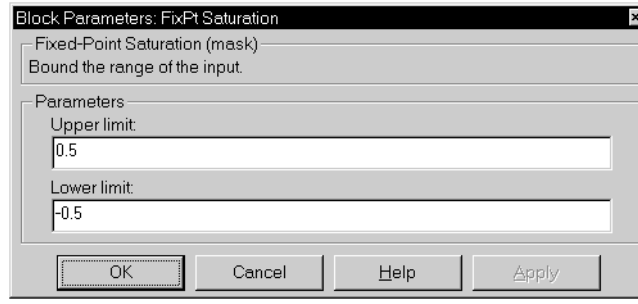
## Description




The FixPt Saturation block is a masked S-function that limits the input signal to upper and lower saturation values.

You specify the upper bound of the input with the **Upper limit** parameter and the lower bound of the input with the **Lower limit** parameter. If the input signal is outside these limits, the output saturates to one of the bounds.

## Parameters and Dialog Box




**Upper limit**

The upper bound on the input signal.

**Lower limit**

The lower bound on the input signal.

## Conversions

Both the **Upper limit** and **Lower limit** parameters are converted to the input data type offline using round-to-nearest and saturation.

## Characteristics

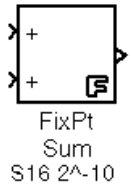
| | |
|---|---|
| Input Port | Any data type supported by the blockset |
| Output Port | Same as input data type |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | Of input and limits |
| States | 0 |
| Vectorized | Yes |

# FixPt Sum

## Purpose

Add or subtract inputs.

## Description




The FixPt Sum block is a masked S-function that performs addition or subtraction on its inputs.

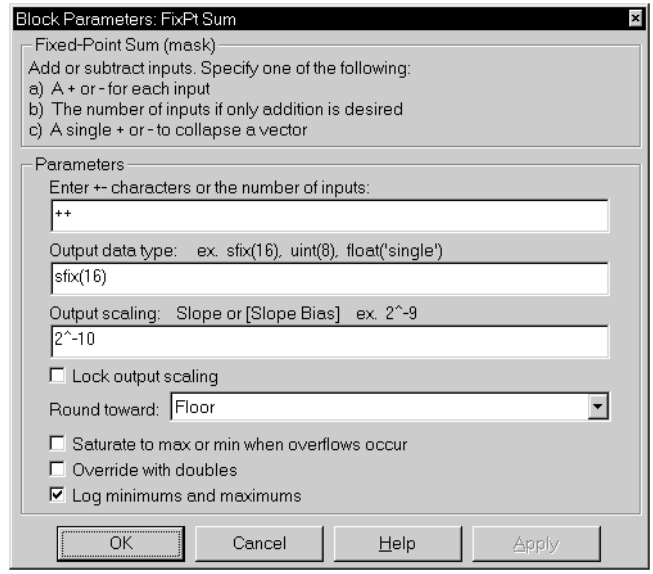
The operations are specified with the **Enter +- characters or the number of inputs** parameter. The plus-minus characters indicate the operations to be performed on the inputs:

- If there are two or more inputs, then the number of plus-minus characters must equal the number of inputs. For example, '+-+' requires three inputs and configures the block to subtract the second input from the first and add the third.
- If only addition of inputs is required, then a numeric parameter value equal to the number of inputs can be supplied instead of plus-minus characters.
- If only one vector is input, then a single '+' or '-' will collapse the vector using the specified operation.

For a description of all other parameters, refer to "Block Parameters" on page 9-4.

The block icon indicates which operations are performed on the inputs. For a description of how the block icon displays the output data type, refer to "Block Icon Display" on page 9-7.

## Parameters and Dialog Box




**Enter +- characters or the number of inputs**

Enter as many plus or minus characters as there are inputs. For addition only, you can enter the number of inputs since this is the default operation.

**Output data type**

Any data type supported by the Fixed-Point Blockset.

**Output scaling**

Radix point-only or slope/bias scaling. These scaling modes are only available for generalized fixed-point data types.

**Lock output scaling**

If checked, **Output scaling** is locked. This feature is only available for generalized fixed-point output.

**Round toward**

Rounding mode for the fixed-point output.

**Saturate to max or min when overflows occur**

If checked, fixed-point overflows saturate. Otherwise, they wrap.

**Override with doubles**

If checked, the **Output data type** is overridden with doubles.

**Log minimums and maximums**

If checked, minimum and maximum simulation values are logged to the workspace.

## Operations

The FixPt Sum block first converts the input data type(s) to the output data type using the specified rounding and overflow modes, and then performs the specified operations. Refer to "Rules for Arithmetic Operations" on page 4-29 for more information about the rules this block adheres to when performing operations.

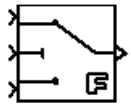
## Characteristics

| | |
|---|---|
| Input Ports | Any data type supported by the blockset |
| Output Port | Any data type supported by the blockset |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | Yes |
| States | 0 |
| Vectorized | Yes |

## Purpose

Switch output between input one or input three based on the value of input two.

## Description




The FixPt Switch block is a masked S-function that passes through input one or input three based on the value of input two. Input two is called the *control input*.

Input one is passed through when input two is greater than or equal to the value of the **Threshold** parameter. Otherwise, it passes through input three. The threshold value is converted to the data type of input two.

For a description of all other parameters, refer to “Block Parameters” on page 9-4.

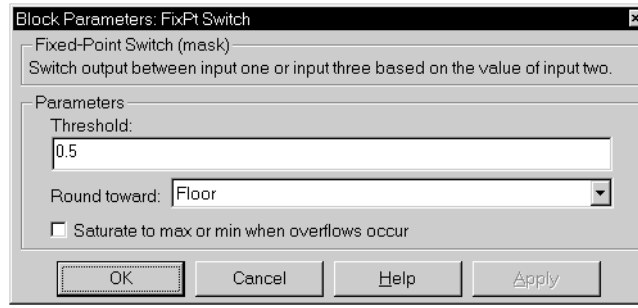
---

**Note:** The output data type is determined by the input with the largest positive range. If input 1 has a larger positive range than input 3, then it specifies the output data type. Otherwise, input 3 specifies the output data type.

---

The block icon displays which input is being passed to the output.

## Parameters and Dialog Box




### Threshold

Switch threshold that determines which input is passed to the output.

### Round toward

Rounding mode for the fixed-point output.

**Saturate to max or min when overflows occur**

If checked, fixed-point overflows saturate. Otherwise, they wrap.

## Conversions

The **Threshold** parameter is converted offline to the second input's data type using round-to-nearest and saturation. Refer to "Parameter Conversions" on page 4-26 for more information about parameter conversions.

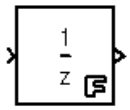
## Characteristics

| | |
|---|---|
| Input Ports | Any data type supported by the blockset |
| Output Port | Same as input port one |
| Direct Feedthrough | Yes |
| Sample Time | Inherited |
| Scalar Expansion | Yes |
| States | 0 |
| Vectorized | Yes |

## Purpose

Delay a signal one sample period.

## Description




The FixPt Unit Delay block delays its input by the specified sample period. This block is equivalent to the $z^{-1}$ discrete-time operator. The block accepts one input and generates one output, both of which can be scalar or vector. If the input is a vector, all elements of the vector are delayed by the same sample period.

This block provides a mechanism for discretizing one or more signals in time, or resampling the signal at a different rate. If your model contains multirate transitions, then you must add FixPt Unit Delay blocks between the slow to fast transitions. The sample rate of the FixPt Unit Delay must be set to that of the slower block.

For fast to slow transitions, refer to `FixPt Zero-Order Hold` on page 9-61. For more information about multirate transitions, refer to the *Using Simulink* book or the *Real-Time Workshop User's Guide*.

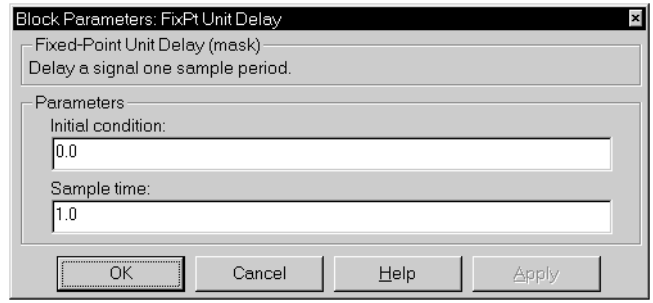
The block output for the first sampling period is specified by the **Initial condition** parameter. Careful selection of this parameter can minimize unwanted output behavior. The time between samples is specified with the **Sample time** parameter.

---

**Note:** The FixPt Unit Delay block accepts continuous sample times. When it has a continuous sample time, the block is equivalent to the built-in Memory block.

---

## Parameters and Dialog Box



**Initial condition**

The initial output of the simulation.

**Sample time**

Sample time.

## Conversions

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

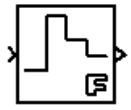
## Characteristics

| | |
|---|---|
| Input Port | Any data type supported by the blockset |
| Output Port | Same as the input |
| Direct Feedthrough | No |
| Sample Time | Discrete or continuous |
| Scalar Expansion | Of input or initial conditions |
| States | As many as there are outputs |
| Vectorized | Yes |

## Purpose

Implement a zero-order hold of one sample period.

## Description



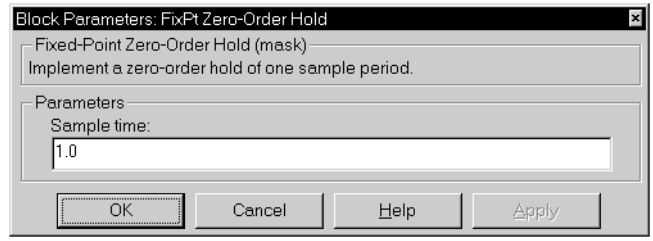

The FixPt Zero-Order Hold block samples and holds its input for the specified sample period. The block accepts one input and generates one output, both of which can be scalar or vector. If the input is a vector, all elements of the vector are held for the same sample period.

This block provides a mechanism for discretizing one or more signals in time, or resampling the signal at a different rate. If your model contains multirate transitions, you must add FixPt Zero-Order Hold blocks between the fast to slow transitions. The sample rate of the FixPt Zero-Order Hold must be set to that of the slower block.

For slow to fast transitions, refer to `FixPt Unit Delay` on page 9-59. For more information about multirate transitions, refer to the *Using Simulink* book or the *Real-Time Workshop User's Guide*.

The time between samples is specified with the **Sample time** parameter.

## Parameters and Dialog Box




**Sample time**

Sample time.

## Characteristics

| | |
|---|---|
| Input Port | Any data type supported by the blockset |
| Output Port | Same as the input |
| Direct Feedthrough | Yes |
| Sample Time | Discrete |
| Scalar Expansion | No |

| | |
|---|---|
| States | 0 |
| Vectorized | Yes |

A

# Code Generation


Languages . . . . . A-2
Storage Class of Variables . . . . . A-2
Storage Class of Parameters . . . . . A-2
Rounding Modes . . . . . A-2
Overflow Handling . . . . . A-3
Blocks . . . . . A-3
Scaling . . . . . A-3
Pure Integer Code . . . . . A-3

With the Real-Time Workshop, you can generate C code for execution on a fixed-point embedded processor. 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 code is structured so that key operations can be readily replaced by optimized target-specific libraries that you supply. You can also use the Target Language Compiler to customize the generated code. For more information about code generation, refer to the *Real-Time Workshop User's Guide* and the *Target Language Compiler Reference Guide*.

All fixed-point blocks support code generation, but not every simulation feature is supported. The code generation support is described below.

## Languages

- C support only
- No Ada support

## Storage Class of Variables

- Integer sizes provided by the target C compiler are supported. For example, if a C compiler for a particular embedded processor specifies 8-bit characters and 16-, 32-, and 64-bit integers, then variables can be 8, 16, 32, or 64 bits, either signed or unsigned.
- No floating-point support

## Storage Class of Parameters

- The Real-Time Workshop external mode support requires that parameters be 8, 16, or 32 bits, either signed or unsigned. The parameter size must also be compatible with the target C compiler.
- No floating-point support

## Rounding Modes

- All four rounding modes are supported.
- Round-to-floor generates the most efficient code for most cases.

## Overflow Handling

- Saturation mode is supported.
- Wrapping mode is supported and generates the most efficient code.
- Automatic exclusion of saturation code when hardware saturation is available is currently not supported. Wrapping must be selected for the Real-Time Workshop to exclude saturation code.

## Blocks

All blocks generate code for all operations with a few exceptions:

- The FixPt Look-Up Table and FixPt Look-Up Table (2D) blocks generate code for all look-up methods except extrapolation.
- A few combinations of scaling and operations lead to highly inefficient code. These few cases are excluded and described in the next section.

## Scaling

- Radix point-only scaling is supported.
- Slope/bias scaling is supported for all blocks except when it leads to highly inefficient code. All blocks except four support all cases of slope/bias scaling. These four blocks, FixPt Product, FixPt Gain, FixPt Matrix Gain, and FixPt FIR support matched slope/bias scaling where the block input signals and output signals have the same slopes and biases, but not mismatched slope/bias scaling. For more information about matched and mismatched slope/bias scaling, refer to “Signal Conversions” on page 4-26.

  It is generally recommended that signals with slope/bias scaling (such as a sensor input) are immediately converted to radix point-only scaling. This will typically produce more efficient code.

## Pure Integer Code

All blocks generate pure integer code except for the FixPt Gateway In and FixPt Gateway Out blocks. These blocks must generate floating-point code when handling floating-point input or output. However, if the input or output is an integer and the block is configured to treat the input or output as a stored integer, then these blocks will also generate pure integer code.

B

# 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 Clifs, New Jersey. 1990.

**7** Ogata, K., *Discrete-Time Control Systems*, *Second Edition*; Prentice Hall, Englewood Cliffs, New Jersey, 1995.

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

# Index


## A

accumulation
  scaling recommendations 4-18
  using slope/bias encoding 4-18
accumulator data type 7-4
  in feedback controller demo 6-8
addition
  blockset rules 4-29
  scaling recommendations 4-16
  using slope/bias encoding 4-15
ALU's 4-29
arithmetic shift 4-41
`autofixexp` 8-3
automatic scaling
  and feedback controller demo 6-11
automatic scaling tools
  GUI 8-8
  script 8-3

## B

backward integrator realization 7-9
base data type 7-4
  in feedback controller demo 6-8
binary point 3-3
bit multipliers 3-7
bit shifts 4-40
bits 3-3
block configuration 2-10
  selecting a data type 2-11
  selecting a data type scaling 2-13
block icon display 9-7
block parameters 9-4
  *See also* dialog box parameters
blocks
  FixPt Constant 9-12
  FixPt Conversion 4-43, 9-14
  FixPt Conversion Inherited 9-16
  FixPt FIR 4-46, 9-18
  FixPt Gain 4-44, 9-22
  FixPt Gateway In 1-9, 2-10, 9-25
  FixPt Gateway Out 1-9, 9-27
  FixPt Logical Operator 9-29
  FixPt Look-Up Table 9-32
  FixPt Look-Up Table (2-D) 9-37
  FixPt Matrix Gain 3-12, 9-42
  FixPt Product 4-36, 4-40, 9-45
  FixPt Relational Operator 9-48
  FixPt Relay 9-50
  FixPt Saturation 9-53
  FixPt Sum 4-32, 9-54
  FixPt Switch 9-57
  FixPt Unit Delay 9-59
  FixPt Zero-Order Hold 9-61
Bode plot 6-6

## C

chopping 4-8
code generation A-2
  and multiplication 4-35
  and signal conversions 4-28
  and summation 4-31
computational noise 4-2
  and rounding 4-3
computational units 4-29
contiguous bits 3-16
conversions 4-26
  parameter 4-26
  signal 4-26
  *See also* online conversion, offline conversion
converting old models 8-2, 8-6

## D

data type selection 9-4
data types 2-11, 3-9
  fractional numbers 2-11
  generalized fixed-point numbers 2-12
  IEEE numbers 2-12
  integers 2-11
demos 2-20
denormalized numbers 3-21
derivative realization
  filtered 7-12
  unfiltered 7-14
development cycle 1-6
dialog box parameters
  data type 9-4
  lock output scaling 9-6
  logging min/max data 9-7
  overflow handling 9-6
  overriding with doubles 9-6
  rounding 9-6
  scaling 9-5
digital controller 6-7
digital filter 5-2
direct form realization 5-4
  in feedback controller demo 6-8
division
  blockset rules 4-38
  scaling recommendations 4-22, 4-23
  using slope/bias encoding 4-22
double bits 4-34, 7-4
double precision format 3-18
dynamic parameter dialog box 2-14

## E

encoding scheme 3-5
`eps` 3-21
examples
  constant scaling for best precision 3-11
  conversions and arithmetic operations 4-45
  converting from doubles to fixed-point 2-15
  division process 4-39
  fixed-point format 3-7
  fixed-point scaling 3-9
  FixPt FIR 9-21
  FixPt Look-Up Table 9-36
  FixPt Look-Up Table (2D) 9-40
  limitations on precision and errors 4-9
  limitations on range 4-14
  maximizing precision 4-10
  multiplication process 4-36
  saturation and wrapping 4-12
  selecting a measurement scale 2-4
  shifting bits and the radix point 4-42
  shifting bits but not the radix point 4-43
  summation process 4-31
exceptional arithmetic 3-21
exponent for IEEE numbers 3-17

## F

feedback design 6-3
filter
  digital 5-2
  lead-lag 7-18
filtered derivative realization 7-12
filters and systems 7-2
`FixDispPref` 2-18, 9-7
fixed-point GUI 8-8
  and feedback controller demo 6-9
fixed-point numbers
  general format 3-3
  scaling 3-4
`fixptlib` 2-9

FixUseDbl 6-11
forward integrator realization 7-10
fpupdate 8-6
fraction for IEEE numbers 3-17
fractional numbers 2-11
  and guard bits 4-14
fractional slope 3-5
fxptdlg 8-8

## G

gain
  scaling recommendations 4-21, 4-22
  using slope/bias encoding 4-20
generalized fixed-point numbers 2-12
global override with doubles 6-11
guard bits 4-14
GUI 8-8

## H

help 1-14
hidden bit 3-17

## I

icon display. *See* block icon display
IEEE
  double precision format 3-18
  general format 3-17
  nonstandard format 3-19
  precision 3-20
  range 3-20
  single precision format 3-18
infinity 3-22, 4-11
installation xvi
integers 2-11
integrator realization
  backward 7-9
  forward 7-10
  trapezoidal 7-7

## L

least significant bit 3-3
library 2-9
limit cycles 4-2
  in feedback controller demo 6-15
lock output scaling 9-6
  and feedback controller demo 6-14
logging simulation results 9-7
logical shift 4-41
LSB. *See* least significant bit

## M

MAC's 4-29
maximum absolute error 9-7
measurement scales 2-2
modeling the system 1-6
most significant bit 3-3
MSB. *See* most significant bit
multiplication
  blockset rules 4-34
  scaling recommendations 4-19, 4-20
  using slope/bias encoding 4-19

## N

NaNs 3-22, 4-11
nonstandard IEEE format 3-19

## O

offline conversions 4-26
  for addition and subtraction 4-30
  for multiplication 4-35
  for signals 4-27
online conversions
  for addition and subtraction 4-30
  for multiplication 4-35
  for signals 4-27
online help 1-14
overflow 3-20, 4-2, 4-11
overflow handling 4-11, 9-6
overriding with doubles 9-6
  and feedback controller demo 6-11
  global override 6-11
  individual override 6-16

## P

padding with trailing zeros 4-8
parallel form realization 5-9
parameter conversion 4-26
precision
  of fixed-point numbers 3-9
  of IEEE floating-point numbers 3-20
prerequisites xi, xiv

## Q

quantization 4-2
  and feedback controller demo 6-10
  and rounding 4-3
  of a real-world value 2-17, 3-6

## R

radix point 3-3
radix point-only scaling 3-5
range
  of fixed-point numbers 3-8
  of IEEE floating-point numbers 3-20
`RangeFactor` 6-12, 8-3
realizations
  and data types 7-3
  and scaling 7-3
  derivative 7-12
  design constraints 5-2
  direct form 5-4
  integrator 7-7
  lead-lag filter 7-18
  parallel form 5-9
  series cascade form 5-7
  state-space 7-21
real-world value 3-5
release information 1-14
rounding modes 4-3, 9-6
  toward ceiling 4-6
  toward floor 4-7
  toward nearest 4-5
  toward zero 4-4

## S

saturation 4-11
scaling 2-13, 9-5
  and accumulation 4-18
  and addition 4-15
  and division 4-22
  and gain 4-20
  and multiplication 4-19
  constant scaling for best precision 3-11
  radix point-only 2-13, 3-5
  slope/bias 2-13, 3-6
scientific notation 3-15

series cascade form realization 5-7
shifts 4-40
  using the FixPt Conversion block 4-42
  using the FixPt Gain block 4-43
showfixptsimranges 8-13
  and feedback controller demo 6-11
sign bit for IEEE numbers 3-17
sign extension 4-14
signal comparison tool 8-9
signal conversions 4-26
single precision format 3-18
slope/bias scaling 3-6
state-space realization 7-21
stored integer value 2-16, 9-27
subtraction. *See* addition

## T

targeting an embedded processor 7-4
  design rules 7-5
  operation assumptions 7-4
  size assumptions 7-4
trapezoidal integrator realization 7-7
truncation 4-8
two's complement 3-3
typographical conventions xv

## U

underflow 3-20
unfiltered derivative. *See* derivative realization
updating old models 8-6

## W

wrapping 4-11