

Fixed-Point Toolbox™ 3

User's Guide

MATLAB®

How to Contact MathWorks



www.mathworks.com Web
comp.soft-sys.matlab Newsgroup
www.mathworks.com/contact_TS.html Technical Support



suggest@mathworks.com Product enhancement suggestions
bugs@mathworks.com Bug reports
doc@mathworks.com Documentation error reports
service@mathworks.com Order status, license renewals, passcodes
info@mathworks.com Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Fixed-Point Toolbox™ User's Guide

© COPYRIGHT 2004–2011 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

June 2004	First printing	New for Version 1.0 (Release 14)
October 2004	Online only	Version 1.1 (Release 14SP1)
March 2005	Online only	Version 1.2 (Release 14SP2)
September 2005	Online only	Version 1.3 (Release 14SP3)
October 2005	Second printing	Version 1.3
March 2006	Online only	Version 1.4 (R2006a)
September 2006	Third printing	Version 1.5 (R2006b)
March 2007	Fourth printing	Version 2.0 (R2007a)
September 2007	Online only	Revised for Version 2.1 (R2007b)
March 2008	Online only	Revised for Version 2.2 (R2008a)
October 2008	Online only	Revised for Version 2.3 (R2008b)
March 2009	Online only	Revised for Version 2.4 (R2009a)
September 2009	Online only	Revised for Version 3.0 (R2009b)
March 2010	Online only	Revised for Version 3.1 (R2010a)
September 2010	Online only	Revised for Version 3.2 (R2010b)
April 2011	Online only	Revised for Version 3.3 (R2011a)

Getting Started

1

Product Overview	1-2
System Setup	1-3
Installation	1-3
Required Products	1-3
Related Products	1-4
Licensing	1-4
Getting Help	1-5
Getting Help in This Document	1-5
Getting Help at the MATLAB Command Line	1-5
Display Settings	1-7
Displaying the fimath Properties of fi Objects	1-8
Hiding the fimath Properties of fi Objects	1-9
Shortening the numerictype Display of fi Objects	1-10
Demos	1-11

Fixed-Point Concepts

2

Fixed-Point Data Types	2-2
Scaling	2-4
Precision and Range	2-5
Range	2-5
Precision	2-6

Arithmetic Operations	2-10
Modulo Arithmetic	2-10
Two's Complement	2-11
Addition and Subtraction	2-12
Multiplication	2-13
Casts	2-19
fi Objects Compared to C Integer Data Types	2-22
Integer Data Types	2-22
Unary Conversions	2-24
Binary Conversions	2-25
Overflow Handling	2-28

Working with fi Objects

3

Constructing fi Objects	3-2
fi Object Syntaxes	3-2
Examples of Constructing fi Objects	3-3
Casting fi Objects	3-12
Overwriting by Assignment	3-12
Ways to Cast with MATLAB Software	3-12
fi Object Properties	3-17
Data Properties	3-17
fimath Properties	3-17
numerictype Properties	3-19
Setting fi Object Properties	3-20
fi Object Functions	3-24

Constructing fimath Objects	4-2
fimath Object Syntaxes	4-2
Building fimath Object Constructors in a GUI	4-4
Working with the Global fimath	4-6
Configuring the Global fimath	4-7
Saving the Global fimath	4-8
Resetting the Global fimath to the Factory Default	4-8
Using the Global fimath with the MATLAB Function Block	4-10
fimath Object Properties	4-11
Math, Rounding, and Overflow Properties	4-11
Setting fimath Object Properties	4-12
Using fimath Properties to Perform Fixed-Point Arithmetic	4-16
fimath Rules for Fixed-Point Arithmetic	4-16
Binary-Point Arithmetic	4-18
[Slope Bias] Arithmetic	4-22
Using fimath to Specify Rounding and Overflow Modes	4-25
Using fimath to Share Arithmetic Rules	4-27
Using the Global fimath to Share Arithmetic Rules	4-27
Using fimath Objects to Share Arithmetic Rules	4-28
Using fimath ProductMode and SumMode	4-30
Example Setup	4-30
FullPrecision	4-31
KeepLSB	4-32
KeepMSB	4-33
SpecifyPrecision	4-34
fimath Object Functions	4-36

Working with fipref Objects

5

Constructing fipref Objects	5-2
fipref Object Properties	5-3
Display, Data Type Override, and Logging Properties	5-3
Setting fipref Object Properties	5-3
Using fipref Objects to Set Display Preferences	5-5
Using fipref Objects to Set Logging Preferences	5-7
Logging Overflows and Underflows as Warnings	5-7
Accessing Logged Information with Functions	5-9
Using fipref Objects to Set Data Type Override Preferences	5-12
Overriding the Data Type of fi Objects	5-12
Using Data Type Override to Help Set Fixed-Point Scaling	5-13
fipref Object Functions	5-15

Working with numerictype Objects

6

Constructing numerictype Objects	6-2
numerictype Object Syntaxes	6-2
Example: Constructing a numerictype Object with Property Name and Property Value Pairs	6-3
Example: Copying a numerictype Object	6-4
Example: Building numerictype Object Constructors in a GUI	6-5
numerictype Object Properties	6-7
Data Type and Scaling Properties	6-7
Setting numerictype Object Properties	6-8

The numerictype Structure	6-11
Valid Values for numerictype Structure Properties	6-11
Properties That Affect the Slope	6-13
Stored Integer Value and Real World Value	6-13
Using numerictype Objects to Share Data Type and	
Scaling Settings of fi objects	6-14
Example 1	6-14
Example 2	6-15
numerictype Object Functions	6-17

Working with quantizer Objects

7

Constructing quantizer Objects	7-2
quantizer Object Properties	7-3
Quantizing Data with quantizer Objects	7-4
Transformations for Quantized Data	7-6
quantizer Object Functions	7-7

Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms

8

What Are Code Acceleration and Code Generation from MATLAB?	8-3
--	------------

Requirements for Generating MEX Files from MATLAB Algorithms	8-4
Functions Supported for Code Acceleration and Code Generation from MATLAB	8-5
Workflow for Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms	8-15
Setting Up a Supported C Compiler to Generate MEX Functions	8-16
Using fiaccel	8-17
Speeding Up Fixed-Point Execution with the fiaccel Function	8-17
Running fiaccel	8-17
Generated Files and Locations	8-18
Using Data Type Override with fiaccel	8-21
Setting Up File Infrastructure and Paths	8-22
Compile Path Search Order	8-22
When to Use the Code Generation Path	8-22
Add Files to the Code Generation Path	8-23
Adding Folders to Search Paths	8-23
Naming Conventions	8-23
Preparing MATLAB Algorithms for Code Generation ..	8-25
Debugging Strategies	8-25
Detecting Errors at Design Time	8-26
Detecting Errors at Compile Time	8-26
Setting MEX Compilation Options	8-28
Working with the MEX Compiler Configuration Object ...	8-28
Modifying Compilation Options at the Command Line	
Using Dot Notation	8-28
MEX Compiler Options Parameters	8-29
How fiaccel Resolves Conflicting Options	8-35
Specifying Properties of Primary Function Inputs	8-36
Why You Must Specify Input Properties	8-36

Properties to Specify	8-36
Rules for Specifying Properties of Primary Inputs	8-40
Methods for Defining Properties of Primary Inputs	8-40
Defining Input Properties by Example at the Command Line	8-41
Best Practices for Accelerating Fixed-Point MATLAB	
Code	8-49
Recommended Compilation Options for fiaccel	8-49
Using Build Scripts	8-50
Using the MATLAB Code Analyzer to Check Code Interactively at Design Time	8-51
Separating Your Test Bench from Your Function Code ...	8-52
Preserving Your Code	8-52
File Naming Conventions	8-52
Working with Fixed-Point Code Generation Reports ..	
Generating the Code Generation Report	8-53
Opening the Code Generation Report	8-54
Viewing Your MATLAB Code	8-54
Viewing Variables in the Variables Tab	8-56
See Also	8-57
Generating MEX Functions from MATLAB Code That	
Uses Global Data	8-58
Workflow Overview	8-58
Declaring Global Variables	8-58
Defining Global Data	8-59
Synchronizing Global Data with MATLAB	8-60
Limitations of Using Global Data	8-63
Defining Input Properties Programmatically in the	
MATLAB File	8-64
How to Use assert with fiaccel	8-64
Rules for Using assert Function	8-69
Example: Specifying Properties of Primary Fixed-Point Inputs	8-69
Example: Specifying Class and Size of Scalar Structure ..	8-70
Example: Specifying Class and Size of Structure Array ..	8-71
Controlling Run-Time Checks	
Types of Run-Time Checks	8-73

When to Disable Run-Time Checks	8-74
How to Disable Run-Time Checks	8-74
MATLAB® Coder	8-76
MATLAB Function Block	8-77
Composing a MATLAB Language Function in a Simulink Model	8-77
Using the MATLAB Function Block with Data Type Override	8-77
Using Fixed-Point Data Types with the MATLAB Function Block	8-79
Example: Implementing a Fixed-Point Direct Form FIR Using the MATLAB Function Block	8-86

Interoperability with Other Products

9

Using fi Objects with Simulink	9-2
Reading Fixed-Point Data from the Workspace	9-2
Writing Fixed-Point Data to the Workspace	9-2
Setting the Value and Data Type of Block Parameters ...	9-6
Logging Fixed-Point Signals	9-6
Accessing Fixed-Point Block Data During Simulation	9-6
Using fi Objects with DSP System Toolbox	9-7
Reading Fixed-Point Signals from the Workspace	9-7
Writing Fixed-Point Signals to the Workspace	9-7
Using fi Objects with dfilter Objects	9-11
Using fiaccel, MATLAB® Coder, or Simulink to Generate Code	9-12

Index

Getting Started

- “Product Overview” on page 1-2
- “System Setup” on page 1-3
- “Getting Help” on page 1-5
- “Display Settings” on page 1-7
- “Demos” on page 1-11

Product Overview

Fixed-Point Toolbox™ provides fixed-point data types and arithmetic in MATLAB®. The toolbox lets you design fixed-point algorithms using MATLAB syntax and execute them at compiled C-code speed. You can reuse these algorithms in Simulink® and pass fixed-point data to and from Simulink models, facilitating bit-true simulation, implementation, and analysis and enabling you to generate test sequences for fixed-point software and hardware verification.

Key features of Fixed-Point Toolbox software include:

- Fixed-point data types in MATLAB with word lengths up to 65535 bits
- Global and local settings for performing fixed-point arithmetic
- Logical and bitwise operators and native integers
- Fixed-point data types usable in both MATLAB and Simulink
- Data logging, data-type override, and other tools for floating-to-fixed-point conversion
- Accelerated execution of fixed-point algorithms in MATLAB

System Setup

In this section...

“Installation” on page 1-3

“Required Products” on page 1-3

“Related Products” on page 1-4

“Licensing” on page 1-4

Installation

Before you begin working, you need to install the product on your computer.

Installing the Fixed-Point Toolbox Software

Fixed-Point Toolbox software uses the same installation procedure as the MATLAB toolboxes. See the MATLAB installation documentation for instructions.

Installing Online Documentation

Installing the documentation is part of the installation process:

- Installation from a DVD — Start the MathWorks® installer. When prompted, select the **Product** check boxes for the products you want to install. The documentation is installed along with the products.
- Installation from a Web download — If you update the Fixed-Point Toolbox software using a Web download and you want to view the documentation with the MATLAB Help browser, you must install the documentation on your hard drive.

Download the files from the Web. Then, start the installer, and select the **Product** check boxes for the products you want to install. The documentation is installed along with the products.

Required Products

The Fixed-Point Toolbox product is part of a family of MathWorks products. To use the toolbox, you must also have a MATLAB license. For more

information about Fixed-Point Toolbox system and product requirements, see [System Requirements](#) on the MathWorks Web site.

Note You can accelerate Fixed-Point Toolbox software when you have a compiler installed on your machine. For the current list of supported compilers, see [Supported and Compatible Compilers](#) on the MathWorks Web site. To setup or modify your compiler configuration, run `mex -setup` at the MATLAB command line.

Related Products

MathWorks provides several products that are relevant to the kinds of tasks you can perform with Fixed-Point Toolbox software.

See [Related Products](#) on the MathWorks Web site for more information.

Licensing

You can use `fi` objects with the `DataType` property set to `double` *without* a Fixed-Point Toolbox™ license when you set the `fipref LoggingMode` property to `off`. A Fixed-Point Toolbox license *is* checked out when you

- Use any `fi` object with any `DataType` other than `double`.
- Create any `fi` object with the `fipref LoggingMode` property set to `on`, including `fi` objects with `DataType double`.
- Load a MAT-file that contains any `fi` object with the `DataType` property set to `single`, `boolean`, `ScaledDouble`, or `Fixed`.

You can prevent the checkout of a Fixed-Point Toolbox™ license when working with Fixed-Point Toolbox™ code by setting the `fipref DataTypeOverride` property to `TrueDoubles`.

Getting Help

In this section...
“Getting Help in This Document” on page 1-5
“Getting Help at the MATLAB Command Line” on page 1-5

Getting Help in This Document

The following chapters discuss the objects of Fixed-Point Toolbox software:

- Chapter 3, “Working with fi Objects”
- Chapter 4, “Working with fimath Objects”
- Chapter 5, “Working with fipref Objects”
- Chapter 6, “Working with numerictype Objects”
- Chapter 7, “Working with quantizer Objects”

To get in-depth information about the properties of these objects, refer to the Property Reference.

To get in-depth information about the functions of these objects, refer to the Function Reference.

Getting Help at the MATLAB Command Line

To get command-line help for Fixed-Point Toolbox objects, type

```
help objectname
```

For example,

```
help fi
help fimath
help fipref
help numerictype
help quantizer
```

To get command-line help for Fixed-Point Toolbox functions, type

```
help embedded.fi/functionname
```

For example,

```
help embedded.fi/abs  
help embedded.fi/bitset  
help embedded.fi/sqrt
```

To invoke Help Browser documentation for Fixed-Point Toolbox functions from the MATLAB command line, type

```
doc fixedpoint/functionname
```

For example,

```
doc fixedpoint/int  
doc fixedpoint/add  
doc fixedpoint/savefipref  
doc fixedpoint/quantize
```

Display Settings

In Fixed-Point Toolbox software, the `fipref` object determines the display properties of `fi` objects. Code examples throughout this User's Guide generally show `fi` objects as they appear with the following `fipref` object properties:

- `NumberDisplay` — 'RealWorldValue'
- `NumericTypeDisplay` — 'full'
- `FimathDisplay` — 'full'

Setting '`FimathDisplay`' to 'full' provides a quick and easy way to differentiate between `fi` objects with a local `fimath` and those that are associated with the global `fimath`. When '`FimathDisplay`' is set to 'full', MATLAB displays `fimath` object properties for `fi` objects with a local `fimath`. MATLAB never displays `fimath` object properties for `fi` objects that are associated with the global `fimath`. Because of this display difference, you can tell when a `fi` object is associated with the global `fimath` just by looking at the output.

Additionally, unless otherwise specified, examples throughout the Fixed-Point Toolbox documentation use the following configuration of the global `fimath`:

```
RoundMode: nearest
OverflowMode: saturate
ProductMode: FullPrecision
MaxProductWordLength: 128
SumMode: FullPrecision
MaxSumWordLength: 128
```

Note If you are using a different global `fimath`, you may see different results when running the examples.

For more information on display settings, refer to Chapter 5, “Working with `fipref` Objects”.

Displaying the fimath Properties of fi Objects

To see the output as it appears in most Fixed-Point Toolbox code examples, set your `fipref` properties as follows and create two `fi` objects:

```
p = fipref('NumberDisplay', 'RealWorldValue',...  
'NumericTypeDisplay', 'full', 'FimathDisplay', 'full');  
a = fi(pi,'RoundMode', 'floor', 'OverflowMode', 'wrap')  
b = fi(pi)
```

MATLAB returns the following:

```
a =  
    3.1415  
  
          DataTypeMode: Fixed-point: binary point scaling  
          Signedness: Signed  
          WordLength: 16  
          FractionLength: 13  
  
          RoundMode: floor  
          OverflowMode: wrap  
          ProductMode: FullPrecision  
MaxProductWordLength: 128  
          SumMode: FullPrecision  
MaxSumWordLength: 128  
  
b =  
    3.1416  
  
          DataTypeMode: Fixed-point: binary point scaling  
          Signedness: Signed  
          WordLength: 16  
          FractionLength: 13
```

MATLAB displays `fimath` object properties in the output of `fi` object `a` because `a` has a local `fimath`.

MATLAB does not display any `fimath` object properties in the output of `fi` object `b` because `b` associates itself with the global `fimath`.

Hiding the fimath Properties of fi Objects

If you are working with multiple `fi` objects that have local `fimaths`, you may want to turn off the `fimath` object display:

- `NumberDisplay` — `'RealWorldValue'`
- `NumericTypeDisplay` — `'full'`
- `FimathDisplay` — `'none'`

For example,

```
p = fipref('NumberDisplay', 'RealWorldValue',...
'NumericTypeDisplay', 'full', 'FimathDisplay', 'none')

p =

    NumberDisplay: 'RealWorldValue'
 NumericTypeDisplay: 'full'
    FimathDisplay: 'none'
      LoggingMode: 'Off'
  DataTypeOverride: 'ForceOff'

F = fimath('RoundMode', 'floor', 'OverflowMode', 'wrap');
a = fi(pi, F)

a =

    3.1415

    DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
 FractionLength: 13
```

Although this setting helps decrease the amount of output produced, it also makes it impossible to tell from the output whether a `fi` object is associated with the global `fimath`. To do so, you can use the `isfimathlocal` function. For example,

```
isfimathlocal(a)
```

```
ans =  
    1
```

When the `isfimathlocal` function returns 1, the `fi` object has a local `fimath`. If the function returns 0, the `fi` object is associated with the global `fimath`. For more information on the global `fimath`, see “Working with the Global `fimath`” on page 4-6 in the *Fixed-Point Toolbox User’s Guide*.

Shortening the numericType Display of fi Objects

To reduce the amount of output even further, you can set the `NumericTypeDisplay` to 'short'. For example,

```
p = fipref('NumberDisplay', 'RealWorldValue',...  
          'NumericTypeDisplay', 'short', 'FimathDisplay', 'full');  
  
a = fi(pi)  
  
a =  
    3.1416  
    s16,13
```

Demos

You can find interactive Fixed-Point Toolbox demos in the MATLAB Help browser. Fixed-Point Toolbox software includes the following demos:

- Fixed-Point Basics — Demonstrates the basic use of the fixed-point `fi` object
- Number Circle — Illustrates the definitions of unsigned and signed two's complement integer and fixed-point numbers
- Binary Point Scaling — Explains binary point-only scaling
- Fixed-Point Data Type Override, Min/Max Logging, and Scaling — Steps through the workflow of using doubles override and min/max logging in the toolbox to choose appropriate scaling for a fixed-point algorithm
- Fixed-Point C Development — Shows how to use the parameters from a fixed-point MATLAB program in a fixed-point C program
- Fixed-Point Algorithm Development — Presents the development and verification of a simple fixed-point algorithm
- Fixed-Point Fast Fourier Transform (FFT) — Provides an example of converting a textbook Fast Fourier Transform algorithm into fixed-point MATLAB code and then into fixed-point C code
- Analysis of a Fixed-Point State-Space System with Limit Cycles — Demonstrates a limit cycle detection routine applied to a state-space system
- Quantization Error — Demonstrates the statistics of the error when signals are quantized using various rounding methods
- Fixed-Point Lowpass Filtering Using MATLAB for Code Generation — Steps through generating a MEX function from MATLAB code, running the generated MEX function, and displaying the results
- Fixed-Point ATAN2 Calculation — Uses the CORDIC algorithm and polynomial approximation to perform a fixed-point calculation of the four quadrant inverse tangent
- Fixed-Point Sine and Cosine Calculation — Uses the CORDIC approximation functions to compute the sine and cosine of fixed-point data

To access these demos, click the **Demos** entry for Fixed-Point Toolbox in the **Contents** pane of the Help browser, or type `demo('toolbox','fixed-point')` at the MATLAB command line.

Fixed-Point Concepts

- “Fixed-Point Data Types” on page 2-2
- “Scaling” on page 2-4
- “Precision and Range” on page 2-5
- “Arithmetic Operations” on page 2-10
- “fi Objects Compared to C Integer Data Types” on page 2-22

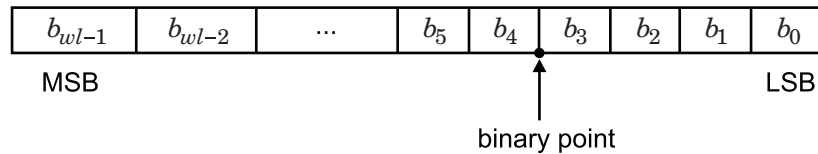
Fixed-Point Data Types

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

Binary numbers are represented as either fixed-point or floating-point data types. This chapter discusses many terms and concepts relating to fixed-point numbers, data types, and mathematics.

A fixed-point data type is characterized by the word length in bits, the position of the binary point, and whether it is signed or unsigned. The position of the binary point is the means by which fixed-point values are scaled and interpreted.

For example, a binary representation of a generalized fixed-point number (either signed or unsigned) is shown below:



where

- b_i is the i^{th} binary digit.
- wl is the word length in bits.
- b_{wl-1} is the location of the most significant, or highest, bit (MSB).
- b_0 is the location of the least significant, or lowest, bit (LSB).
- The binary point is shown four places to the left of the LSB. In this example, therefore, the number is said to have four fractional bits, or a fraction length of four.

Fixed-point data types can be either signed or unsigned. Signed binary fixed-point numbers are typically represented in one of three ways:

- Sign/magnitude
- One's complement
- Two's complement

Two's complement is the most common representation of signed fixed-point numbers and is the only representation used by Fixed-Point Toolbox documentation. Refer to “Two's Complement” on page 2-11 for more information.

Scaling

Fixed-point numbers can be encoded according to the scheme

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

where the slope can be expressed as

$$\text{slope} = \text{slope adjustment factor} \times 2^{\text{fixed exponent}}$$

The integer is sometimes called the *stored integer*. This is the raw binary number, in which the binary point assumed to be at the far right of the word. In Fixed-Point Toolbox documentation, the negative of the fixed exponent is often referred to as the *fraction length*.

The slope and bias together represent the scaling of the fixed-point number. In a number with zero bias, only the slope affects the scaling. A fixed-point number that is only scaled by binary point position is equivalent to a number in [Slope Bias] representation that has a bias equal to zero and a slope adjustment factor equal to one. This is referred to as binary point-only scaling or power-of-two scaling:

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

or

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

Fixed-Point Toolbox software supports both binary point-only scaling and [Slope Bias] scaling.

Note For examples of binary point-only scaling, see the Fixed-Point Toolbox Binary-Point Scaling demo.

Precision and Range

In this section...

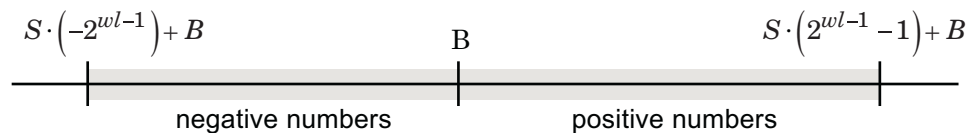
“Range” on page 2-5

“Precision” on page 2-6

Note You must pay attention to the precision and range of the fixed-point data types and scalings you choose in order to know whether rounding methods will be invoked or if overflows or underflows will occur.

Range

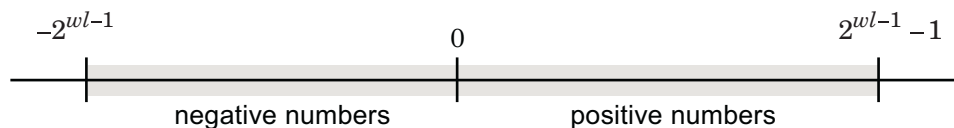
The range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for a two’s complement fixed-point number of word length wl , scaling S and bias B is illustrated below:



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

For example, in two’s complement, negative numbers must be represented as well as zero, so the maximum value is $2^{wl-1} - 1$. Because there is only one representation for zero, there are an unequal number of positive and negative numbers. This means there is a representation for -2^{wl-1} but not for 2^{wl-1} :

For slope = 1 and bias = 0:



Overflow Handling

Because a fixed-point data type represents numbers within a finite range, overflows and underflows can occur if the result of an operation is larger or smaller than the numbers in that range.

Fixed-Point Toolbox software allows you to either *saturate* or *wrap* overflows. Saturation represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used. Wrapping uses modulo arithmetic to cast an overflow back into the representable range of the data type. Refer to “Modulo Arithmetic” on page 2-10 for more information.

When you create a `fi` object, any overflows are saturated. The `OverflowMode` property of the default global `fimath` is `saturate`. You can log overflows and underflows by setting the `LoggingMode` property of the `fipref` object to `on`. Refer to “LoggingMode” for more information.

Precision

The precision of a fixed-point number is the difference between successive values representable by its data type and scaling, which is equal to the value of its least significant bit. The value of the least significant bit, and therefore the precision of the number, is determined by the number of fractional bits. A fixed-point value can be represented to within half of the precision of its data type and scaling.

For example, a fixed-point representation with four bits to the right of the binary point has a precision of 2^{-4} or 0.0625, which is the value of its least significant bit. Any number within the range of this data type and scaling can be represented to within $(2^{-4})/2$ or 0.03125, which is half the precision. This is an example of representing a number with finite precision.

Rounding Methods

When you represent numbers with finite precision, not every number in the available range can be represented exactly. If a number cannot be represented exactly by the specified data type and scaling, a rounding method is used to cast the value to a representable number. Although precision is always lost in the rounding operation, the cost of the operation and the amount of bias that is introduced depends on the rounding method itself. To provide you with

greater flexibility in the trade-off between cost and bias, Fixed-Point Toolbox software currently supports the following rounding methods:

- `ceil` rounds to the closest representable number in the direction of positive infinity.
- `convergent` rounds to the closest representable number. In the case of a tie, `convergent` rounds to the nearest even number. This is the least biased rounding method provided by the toolbox.
- `fix` rounds to the closest representable number in the direction of zero.
- `floor`, which is equivalent to two's complement truncation, rounds to the closest representable number in the direction of negative infinity.
- `nearest` rounds to the closest representable number. In the case of a tie, `nearest` rounds to the closest representable number in the direction of positive infinity. This rounding method is the default for `fi` object creation and `fi` arithmetic.
- `round` rounds to the closest representable number. In the case of a tie, the `round` method rounds:
 - Positive numbers to the closest representable number in the direction of positive infinity.
 - Negative numbers to the closest representable number in the direction of negative infinity.

Choosing a Rounding Method. Each rounding method has a set of inherent properties. Depending on the requirements of your design, these properties could make the rounding method more or less desirable to you. By knowing the requirements of your design and understanding the properties of each rounding method, you can determine which is the best fit for your needs. The most important properties to consider are:

- Cost — Independent of the hardware being used, how much processing expense does the rounding method require?
 - Low — The method requires few processing cycles.
 - Moderate — The method requires a moderate number of processing cycles.
 - High — The method requires more processing cycles.

Note The cost estimates provided here are hardware independent. Some processors have rounding modes built-in, so consider carefully the hardware you are using before calculating the true cost of each rounding mode.

- Bias — What is the expected value of the rounded values minus the original values: $E(\hat{\theta} - \theta)$?
 - $E(\hat{\theta} - \theta) < 0$ — The rounding method introduces a negative bias.
 - $E(\hat{\theta} - \theta) = 0$ — The rounding method is unbiased.
 - $E(\hat{\theta} - \theta) > 0$ — The rounding method introduces a positive bias.
- Possibility of Overflow — Does the rounding method introduce the possibility of overflow?
 - Yes — The rounded values may exceed the minimum or maximum representable value.
 - No — The rounded values will never exceed the minimum or maximum representable value.

The following table shows a comparison of the different rounding methods available in both Fixed-Point Toolbox and Simulink® Fixed Point™ products.

Fixed-Point Toolbox Rounding Method	Simulink Fixed Point Rounding Mode	Cost	Bias	Possibility of Overflow
ceil	Ceiling	Low	Large positive	Yes
convergent	Convergent	High	Unbiased	Yes
fix	Zero	Low	<ul style="list-style-type: none"> • Large positive for negative samples • Unbiased for samples with evenly distributed positive and negative values • Large negative for positive samples 	No
floor	Floor	Low	Large negative	No
nearest	Nearest	Moderate	Small positive	Yes
round	Round	High	<ul style="list-style-type: none"> • Small negative for negative samples • Unbiased for samples with evenly distributed positive and negative values • Small positive for positive samples 	Yes
N/A	Simplest (Simulink Fixed Point only)	Low	Depends on the operation	No

Arithmetic Operations

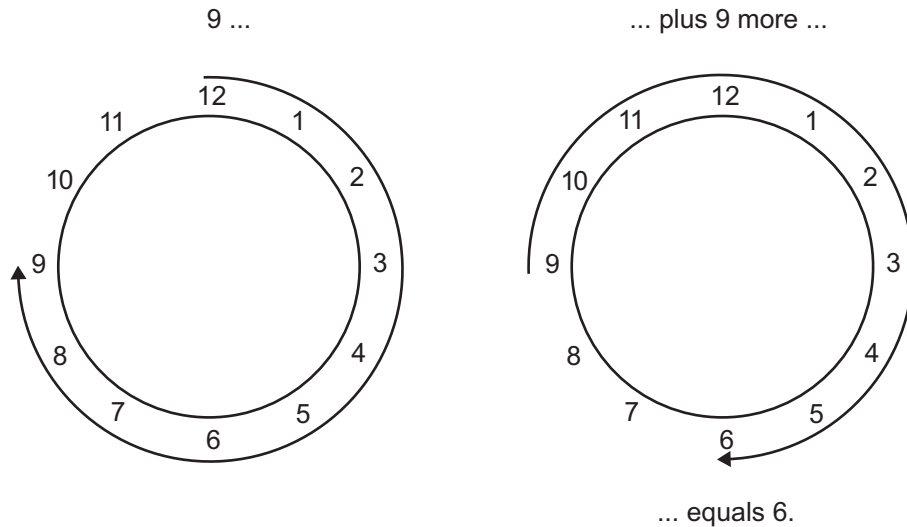
In this section...
“Modulo Arithmetic” on page 2-10
“Two’s Complement” on page 2-11
“Addition and Subtraction” on page 2-12
“Multiplication” on page 2-13
“Casts” on page 2-19

Note These sections will help you understand what data type and scaling choices result in overflows or a loss of precision.

Modulo Arithmetic

Binary math is based on modulo arithmetic. Modulo arithmetic uses only a finite set of numbers, wrapping the results of any calculations that fall outside the given set back into the set.

For example, the common everyday clock uses modulo 12 arithmetic. Numbers in this system can only be 1 through 12. Therefore, in the “clock” system, 9 plus 9 equals 6. This can be more easily visualized as a number circle:



Similarly, binary math can only use the numbers 0 and 1, and any arithmetic results that fall outside this range are wrapped “around the circle” to either 0 or 1.

Two’s Complement

Two’s complement is a way to interpret a binary number. In two’s complement, positive numbers always start with a 0 and negative numbers always start with a 1. If the leading bit of a two’s complement number is 0, the value is obtained by calculating the standard binary value of the number. If the leading bit of a two’s complement number is 1, the value is obtained by assuming that the leftmost bit is negative, and then calculating the binary value of the number. For example,

$$01 = (0 + 2^0) = 1$$

$$11 = \left((-2^1) + (2^0) \right) = (-2 + 1) = -1$$

To compute the negative of a binary number using two’s complement,

- 1 Take the one’s complement, or “flip the bits.”

2 Add a $2^{(-FL)}$ using binary math, where FL is the fraction length.

3 Discard any bits carried beyond the original word length.

For example, consider taking the negative of 11010 (-6). First, take the one's complement of the number, or flip the bits:

$$11010 \rightarrow 00101$$

Next, add a 1, wrapping all numbers to 0 or 1:

$$\begin{array}{r} 00101 \\ +1 \\ \hline 00110 \quad (6) \end{array}$$

Addition and Subtraction

The addition of fixed-point numbers requires that the binary points of the addends be aligned. The addition is then performed using binary arithmetic so that no number other than 0 or 1 is used.

For example, consider the addition of 010010.1 (18.5) with 0110.110 (6.75):

$$\begin{array}{r} 010010.1 \quad (18.5) \\ +0110.110 \quad (6.75) \\ \hline 011001.010 \quad (25.25) \end{array}$$

Fixed-point subtraction is equivalent to adding while using the two's complement value for any negative values. In subtraction, the addends must be sign-extended to match each other's length. For example, consider subtracting 0110.110 (6.75) from 010010.1 (18.5):

$$\begin{array}{r} 010010.100 \quad (18.5) \\ -0110.110 \quad (6.75) \\ \hline \end{array} \xrightarrow[\text{and sign extension}]{\text{two's complement}} \begin{array}{r} 010010.100 \quad (18.5) \\ +111001.010 \quad (-6.75) \\ \hline 1001011.110 \quad (11.75) \end{array}$$

Carry bit is discarded

The default global `fimath` has a value of 1 (true) for the `CastBeforeSum` property. This casts addends to the sum data type before addition. Therefore, no further shifting is necessary during the addition to line up the binary points.

If `CastBeforeSum` has a value of 0 (false), the addends are added with full precision maintained. After the addition the sum is then quantized.

Multiplication

The multiplication of two's complement fixed-point numbers is directly analogous to regular decimal multiplication, with the exception that the intermediate results must be sign-extended so that their left sides align before you add them together.

For example, consider the multiplication of 10.11 (-1.25) with 011 (3):

$$\begin{array}{r}
 10.11 \text{ (-1.25)} \\
 \quad 011 \text{ (3)} \\
 \hline
 11011 \\
 \quad 1011 \\
 \hline
 1100.01 \text{ (-3.75)}
 \end{array}$$

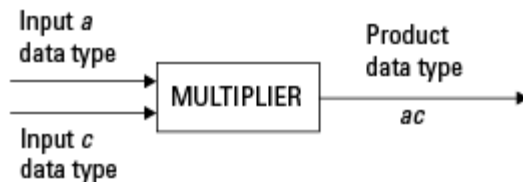
The extra 1 is the result of necessary sign extension.

The number of fractional bits of the result is the sum of the number of fractional bits of the factors.

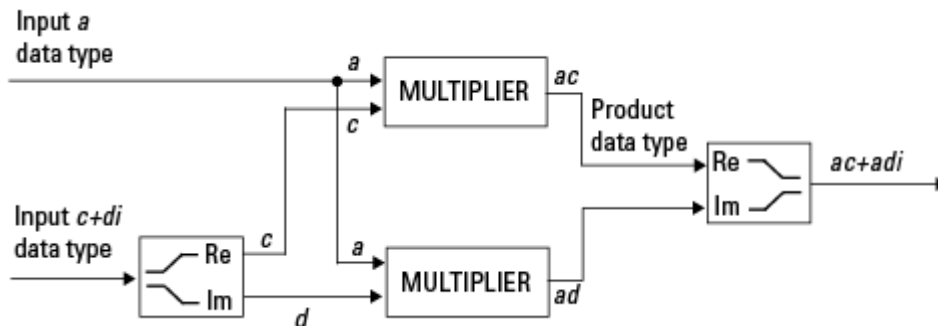
Multiplication Data Types

The following diagrams show the data types used for fixed-point multiplication using Fixed-Point Toolbox software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex multiplication.

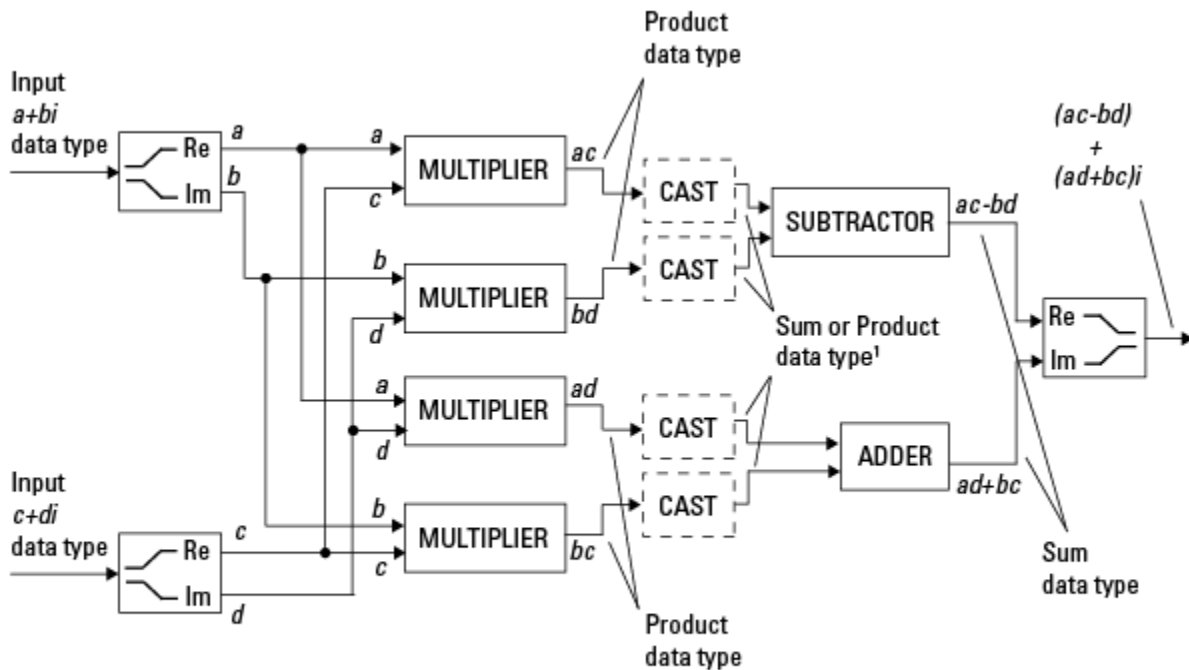
Real-Real Multiplication. The following diagram shows the data types used by the toolbox in the multiplication of two real numbers. The software returns the output of this operation in the product data type, which is governed by the fimath object ProductMode property.



Real-Complex Multiplication. The following diagram shows the data types used by the toolbox in the multiplication of a real and a complex fixed-point number. Real-complex and complex-real multiplication are equivalent. The software returns the output of this operation in the product data type, which is governed by the fimath object ProductMode property:



Complex-Complex Multiplication. The following diagram shows the multiplication of two complex fixed-point numbers. Note that the software returns the output of this operation in the sum data type, which is governed by the fimath object SumMode property. The intermediate product data type is determined by the fimath object ProductMode property.



¹ Sum data type if CastBeforeSum is true,
Product data type if CastBeforeSum is false

When the fimath object CastBeforeSum property is true, the casts to the sum data type are present after the multipliers in the preceding diagram. In C code, this is equivalent to

```
acc=ac;
acc-=bd;
```

for the subtractor, and

```
acc=ad;
acc+=bc;
```

for the adder, where *acc* is the accumulator. When the CastBeforeSum property is false, the casts are not present, and the data remains in the product data type before the subtraction and addition operations.

Multiplication with fimath

In the following examples, let

```
F = fimath('ProductMode','FullPrecision',...
'SumMode','FullPrecision')
T1 = numerictype('WordLength',24,'FractionLength',20)
T2 = numerictype('WordLength',16,'FractionLength',10)
```

Real*Real. Notice that the word length and fraction length of the result z are equal to the sum of the word lengths and fraction lengths, respectively, of the multiplicands. This is because the `fimath` `SumMode` and `ProductMode` properties are set to `FullPrecision`:

```
P = fipref;
P.FimathDisplay = 'none';
x = fi(5, T1, F)
```

```
x =
```

```
5
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 24
FractionLength: 20
```

```
y = fi(10, T2, F)
```

```
y =
```

```
10
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 10
```

```
z = x*y
```


z =

50

```

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
          WordLength: 40
        FractionLength: 30

```

Real*Complex. Notice that the word length and fraction length of the result z are equal to the sum of the word lengths and fraction lengths, respectively, of the multiplicands. This is because the `fimath` `SumMode` and `ProductMode` properties are set to `FullPrecision`:

x = `fi(5,T1,F)`

x =

5

```

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
          WordLength: 24
        FractionLength: 20

```

y = `fi(10+2i,T2,F)`

y =

10.0000 + 2.0000i

```

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
          WordLength: 16
        FractionLength: 10

```

$z = x*y$

$z =$

50.0000 +10.0000i

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 40
FractionLength: 30

Complex*Complex. Complex-complex multiplication involves an addition as well as multiplication, so the word length of the full-precision result has one more bit than the sum of the word lengths of the multiplicands:

$x = \text{fi}(5+6i, T1, F)$

$x =$

5.0000 + 6.0000i

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 24
FractionLength: 20

$y = \text{fi}(10+2i, T2, F)$

$y =$

10.0000 + 2.0000i

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 10

```
z = x*y
```

```
z =
```

```
38.0000 +70.0000i
```

```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 41  
FractionLength: 30
```

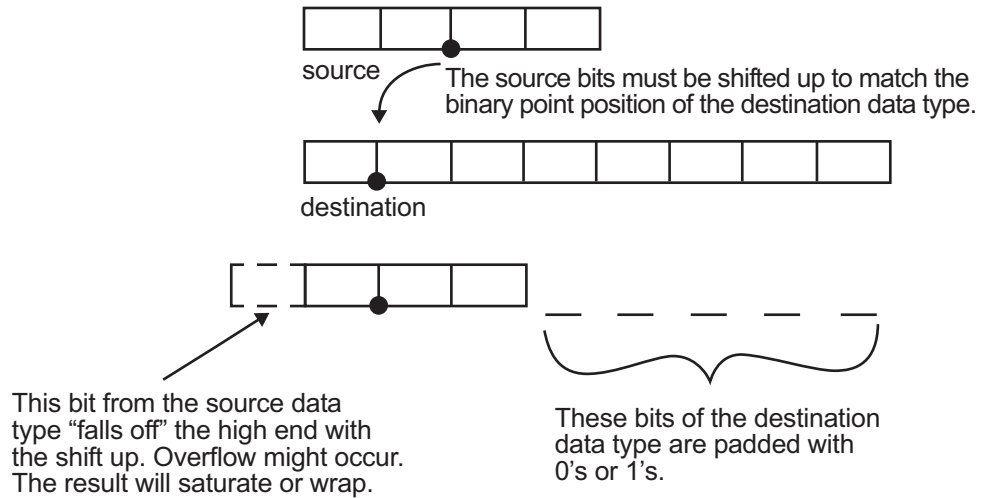
Casts

The `fimath` object allows you to specify the data type and scaling of intermediate sums and products with the `SumMode` and `ProductMode` properties. It is important to keep in mind the ramifications of each cast when you set the `SumMode` and `ProductMode` properties. Depending upon the data types you select, overflow and/or rounding might occur. The following two examples demonstrate cases where overflow and rounding can occur.

Note For more examples of casting, see “Casting fi Objects” on page 3-12.

Casting from a Shorter Data Type to a Longer Data Type

Consider the cast of a nonzero number, represented by a 4-bit data type with two fractional bits, to an 8-bit data type with seven fractional bits:



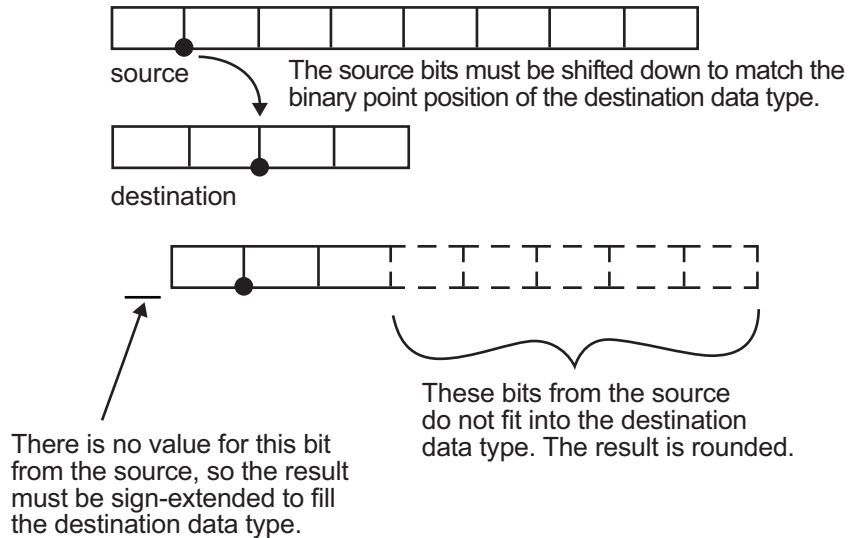
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 0's or 1's:

- If overflow does not occur, the empty bits are padded with 0's.
- If wrapping occurs, the empty bits are padded with 0's.
- If saturation occurs,
 - The empty bits of a positive number are padded with 1's.
 - The empty bits of a negative number are padded with 0's.

You can see that even with a cast from a shorter data type to a longer data type, overflow can still occur. This can happen when the integer length of the source data type (in this case two) is longer than the integer length of the destination data type (in this case one). Similarly, rounding might be necessary even when casting from a shorter data type to a longer data type, if the destination data type and scaling has fewer fractional bits than the source.

Casting from a Longer Data Type to a Shorter Data Type

Consider the cast of a nonzero number, represented by an 8-bit data type with seven fractional bits, to a 4-bit data type with two fractional bits:



As the diagram shows, the source bits are shifted down so that the binary point matches the destination binary point position. There is no value for the highest bit from the source, so sign extension is used to fill the integer portion of the destination data type. The bottom five bits of the source do not fit into the fraction length of the destination. Therefore, precision can be lost as the result is rounded.

In this case, even though the cast is from a longer data type to a shorter data type, all the integer bits are maintained. Conversely, full precision can be maintained even if you cast to a shorter data type, as long as the fraction length of the destination data type is the same length or longer than the fraction length of the source data type. In that case, however, bits are lost from the high end of the result and overflow can occur.

The worst case occurs when both the integer length and the fraction length of the destination data type are shorter than those of the source data type and scaling. In that case, both overflow and a loss of precision can occur.

fi Objects Compared to C Integer Data Types

In this section...
“Integer Data Types” on page 2-22
“Unary Conversions” on page 2-24
“Binary Conversions” on page 2-25
“Overflow Handling” on page 2-28

Note The sections in this topic compare the `fi` object with fixed-point data types and operations in C. In these sections, the information on ANSI C is adapted from Samuel P. Harbison and Guy L. Steele Jr., *C: A Reference Manual*, 3rd ed., Prentice Hall, 1991.

Integer Data Types

This section compares the numerical range of `fi` integer data types to the minimum numerical range of C integer data types, assuming a two’s complement representation.

C Integer Data Types

Many C compilers support a two’s complement representation of signed integer data types. The following table shows the minimum ranges of C integer data types using a two’s complement representation. The integer ranges can be larger than or equal to those shown, but cannot be smaller. The range of a `long` must be larger than or equal to the range of an `int`, which must be larger than or equal to the range of a `short`.

In the two’s complement representation, a signed integer with n bits has a range from -2^{n-1} to $2^{n-1} - 1$, inclusive. An unsigned integer with n bits has a range from 0 to $2^n - 1$, inclusive. The negative side of the range has one more value than the positive side, and zero is represented uniquely.

Integer Type	Minimum	Maximum
signed char	-128	127
unsigned char	0	255
short int	-32,768	32,767
unsigned short	0	65,535
int	-32,768	32,767
unsigned int	0	65,535
long int	-2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295

fi Integer Data Types

The following table lists the numerical ranges of the integer data types of the `fi` object, in particular those equivalent to the C integer data types. The ranges are large enough to accommodate the two's complement representation, which is the only signed binary encoding technique supported by Fixed-Point Toolbox software.

Constructor	Signed	Word Length	Fraction Length	Minimum	Maximum	Closest ANSI C Equivalent
<code>fi(x,1,n,0)</code>	Yes	n (2 to 65,535)	0	-2^{n-1}	$2^{n-1} - 1$	N/A
<code>fi(x,0,n,0)</code>	No	n (2 to 65,535)	0	0	$2^n - 1$	N/A
<code>fi(x,1,8,0)</code>	Yes	8	0	-128	127	signed char
<code>fi(x,0,8,0)</code>	No	8	0	0	255	unsigned char
<code>fi(x,1,16,0)</code>	Yes	16	0	-32,768	32,767	short int
<code>fi(x,0,16,0)</code>	No	16	0	0	65,535	unsigned short

Constructor	Signed	Word Length	Fraction Length	Minimum	Maximum	Closest ANSI C Equivalent
<code>fi(x,1,32,0)</code>	Yes	32	0	-2,147,483,648	2,147,483,647	long int
<code>fi(x,0,32,0)</code>	No	32	0	0	4,294,967,295	unsigned long

Unary Conversions

Unary conversions dictate whether and how a single operand is converted before an operation is performed. This section discusses unary conversions in ANSI C and of `fi` objects.

ANSI C Usual Unary Conversions

Unary conversions in ANSI C are automatically applied to the operands of the unary `!`, `-`, `~`, and `*` operators, and of the binary `<<` and `>>` operators, according to the following table:

Original Operand Type	ANSI C Conversion
char or short	int
unsigned char or unsigned short	int or unsigned int ¹
float	float
Array of T	Pointer to T
Function returning T	Pointer to function returning T

¹If type `int` cannot represent all the values of the original data type without overflow, the converted type is `unsigned int`.

fi Usual Unary Conversions

The following table shows the fi unary conversions:

C Operator	fi Equivalent	fi Conversion
!x	$\sim x = \text{not}(x)$	Result is logical.
~x	bitcmp(x)	Result is same numeric type as operand.
*x	No equivalent	N/A
x<<n	bitshift(x,n) positive n	Result is same numeric type as operand. Round mode is always floor. Overflow mode is obeyed. 0-valued bits are shifted in on the right.
x>>n	bitshift(x,-n)	Result is same numeric type as operand. Round mode is always floor. Overflow mode is obeyed. 0-valued bits are shifted in on the left if the operand is unsigned or signed and positive. 1-valued bits are shifted in on the left if the operand is signed and negative.
+x	+x	Result is same numeric type as operand.
-x	-x	Result is same numeric type as operand. Overflow mode is obeyed. For example, overflow might occur when you negate an unsigned fi or the most negative value of a signed fi.

Binary Conversions

This section describes the conversions that occur when the operands of a binary operator are different data types.

ANSI C Usual Binary Conversions

In ANSI C, operands of a binary operator must be of the same type. If they are different, one is converted to the type of the other according to the first applicable conversion in the following table:

Type of One Operand	Type of Other Operand	ANSI C Conversion
long double	Any	long double
double	Any	double
float	Any	float
unsigned long	Any	unsigned long
long	unsigned	long or unsigned long ¹
long	int	long
unsigned	int or unsigned	unsigned
int	int	int

¹Type long is only used if it can represent all values of type unsigned.

fi Usual Binary Conversions

When one of the operands of a binary operator (+, -, *, .*) is a `fi` object and the other is a MATLAB built-in numeric type, then the non-`fi` operand is converted to a `fi` object before the operation is performed, according to the following table:

Type of One Operand	Type of Other Operand	Properties of Other Operand After Conversion to a fi Object
<code>fi</code>	double or single	<ul style="list-style-type: none"> • Signed = same as the original <code>fi</code> operand • WordLength = same as the original <code>fi</code> operand • FractionLength = set to best precision possible
<code>fi</code>	<code>int8</code>	<ul style="list-style-type: none"> • Signed = 1 • WordLength = 8 • FractionLength = 0

Type of One Operand	Type of Other Operand	Properties of Other Operand After Conversion to a fi Object
fi	uint8	<ul style="list-style-type: none"> • Signed = 0 • WordLength = 8 • FractionLength = 0
fi	int16	<ul style="list-style-type: none"> • Signed = 1 • WordLength = 16 • FractionLength = 0
fi	uint16	<ul style="list-style-type: none"> • Signed = 0 • WordLength = 16 • FractionLength = 0
fi	int32	<ul style="list-style-type: none"> • Signed = 1 • WordLength = 32 • FractionLength = 0
fi	uint32	<ul style="list-style-type: none"> • Signed = 0 • WordLength = 32 • FractionLength = 0
fi	int64	<ul style="list-style-type: none"> • Signed = 1 • WordLength = 64 • FractionLength = 0
fi	uint64	<ul style="list-style-type: none"> • Signed = 0 • WordLength = 64 • FractionLength = 0

Overflow Handling

The following sections compare how ANSI C and Fixed-Point Toolbox software handle overflows.

ANSI C Overflow Handling

In ANSI C, the result of signed integer operations is whatever value is produced by the machine instruction used to implement the operation. Therefore, ANSI C has no rules for handling signed integer overflow.

The results of unsigned integer overflows wrap in ANSI C.

fi Overflow Handling

Addition and multiplication with `fi` objects yield results that can be exactly represented by a `fi` object, up to word lengths of 65,535 bits or the available memory on your machine. This is not true of division, however, because many ratios result in infinite binary expressions. You can perform division with `fi` objects using the `divide` function, which requires you to explicitly specify the numeric type of the result.

The conditions under which a `fi` object overflows and the results then produced are determined by the associated `fimath` object. You can specify certain overflow characteristics separately for sums (including differences) and products. Refer to the following table:

fimath Object Properties Related to Overflow Handling	Property Value	Description
OverflowMode	'saturate'	Overflows are saturated to the maximum or minimum value in the range.
	'wrap'	Overflows wrap using modulo arithmetic if unsigned, two's complement wrap if signed.

fimath Object Properties Related to Overflow Handling	Property Value	Description
ProductMode	'FullPrecision'	<p>Full-precision results are kept. Overflow does not occur. An error is thrown if the resulting word length is greater than MaxProductWordLength.</p> <p>The rules for computing the resulting product word and fraction lengths are given in “ProductMode” in the Property Reference.</p>
	'KeepLSB'	<p>The least significant bits of the product are kept. Full precision is kept, but overflow is possible. This behavior models the C language integer operations.</p> <p>The resulting word length is determined by the ProductWordLength property. If ProductWordLength is greater than is necessary for the full-precision product, then the result is stored in the least significant bits. If ProductWordLength is less than is necessary for the full-precision product, then overflow occurs.</p> <p>The rule for computing the resulting product fraction length is given in “ProductMode” in the Property Reference.</p>

fimath Object Properties Related to Overflow Handling	Property Value	Description
	'KeepMSB'	<p>The most significant bits of the product are kept. Overflow is prevented, but precision may be lost.</p> <p>The resulting word length is determined by the ProductWordLength property. If ProductWordLength is greater than is necessary for the full-precision product, then the result is stored in the most significant bits. If ProductWordLength is less than is necessary for the full-precision product, then rounding occurs.</p> <p>The rule for computing the resulting product fraction length is given in “ProductMode” in the Property Reference.</p>
	'SpecifyPrecision'	<p>You can specify both the word length and the fraction length of the resulting product.</p>
ProductWordLength	Positive integer	<p>The word length of product results when ProductMode is 'KeepLSB', 'KeepMSB', or 'SpecifyPrecision'.</p>
MaxProductWordLength	Positive integer	<p>The maximum product word length allowed when ProductMode is 'FullPrecision'. The default is 128 bits. The maximum is 65,535 bits. This property can help ensure that your simulation does not exceed your hardware requirements.</p>
ProductFractionLength	Integer	<p>The fraction length of product results when ProductMode is 'Specify Precision'.</p>

fimath Object Properties Related to Overflow Handling	Property Value	Description
SumMode	'FullPrecision'	<p>Full-precision results are kept. Overflow does not occur. An error is thrown if the resulting word length is greater than MaxSumWordLength.</p> <p>The rules for computing the resulting sum word and fraction lengths are given in “SumMode” in the Property Reference.</p>
	'KeepLSB'	<p>The least significant bits of the sum are kept. Full precision is kept, but overflow is possible. This behavior models the C language integer operations.</p> <p>The resulting word length is determined by the SumWordLength property. If SumWordLength is greater than is necessary for the full-precision sum, then the result is stored in the least significant bits. If SumWordLength is less than is necessary for the full-precision sum, then overflow occurs.</p> <p>The rule for computing the resulting sum fraction length is given in “SumMode” in the Property Reference.</p>
	'KeepMSB'	<p>The most significant bits of the sum are kept. Overflow is prevented, but precision may be lost.</p> <p>The resulting word length is determined by the SumWordLength property. If SumWordLength is greater than is necessary for the full-precision sum, then the result is stored in the most significant bits. If SumWordLength is less than is necessary for the full-precision sum, then rounding occurs.</p>

fimath Object Properties Related to Overflow Handling	Property Value	Description
		The rule for computing the resulting sum fraction length is given in “SumMode” in the Property Reference.
	'SpecifyPrecision'	You can specify both the word length and the fraction length of the resulting sum.
SumWordLength	Positive integer	The word length of sum results when SumMode is 'KeepLSB', 'KeepMSB', or 'SpecifyPrecision'.
MaxSumWordLength	Positive integer	The maximum sum word length allowed when SumMode is 'FullPrecision'. The default is 128 bits. The maximum is 65,535 bits. This property can help ensure that your simulation does not exceed your hardware requirements.
SumFractionLength	Integer	The fraction length of sum results when SumMode is 'SpecifyPrecision'.

Working with fi Objects

- “Constructing fi Objects” on page 3-2
- “Casting fi Objects” on page 3-12
- “fi Object Properties” on page 3-17
- “fi Object Functions” on page 3-24

Constructing fi Objects

In this section...
“fi Object Syntaxes” on page 3-2
“Examples of Constructing fi Objects” on page 3-3

fi Object Syntaxes

You can create `fi` objects using Fixed-Point Toolbox software in any of the following ways:

- You can use the `fi` constructor function to create a new `fi` object.
- You can use the `sfi` constructor function to create a new signed `fi` object.
- You can use the `ufi` constructor function to create a new unsigned `fi` object.
- You can use any of the `fi` constructor functions to copy an existing `fi` object.

To get started, type

```
a = fi(0)
```

to create a `fi` object with the default data type and a value of 0.

```
a =
```

```
0
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 16  
      FractionLength: 15
```

This constructor syntax creates a signed `fi` object with a value of 0, word length of 16 bits, and fraction length of 15 bits. Because you did not specify any `fimath` object properties in the `fi` constructor, the resulting `fi` object `a` associates itself with the global `fimath`.

To see all of the `fi`, `sfi`, and `ufi` constructor syntaxes, refer to the respective reference pages.

Note For information on the display format of `fi` objects, refer to “Display Settings” on page 1-7.

Examples of Constructing fi Objects

The following examples show you several different ways to construct `fi` objects. For other, more basic examples of constructing `fi` objects, see the Examples section of the following constructor function reference pages:

- `fi`
- `sfi`
- `ufi`

Constructing a fi Object with Property Name/Property Value Pairs

You can use property name/property value pairs to set `fi` and `fimath` object properties when you create the `fi` object:

```
a = fi(pi, 'roundmode', 'floor', 'overflowmode', 'wrap')
```

```
a =
```

```
3.1415
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 16  
      FractionLength: 13
```

```
      RoundMode: floor  
      OverflowMode: wrap  
      ProductMode: FullPrecision  
      MaxProductWordLength: 128  
      SumMode: FullPrecision
```

```
MaxSumWordLength: 128
```

You do not have to specify every `fi` object property in the `fi` constructor. The `fi` object inherits all unspecified `fi` object properties from the global `fi`.

- If you specify at least one `fi` object property in the `fi` constructor, the `fi` object will have a local `fi` object. The `fi` object inherits the remaining unspecified `fi` object properties from the global `fi`.
- If you do not specify any `fi` object properties in the `fi` object constructor, the `fi` object associates itself with the global `fi`. See “Working with the Global `fi`” on page 4-6 for more information.

Constructing a fi Object Using a numeric type Object

You can use a numeric type object to define a `fi` object:

```
T = numericType
```

```
T =
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 16  
      FractionLength: 15
```

```
a = fi(pi, T)
```

```
a =
```

```
1.0000
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 16  
      FractionLength: 15
```

You can also use a `fimath` object with a `numericType` object to define a `fi` object:

```
F = fimath('RoundMode', 'nearest',...
'OverflowMode', 'saturate',...
'ProductMode', 'FullPrecision',...
'MaxProductWordLength', 128,...
'SumMode', 'FullPrecision',...
'MaxSumWordLength', 128)
```

```
F =
```

```
          RoundMode: nearest
          OverflowMode: saturate
          ProductMode: FullPrecision
MaxProductWordLength: 128
          SumMode: FullPrecision
          MaxSumWordLength: 128
```

```
a = fi(pi, T, F)
```

```
a =
```

```
1.0000
```

```
          DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
          WordLength: 16
          FractionLength: 15
```

```
          RoundMode: nearest
          OverflowMode: saturate
          ProductMode: FullPrecision
MaxProductWordLength: 128
          SumMode: FullPrecision
          MaxSumWordLength: 128
```

Note The syntax `a = fi(pi,T,F)` is equivalent to `a = fi(pi,F,T)`. You can use both statements to define a `fi` object using a `fimath` object and a `numericType` object.

Constructing a fi Object Using a fimath Object

You can create a `fi` object using a specific `fimath` object. When you do so, a local `fimath` object is assigned to the `fi` object you create. If you do not specify any `numericType` object properties, the word length of the `fi` object defaults to 16 bits. The fraction length is determined by best precision scaling:

```
F = fimath('RoundMode', 'nearest',...
'OverflowMode', 'saturate',...
'ProductMode', 'FullPrecision',...
'MaxProductWordLength', 128,...
'SumMode', 'FullPrecision',...
'MaxSumWordLength', 128)
```

```
F =
```

```
          RoundMode: nearest
      OverflowMode: saturate
          ProductMode: FullPrecision
MaxProductWordLength: 128
              SumMode: FullPrecision
      MaxSumWordLength: 128
```

```
F.OverflowMode = 'wrap'
```

```
F =
```

```
          RoundMode: nearest
      OverflowMode: wrap
          ProductMode: FullPrecision
MaxProductWordLength: 128
              SumMode: FullPrecision
```

```
MaxSumWordLength: 128

a = fi(pi, F)

a =

    3.1416

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 16
    FractionLength: 13

    RoundMode: nearest
    OverflowMode: wrap
    ProductMode: FullPrecision
MaxProductWordLength: 128
    SumMode: FullPrecision
MaxSumWordLength: 128
```

You can also create fi objects using a fimath object while specifying various numericity properties at creation time:

```
b = fi(pi, 0, F)

b =

    3.1416

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Unsigned
    WordLength: 16
    FractionLength: 14

    RoundMode: nearest
    OverflowMode: wrap
    ProductMode: FullPrecision
MaxProductWordLength: 128
    SumMode: FullPrecision
MaxSumWordLength: 128
```

```
c = fi(pi, 0, 8, F)
```

```
c =
```

```
3.1406
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Unsigned  
    WordLength: 8  
    FractionLength: 6
```

```
    RoundMode: nearest  
    OverflowMode: wrap  
    ProductMode: FullPrecision  
MaxProductWordLength: 128  
    SumMode: FullPrecision  
MaxSumWordLength: 128
```

```
d = fi(pi, 0, 8, 6, F)
```

```
d =
```

```
3.1406
```

```
    DataTypeMode: Fixed-point: binary point scaling  
    Signedness: Unsigned  
    WordLength: 8  
    FractionLength: 6
```

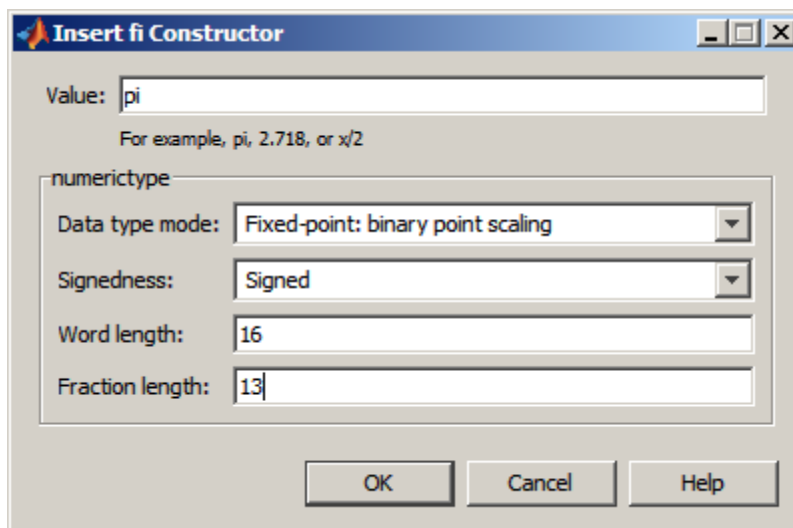
```
    RoundMode: nearest  
    OverflowMode: wrap  
    ProductMode: FullPrecision  
MaxProductWordLength: 128  
    SumMode: FullPrecision  
MaxSumWordLength: 128
```


Building fi Object Constructors in a GUI

When you are working with files in MATLAB, you can build your `fi` object constructors using the **Insert fi Constructor** dialog box. After specifying the value and properties of the `fi` object in the dialog box, you can insert the prepopulated `fi` object constructor string at a specific location in your file.

For example, to create a signed `fi` object with a value of `pi`, a word length of 16 bits and a fraction length of 13 bits, perform the following steps:

- 1 Open the **Insert fi Constructor** dialog box by selecting **Tools > Fixed-Point Toolbox > Insert fi Constructor** from the editor menu.
- 2 Use the edit boxes and drop-down menus to specify the following properties of the `fi` object:
 - **Value** = `pi`
 - **Data type mode** = Fixed-point: binary point scaling
 - **Signedness** = Signed
 - **Word length** = 16
 - **Fraction length** = 13



- 3** To insert the `fi` object constructor string in your file, place your cursor at the desired location in the file, and click **OK** on the **Insert fi Constructor** dialog box. Clicking **OK** closes the **Insert fi Constructor** dialog box and automatically populates the `fi` object constructor string in your file:

```
7 fi(pi, 1, 16, 13)
```

Determining Property Precedence

The value of a property is taken from the last time it is set. For example, create a `numerictype` object with a value of `true` for the `Signed` property and a `FractionLength` of 14:

```
T = numerictype('Signed', true, 'FractionLength', 14)
```

```
T =
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 14
```

Now, create the following `fi` object in which you specify the `numerictype` property *after* the `Signed` property, so that the resulting `fi` object is signed:

```
a = fi(pi, 'Signed', false, 'numerictype', T)
```

```
a =
```

```
1.9999
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 16
FractionLength: 14
```

Contrast the `fi` object in this code sample with the `fi` object in the following code sample. The `numerictype` property in the following code sample is specified *before* the `Signed` property, so the resulting `fi` object is unsigned:

```
b = fi(pi, 'numerictype', T, 'Signed', false)

b =

    3.1416

        DataTypeMode: Fixed-point: binary point scaling
           Signedness: Unsigned
          WordLength: 16
    FractionLength: 14
```

Copying a fi Object

To copy a `fi` object, simply use assignment, as in the following example:

```
a = fi(pi)

a =

    3.1416

        DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
          WordLength: 16
    FractionLength: 13

b = a

b =

    3.1416

        DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
          WordLength: 16
    FractionLength: 13
```

Casting fi Objects

In this section...
“Overwriting by Assignment” on page 3-12
“Ways to Cast with MATLAB Software” on page 3-12

Overwriting by Assignment

Because MATLAB software does not have type declarations, an assignment like `A = B` replaces the type and content of `A` with the type and content of `B`. If `A` does not exist at the time of the assignment, MATLAB creates the variable `A` and assigns it the same type and value as `B`. Such assignment happens with all types in MATLAB—objects and built-in types alike—including `fi`, `double`, `single`, `int8`, `uint8`, `int16`, etc.

For example, the following code overwrites the value and `int8` type of `A` with the value and `int16` type of `B`:

```
A = int8(0);
B = int16(32767);
A = B

A =

    32767

class(A)

ans =

    int16
```

Ways to Cast with MATLAB Software

You may find it useful to cast data into another type—for example, when you are casting data from an accumulator to memory. There are several ways to cast data in MATLAB. The following sections provide examples of three different methods:

- Casting by Subscripted Assignment
- Casting by Conversion Function
- Casting with the Fixed-Point Toolbox `reinterpretcast` Function

Casting by Subscripted Assignment

The following subscripted assignment statement retains the type of `A` and saturates the value of `B` to an `int8`:

```
A = int8(0);  
B = int16(32767);  
A(:) = B
```

```
A =
```

```
    127
```

```
class(A)
```

```
ans =
```

```
int8
```

The same is true for `fi` objects:

```
fipref('NumericTypeDisplay', 'short');  
A = fi(0, true, 8, 0);  
B = fi(32767, true, 16, 0);  
A(:) = B
```

```
A =
```

```
    127
```

```
    s8,0
```

Note For more information on subscripted assignments, see the `subsasgn` function.

Casting by Conversion Function

You can convert from one data type to another by using a conversion function. In this example, A does not have to be predefined because it is overwritten.

```
B = int16(32767);  
A = int8(B)
```

```
A =
```

```
    127
```

```
class(A)
```

```
ans =
```

```
int8
```

The same is true for fi objects:

```
B = fi(32767, true, 16, 0)  
A = fi(B, 1, 8, 0)
```

```
B =
```

```
    32767  
s16,0
```

```
A =
```

```
    127  
s8,0
```

Using a numerictype Object in the fi Conversion Function. Often a specific numerictype is used in many places, and it is convenient to predefine numerictype objects for use in the conversion functions. Predefining these objects is a good practice because it also puts the data type specification in one place.

```
T8 = numerictype(1,8,0)
```

```
T8 =
```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 8
        FractionLength: 0

T16 = numericity(1,16,0)

T16 =

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 0

B = fi(32767,T16)

B =

        32767
        s16,0

A = fi(B, T8)

A =

        127
        s8,0

```

Casting with the reinterpretcast Function

You can convert fixed-point and built-in data types without changing the underlying data. The Fixed-Point Toolbox `reinterpretcast` function performs this type of conversion.

In the following example, `B` is an unsigned `fi` object with a word length of 8 bits and a fraction length of 5 bits. The `reinterpretcast` function converts `B` into a signed `fi` object `A` with a word length of 8 bits and a fraction length of 1

bit. The real-world values of A and B differ, but their binary representations are the same.

```
B = fi([pi/4 1 pi/2 4], false, 8, 5)
T = numericType(true, 8, 1);
A = reinterpretcast(B, T)
```

```
B =
```

```
0.7813    1.0000    1.5625    4.0000
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Unsigned
      WordLength: 8
      FractionLength: 5
```

```
A =
```

```
12.5000    16.0000    25.0000   -64.0000
```

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 8
      FractionLength: 1
```

To verify that the underlying data has not changed, compare the binary representations of A and B:

```
binary_B = bin(B)
binary_A = bin(A)
```

```
binary_A =
```

```
00011001    00100000    00110010    10000000
```

```
binary_B =
```

```
00011001    00100000    00110010    10000000
```


fi Object Properties

In this section...

- “Data Properties” on page 3-17
- “fimath Properties” on page 3-17
- “numericity Properties” on page 3-19
- “Setting fi Object Properties” on page 3-20

Data Properties

The data properties of a `fi` object are always writable.

- `bin` — Stored integer value of a `fi` object in binary
- `data` — Numerical real-world value of a `fi` object
- `dec` — Stored integer value of a `fi` object in decimal
- `double` — Real-world value of a `fi` object, stored as a MATLAB `double` data type
- `hex` — Stored integer value of a `fi` object in hexadecimal
- `int` — Stored integer value of a `fi` object, stored in a built-in MATLAB integer data type. You can also use `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, and `uint64` to get the stored integer value of a `fi` object in these formats
- `oct` — Stored integer value of a `fi` object in octal

To learn more about these properties, see “fi Object Properties” in the Fixed-Point Toolbox Reference.

fimath Properties

In general, the `fimath` properties associated with `fi` objects depend on how you create the `fi` object:

- When you specify one or more `fimath` object properties in the `fi` constructor, the resulting `fi` object has a local `fimath` object.

- When you do not specify any `fi` object properties in the `fi` constructor, the resulting `fi` object associates itself with the global `fi` object. `fi` objects created in this way do not have a local `fi` object.

To determine whether a `fi` object has a local `fi` object, use the `isfimathlocal` function.

The `fi` object properties associated with `fi` objects determine how fixed-point arithmetic is performed. These `fi` object properties can come from a local `fi` object or from the global `fi` object. To learn more about the relationship between `fi` objects and the global `fi` object in fixed-point arithmetic, see “`fi` Object Rules for Fixed-Point Arithmetic” on page 4-16.

The following `fi` object properties are, by transitivity, also properties of the `fi` object. You can set these properties for individual `fi` objects or, for multiple `fi` objects, by configuring the global `fi` object. The following `fi` object properties are always writable.

- `CastBeforeSum` — Whether both operands are cast to the sum data type before addition

Note This property is hidden when the `SumMode` is set to `FullPrecision`.

- `MaxProductWordLength` — Maximum allowable word length for the product data type
- `MaxSumWordLength` — Maximum allowable word length for the sum data type
- `OverflowMode` — Overflow mode
- `ProductBias` — Bias of the product data type
- `ProductFixedExponent` — Fixed exponent of the product data type
- `ProductFractionLength` — Fraction length, in bits, of the product data type
- `ProductMode` — Defines how the product data type is determined
- `ProductSlope` — Slope of the product data type

- `ProductSlopeAdjustmentFactor` — Slope adjustment factor of the product data type
- `ProductWordLength` — Word length, in bits, of the product data type
- `RoundMode` — Rounding mode
- `SumBias` — Bias of the sum data type
- `SumFixedExponent` — Fixed exponent of the sum data type
- `SumFractionLength` — Fraction length, in bits, of the sum data type
- `SumMode` — Defines how the sum data type is determined
- `SumSlope` — Slope of the sum data type
- `SumSlopeAdjustmentFactor` — Slope adjustment factor of the sum data type
- `SumWordLength` — The word length, in bits, of the sum data type

To learn more about these properties, see the “`fimath` Object Properties” in the Fixed-Point Toolbox Reference.

To learn more about the global `fimath`, see “Working with the Global `fimath`” on page 4-6.

numericType Properties

When you create a `fi` object, a `numericType` object is also automatically created as a property of the `fi` object:

`numericType` — Object containing all the data type information of a `fi` object, Simulink signal or model parameter

The following `numericType` properties are, by transitivity, also properties of a `fi` object. The following properties of the `numericType` object become read only after you create the `fi` object. However, you can create a copy of a `fi` object with new values specified for the `numericType` properties:

- `Bias` — Bias of a `fi` object
- `DataType` — Data type category associated with a `fi` object

- `DataTypeMode` — Data type and scaling mode of a `fi` object
- `FixedExponent` — Fixed-point exponent associated with a `fi` object
- `FractionLength` — Fraction length of the stored integer value of a `fi` object in bits
- `Scaling` — Fixed-point scaling mode of a `fi` object
- `Signed` — Whether a `fi` object is signed or unsigned
- `Signedness` — Whether a `fi` object is signed or unsigned

Note `numericType` objects can have a `Signedness` of `Auto`, but all `fi` objects must be `Signed` or `Unsigned`. If a `numericType` object with `Auto Signedness` is used to create a `fi` object, the `Signedness` property of the `fi` object automatically defaults to `Signed`.

- `Slope` — Slope associated with a `fi` object
- `SlopeAdjustmentFactor` — Slope adjustment associated with a `fi` object
- `WordLength` — Word length of the stored integer value of a `fi` object in bits

For further details on these properties, see the Property Reference.

There are two ways to specify properties for `fi` objects in Fixed-Point Toolbox software. Refer to the following sections:

- “Setting Fixed-Point Properties at Object Creation” on page 3-21
- “Using Direct Property Referencing with `fi`” on page 3-21

Setting `fi` Object Properties

You can set `fi` object properties in two ways:

- Setting the properties when you create the object
- Using direct property referencing

Setting Fixed-Point Properties at Object Creation

You can set properties of `fi` objects at the time of object creation by including properties after the arguments of the `fi` constructor function. For example, to set the overflow mode to `wrap` and the rounding mode to `convergent`,

```
a = fi(pi, 'OverflowMode', 'wrap', 'RoundMode', 'convergent')
```

```
a =
```

```
3.1416
```

```
        DataTypeMode: Fixed-point: binary point scaling  
        Signedness: Signed  
        WordLength: 16  
        FractionLength: 13
```

```
        RoundMode: convergent  
        OverflowMode: wrap  
        ProductMode: FullPrecision  
MaxProductWordLength: 128  
        SumMode: FullPrecision  
MaxSumWordLength: 128
```

Using Direct Property Referencing with `fi`

You can reference directly into a property for setting or retrieving `fi` object property values using MATLAB structure-like referencing. You do so by using a period to index into a property by name.

For example, to get the `WordLength` of `a`,

```
a.WordLength
```

```
ans =
```

```
16
```

To set the `OverflowMode` of `a`,

```
a.OverflowMode = 'wrap'

a =

    3.1416

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
           WordLength: 16
    FractionLength: 13

        RoundMode: convergent
      OverflowMode: wrap
        ProductMode: FullPrecision
MaxProductWordLength: 128
          SumMode: FullPrecision
    MaxSumWordLength: 128
```

You can also use direct property referencing to associate `fi` objects with the global `fimath`. For example, if you have a `fi` object `b` with a local `fimath` object, you can remove the local `fimath` object and force `b` to associate itself with the global `fimath`:

```
b = fi(pi, 1, 'RoundMode', 'Floor')

b =

    3.1415

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
           WordLength: 16
    FractionLength: 13

        RoundMode: floor
      OverflowMode: saturate
        ProductMode: FullPrecision
MaxProductWordLength: 128
          SumMode: FullPrecision
    MaxSumWordLength: 128
```

```
b.fimath = []
```

```
b =  
    3.1415
```

```
        DataTypeMode: Fixed-point: binary point scaling  
        Signedness: Signed  
        WordLength: 16  
        FractionLength: 13
```

```
isfimathlocal(b)
```

```
ans =  
    0
```

fi Object Functions

You can learn about the functions associated with `fi` objects in the Function Reference.

The following data-access functions can be also used to get the data in a `fi` object using dot notation.

- `bin`
- `data`
- `dec`
- `double`
- `hex`
- `int`
- `oct`

For example,

```
a = fi(pi);  
n = int(a)
```

```
n =
```

```
25736
```

```
a.int
```

```
ans =
```

```
25736
```

```
h = hex(a)
```

```
h =
```

```
6488
```


a.hex

ans =

6488

Working with fimath Objects

- “Constructing fimath Objects” on page 4-2
- “Working with the Global fimath” on page 4-6
- “fimath Object Properties” on page 4-11
- “Using fimath Properties to Perform Fixed-Point Arithmetic” on page 4-16
- “Using fimath to Specify Rounding and Overflow Modes” on page 4-25
- “Using fimath to Share Arithmetic Rules” on page 4-27
- “Using fimath ProductMode and SumMode” on page 4-30
- “fimath Object Functions” on page 4-36

Constructing fimath Objects

In this section...
“fimath Object Syntaxes” on page 4-2
“Building fimath Object Constructors in a GUI” on page 4-4

fimath Object Syntaxes

The arithmetic attributes of `fi` objects are defined by either a local `fimath` object, or the global `fimath`.

You can create `fimath` objects in Fixed-Point Toolbox software in one of two ways:

- You can use the `fimath` constructor function to create new `fimath` objects.
- You can use the `fimath` constructor function to copy an existing `fimath` object.

To get started, type

```
F = fimath
```

to create a `fimath` object. If your global `fimath` is set to the factory default configuration, you get the following output:

```
F =
```

```
RoundMode: nearest  
OverflowMode: saturate  
ProductMode: FullPrecision  
MaxProductWordLength: 128  
SumMode: FullPrecision  
MaxSumWordLength: 128
```

To learn how to configure the global `fimath`, see “Working with the Global `fimath`” on page 4-6.

To copy a `fimath` object, simply use assignment as in the following example:

```
F = fimath;  
G = F;  
isequal(F,G)
```

```
ans =
```

```
1
```

The syntax

```
F = fimath(...'PropertyName',PropertyValue...)
```

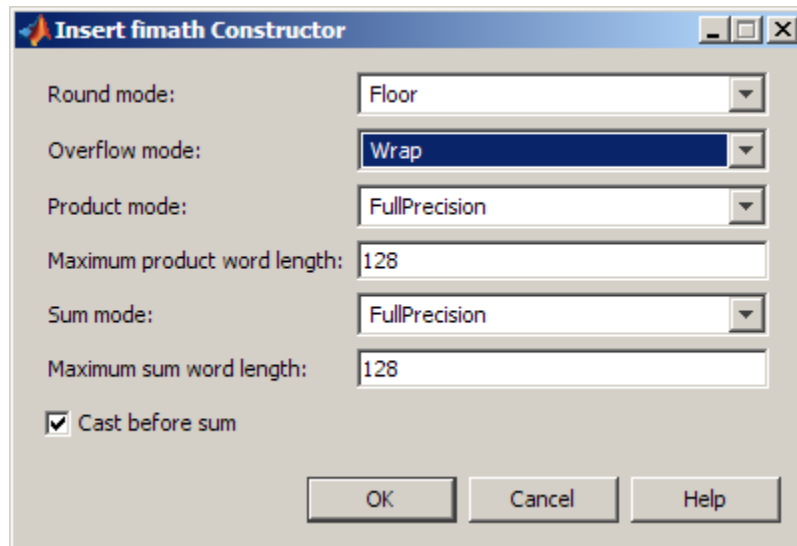
allows you to set properties for a `fimath` object at object creation with property name/property value pairs. Refer to “Setting `fimath` Properties at Object Creation” on page 4-12.

Building `fimath` Object Constructors in a GUI

When you are working with files in MATLAB, you can build your `fimath` object constructors using the **Insert `fimath` Constructor** dialog box. After specifying the properties of the `fimath` object in the dialog box, you can insert the prepopulated `fimath` object constructor string at a specific location in your file.

For example, to create a `fimath` object that uses convergent rounding and wraps on overflow, perform the following steps:

- 1 Open the **Insert `fimath` Constructor** dialog box by selecting **Tools > Fixed-Point Toolbox > Insert `fimath` Constructor** from the editor menu.
- 2 Use the edit boxes and drop-down menus to specify the following properties of the `fimath` object:
 - **Round mode** = Floor
 - **Overflow mode** = Wrap
 - **Product mode** = FullPrecision
 - **Maximum product word length** = 128
 - **Sum mode** = FullPrecision
 - **Maximum sum word length** = 128
 - **Cast before sum** = Checked



- 3 To insert the `fimath` object constructor string in your file, place your cursor at the desired location in the file. Then click **OK** on the **Insert fimath Constructor** dialog box. Clicking **OK** closes the **Insert fimath Constructor** dialog box and automatically populates the `fimath` object constructor string in your file:

```
6      F = fimath('RoundMode', 'Floor', ...
7              'OverflowMode', 'Wrap', ...
8              'ProductMode', 'FullPrecision', ...
9              'MaxProductWordLength', 128, ...
10             'SumMode', 'FullPrecision', ...
11             'MaxSumWordLength', 128, ...
12             'CastBeforeSum', true)
```

Working with the Global fimath

The *global fimath* is the set of `fimath` properties that `fi` objects associate with, and `fimath` constructors return, when you do not specify any `fimath` properties in the constructor. If you do not specify any `fimath` properties in a `fi` object constructor, the resulting `fi` object associates itself with the global `fimath`. If you specify one or more `fimath` object properties in a `fi` object constructor, the resulting `fi` object has a local `fimath` object. All `fimath` object properties that you do not specify in a `fi` or `fimath` object constructor inherit their values from the global `fimath`. The factory-default setting of the global `fimath` has the following properties:

```
RoundMode: nearest
OverflowMode: saturate
ProductMode: FullPrecision
MaxProductWordLength: 128
SumMode: FullPrecision
MaxSumWordLength: 128
```

Note `fi` objects that are associated with the global `fimath` and created from a floating-point value always get constructed with a `RoundMode` of `nearest` and an `OverflowMode` of `saturate`. To construct `fi` objects with different `RoundMode` and `OverflowMode` properties, specify the desired `RoundMode` and `OverflowMode` properties in the `fi` constructor.

If you create a `fi` object with a local `fimath` object, you can use dot notation to change the local `fimath` object properties. You can also use dot notation to remove a local `fimath` object from an existing `fi` object, and associate that `fi` object with the global `fimath`. For example, if the `fi` object `a` has a local `fimath` object, you can use the syntax `a.fimath = []` to remove its local `fimath` object. When you do so, `a` becomes associated with the global `fimath`.

You can configure your global `fimath` settings using the Fixed-Point Toolbox `globalfimath`, `saveglobalfimathpref`, `removeglobalfimathpref`, and `resetglobalfimath` functions. The global `fimath` allows you to define your own set of `fimath` properties for all `fi` and `fimath` objects to use as a default. See the following sections for more information about the Fixed-Point Toolbox global `fimath` functions:

- “Configuring the Global fimath” on page 4-7
- “Saving the Global fimath” on page 4-8
- “Resetting the Global fimath to the Factory Default” on page 4-8
- “Using the Global fimath with the MATLAB Function Block” on page 4-10

For information on how `fi` objects with local `fimath` objects interact with `fi` objects associated with the global `fimath`, see the following section:

“`fimath` Rules for Fixed-Point Arithmetic” on page 4-16

Configuring the Global fimath

You can configure the global `fimath` using the `globalfimath` function. When you provide an output argument, this function returns a handle object to the global `fimath`. You can then control the global `fimath` properties through that handle object. The following syntaxes provide you with a handle object to the global `fimath`:

- `G = globalfimath` — Use this syntax when you want to get a handle object to the global `fimath`, but do not want to change any of the current global `fimath` properties. You can modify the properties of the global `fimath` later using dot notation on the object `G`. For example, the following code gets a handle object to the global `fimath`, and then uses that handle object to change the `RoundMode` of the global `fimath`.

```
G = globalfimath;
G.RoundMode = 'Floor';
```

- `G = globalfimath(f)` — Use this syntax when you want to configure the global `fimath` using an existing `fimath` object `f`. Because this syntax specifies `G` as an output argument, the function returns a handle object to the global `fimath`. You can then modify the properties of the global `fimath` at any time by changing the properties of `G`. For example,

```
f = fimath('RoundMode', 'Floor');
G = globalfimath(f);
G.RoundMode = 'Nearest';
```

- `G = globalfimath('RoundMode', 'Floor')` — Use this syntax when you want to set the properties of the global `fimath` using property name/property

value pairs. You can specify as many property name/property value pairs as necessary. All properties that you do not specify values for inherit their values from the current global fimath. As always, you can use the handle object to modify the global fimath properties at a later time.

```
G = globalfimath('RoundMode','Floor','OverflowMode','Wrap');
G.OverflowMode = 'Saturate';
```

The global fimath properties you set with the `globalfimath` function apply only in your current MATLAB session. To save the global fimath for use in future MATLAB sessions, use the `saveglobalfimathpref` function.

Saving the Global fimath

If you want to use the current global fimath in future MATLAB sessions, you must save it as a preference. The `saveglobalfimathpref` function saves the current global fimath as a preference, and MATLAB uses that global fimath in all future sessions.

To use the current global fimath in all future MATLAB sessions, type

```
saveglobalfimathpref;
```

at the MATLAB command line. MATLAB now uses the current global fimath in all of your future MATLAB sessions.

Resetting the Global fimath to the Factory Default

There are two ways to reset the global fimath to the factory default in your current MATLAB session:

- Use the `resetglobalfimath` function. To do so, type

```
resetglobalfimath;
```

at the MATLAB command line.

- Use the `reset` method of the global fimath. If `G` is a handle object to the global fimath, type

```
reset(G);
```

at the MATLAB command line.

Unless you choose to configure the global fimath again, MATLAB uses the factory-default setting of the global fimath for the remainder of your MATLAB session. For example, after resetting the global fimath to the factory default, MATLAB returns the following `fi` and `fimath` objects:

```
b = fi(pi)

b =
    3.1416

        DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
           WordLength: 16
      FractionLength: 13

b.fimath

ans =

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

F1 = fimath

F1 =

        RoundMode: nearest
      OverflowMode: saturate
     ProductMode: FullPrecision
MaxProductWordLength: 128
           SumMode: FullPrecision
MaxSumWordLength: 128
```

To reset the global fimath to the factory default for all future MATLAB sessions, remove your global fimath from the MATLAB preferences. To do so, type

```
removeglobalfimathpref;
```

at the MATLAB command line. This command sets the global fimath back to its factory-default configuration for all future MATLAB sessions.

Using the Global fimath with the MATLAB Function Block

For information about using the global fimath with the MATLAB Function block, see the following section:

“Using fimath Objects in MATLAB Function Blocks” on page 8-81

fimath Object Properties

In this section...
“Math, Rounding, and Overflow Properties” on page 4-11
“Setting fimath Object Properties” on page 4-12

Math, Rounding, and Overflow Properties

You can always write to the following properties of fimath objects:

Property	Description
CastBeforeSum	Whether both operands are cast to the sum data type before addition
MaxProductWordLength	Maximum allowable word length for the product data type
MaxSumWordLength	Maximum allowable word length for the sum data type
OverflowMode	Overflow-handling mode
ProductBias	Bias of the product data type
ProductFixedExponent	Fixed exponent of the product data type
ProductFractionLength	Fraction length, in bits, of the product data type
ProductMode	Defines how the product data type is determined
ProductSlope	Slope of the product data type
ProductSlopeAdjustmentFactor	Slope adjustment factor of the product data type
ProductWordLength	Word length, in bits, of the product data type
RoundMode	Rounding mode

Property	Description
SumBias	Bias of the sum data type
SumFixedExponent	Fixed exponent of the sum data type
SumFractionLength	Fraction length, in bits, of the sum data type
SumMode	Defines how the sum data type is determined
SumSlope	Slope of the sum data type
SumSlopeAdjustmentFactor	Slope adjustment factor of the sum data type
SumWordLength	Word length, in bits, of the sum data type

For details about these properties, refer to the Property Reference. To learn how to specify properties for `fimath` objects in Fixed-Point Toolbox software, refer to “Setting `fimath` Object Properties” on page 4-12.

Setting `fimath` Object Properties

- “Setting `fimath` Properties at Object Creation” on page 4-12
- “Using Direct Property Referencing with `fimath`” on page 4-13
- “Setting `fimath` Properties in the Model Explorer” on page 4-14

Setting `fimath` Properties at Object Creation

You can set properties of `fimath` objects at the time of object creation by including properties after the arguments of the `fimath` constructor function.

For example, to set the overflow mode to `saturate` and the rounding mode to `convergent`,

```
F = fimath('OverflowMode', 'saturate', 'RoundMode', 'convergent')
```

```
F =
```

```
RoundMode: convergent
OverflowMode: saturate
ProductMode: FullPrecision
MaxProductWordLength: 128
SumMode: FullPrecision
MaxSumWordLength: 128
```

Using Direct Property Referencing with fimath

You can reference directly into a property for setting or retrieving `fimath` object property values using MATLAB structure-like referencing. You do so by using a period to index into a property by name.

For example, to get the `RoundMode` of `F`,

```
F.RoundMode
ans =
convergent
```

To set the `OverflowMode` of `F`,

```
F.OverflowMode = 'wrap'
F =
RoundMode: convergent
OverflowMode: wrap
ProductMode: FullPrecision
MaxProductWordLength: 128
SumMode: FullPrecision
MaxSumWordLength: 128
```

Setting fimath Properties in the Model Explorer

You can view and change the properties for any `fimath` object defined in the MATLAB workspace in the Model Explorer. Open the Model Explorer by selecting **View > Model Explorer** in any Simulink model, or by typing `daexplr` at the MATLAB command line.

The following figure shows the Model Explorer when you define the following `fimath` objects in the MATLAB workspace:

```
F = fimath
```

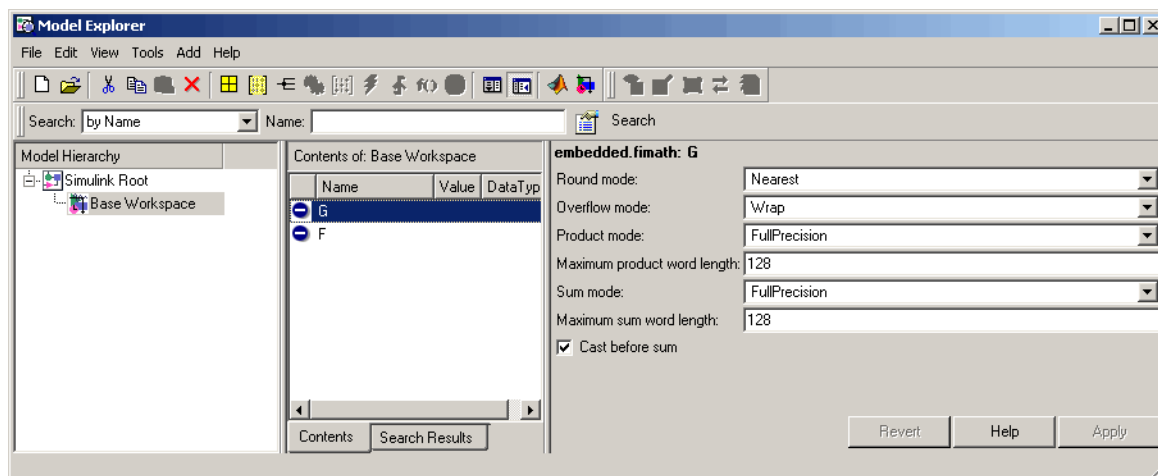
```
F =
```

```
          RoundMode: nearest  
          OverflowMode: saturate  
          ProductMode: FullPrecision  
MaxProductWordLength: 128  
          SumMode: FullPrecision  
MaxSumWordLength: 128
```

```
G = fimath('OverflowMode','wrap')
```

```
G =
```

```
          RoundMode: nearest  
          OverflowMode: wrap  
          ProductMode: FullPrecision  
MaxProductWordLength: 128  
          SumMode: FullPrecision  
MaxSumWordLength: 128
```

Select the **Base Workspace** node in the **Model Hierarchy** pane to view the current objects in the **Contents** pane. When you select a **fimath** object in the **Contents** pane, you can view and change its properties in the **Dialog** pane.

For more information on working with the Model Explorer, see the following sections of the Fixed-Point Toolbox documentation:

- “Specifying Fixed-Point Parameters in the Model Explorer” on page 8-79
- “Sharing Models with Fixed-Point MATLAB Function Blocks” on page 8-84

Using fimath Properties to Perform Fixed-Point Arithmetic

In this section...

“fimath Rules for Fixed-Point Arithmetic” on page 4-16

“Binary-Point Arithmetic” on page 4-18

“[Slope Bias] Arithmetic” on page 4-22

fimath Rules for Fixed-Point Arithmetic

`fi` `fimath` properties define the rules for performing arithmetic operations on `fi` objects. The `fimath` properties that govern fixed-point arithmetic operations can come from a `fimath` object or the global `fimath`. `fi` objects and the global `fimath` have the following differences:

- Individual `fi` objects can have a local `fimath` object. `fi` objects have a local `fimath` object when you specify `fimath` properties in a `fi` constructor or use dot notation to set any `fimath` object property of a `fi` object.
- The global `fimath` is the set of `fimath` properties that `fi` constructors associate with, and `fimath` constructors return, when you do not specify any `fimath` properties in the constructor. When you use the `sfi` or `ufi` constructor or when you fail to specify any `fimath` properties in the `fi` constructor, the resulting `fi` object associates itself with the global `fimath`. It does not have a local `fimath` object. See “Working with the Global `fimath`” on page 4-6 for more information.

To determine whether a `fi` object has a local `fimath` object or associates with the global `fimath`, use the `isfimathlocal` function.

The following sections discuss how `fi` objects with local `fimath` objects interact with `fi` objects associated with the global `fimath`.

Binary Operations

In binary fixed-point operations such as $c = a + b$, the following rules apply:

- If both `a` and `b` associate with the global `fimath`, the operation uses the global `fimath` to perform the fixed-point arithmetic. The output `fi` object `c` also associates with the global `fimath`.

- If either `a` or `b` has a local `fimath` object, the operation uses that `fimath` object to perform the fixed-point arithmetic. The output `fi` object `c` has the same local `fimath` object as the input.

Unary Operations

In unary fixed-point operations such as `b = abs(a)`, the following rules apply:

- If `a` associates with the global `fimath`, the operation uses the global `fimath` to perform the fixed-point arithmetic. The output `fi` object `b` also associates with the global `fimath`.
- If `a` has a local `fimath` object, the operation uses that `fimath` object to perform the fixed-point arithmetic. The output `fi` object `b` has the same local `fimath` object as the input `a`.

When you specify a `fimath` object in the function call of a unary fixed-point operation, the operation uses the `fimath` object you specify to perform the fixed-point arithmetic. For example, when you use a syntax such as `b = abs(a,F)` or `b = sqrt(a,F)`, the `abs` and `sqrt` operations use the `fimath` object `F` to compute intermediate quantities. The output `fi` object `b` always associates with the global `fimath`.

Concatenation Operations

In fixed-point concatenation operations such as `c = [a b]`, `c = [a;b]` and `c = bitconcat(a,b)`, the following rule applies:

- The `fimath` properties of the left-most `fi` object in the operation determine the `fimath` properties of the output `fi` object `c`.

For example, consider the following scenarios for the operation `d = [a b c]`:

- If `a` is a `fi` object associated with the global `fimath`, the output `fi` object `d` also associates with the global `fimath`.
- If `a` has a local `fimath` object, the output `fi` object `d` has the same local `fimath` object.
- If `a` is not a `fi` object, the output `fi` object `d` inherits the `fimath` properties of the next left-most `fi` object. For example, if `b` is a `fi` object with a local

fimath object, the output fi object d has the same local fimath object as the input fi object b.

fimath Object Operations: add, mpy, sub

The output of the fimath object operations add, mpy, and sub always associates with the global fimath. The operations use the fimath object you specify in the function call, but the output fi object never has a local fimath object.

MATLAB Function Block Operations

Fixed-point operations performed with the MATLAB Function block use the same rules as fixed-point operations performed in MATLAB.

All input signals to the MATLAB Function block that you treat as fi objects associate with whatever you specify for the **MATLAB Function block fimath** parameter. When you set this parameter to Same as MATLAB, your fi objects associate with the global fimath. When you set the **MATLAB Function block fimath** parameter to Specify other, you can define your own set of fimath properties for all fi objects in the MATLAB Function block to associate with. You can choose to treat only fixed-point input signals as fi objects or both fixed-point and integer input signals as fi objects. See “Using fimath Objects in MATLAB Function Blocks” on page 8-81.

Binary-Point Arithmetic

The fimath object and the global fimath encapsulate the math properties of Fixed-Point Toolbox software.

fi objects only have a local fimath object when you explicitly specify fimath properties in the fi constructor. When you use the sfi or ufi constructor or do not specify any fimath properties in the fi constructor, the resulting fi object associates itself with the global fimath.

```
a = fi(pi)
```

```
a =  
    3.1416
```

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

```

        WordLength: 16
        FractionLength: 13

a.fimath
isfimathlocal(a)

ans =

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

ans =
    0

```

To perform arithmetic with `+`, `-`, `.*`, or `*` on two `fi` operands with local `fimath` objects, the local `fimath` objects must be identical. If one of the `fi` operands is associated with the global `fimath`, the `fimath` properties of the two operands need not be identical. See “`fimath` Rules for Fixed-Point Arithmetic” on page 4-16 for more information.

```

a = fi(pi);
b = fi(8);
isequal(a.fimath, b.fimath)

ans =

    1

a + b

ans =

    11.1416

```

```

DataTypeMode: Fixed-point: binary point scaling

```

```
Signedness: Signed
WordLength: 19
FractionLength: 13
```

To perform arithmetic with `+`, `-`, `.*`, or `*`, two `fi` operands must also have the same data type. For example, you can perform addition on two `fi` objects with data type `double`, but not on an object with data type `double` and one with data type `single`:

```
a = fi(3, 'DataType', 'double')
```

```
a =
```

```
3
```

```
DataTypeMode: Double
```

```
b = fi(27, 'DataType', 'double')
```

```
b =
```

```
27
```

```
DataTypeMode: Double
```

```
a + b
```

```
ans =
```

```
30
```

```
DataTypeMode: Double
```

```
c = fi(12, 'DataType', 'single')
```

```
c =
```

```
12
```

```
DataTypeMode: Single
```

```
a + c
??? Math operations are not allowed on FI objects with
different data types.
```

Fixed-point `fi` object operands do not have to have the same scaling. You can perform binary math operations on a `fi` object with a fixed-point data type and a `fi` object with a scaled doubles data type. In this sense, the scaled double data type acts as a fixed-point data type:

```
a = fi(pi)

a =

    3.1416

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 13

b = fi(magic(2), ...
'DataTypeMode', 'Scaled double: binary point scaling')

b =

     1     3
     4     2

        DataTypeMode: Scaled double: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 12

a + b

ans =

    4.1416    6.1416
```

```
7.1416    5.1416
```

```
        DataTypeMode: Scaled double: binary point scaling
        Signedness: Signed
        WordLength: 18
        FractionLength: 13
```

Use the `divide` function to perform division with doubles, singles, or binary point-only scaling `fi` objects.

[Slope Bias] Arithmetic

Fixed-Point Toolbox software supports fixed-point arithmetic using the `fimath` object or the global `fimath` for all binary point-only signals. The toolbox also supports arithmetic for [Slope Bias] signals with the following restrictions:

- [Slope Bias] signals must be real.
- You must set the `SumMode` and `ProductMode` properties of the governing `fimath` to 'SpecifyPrecision' for sum and multiply operations, respectively.
- You must set the `CastBeforeSum` property of the governing `fimath` to 'true'.
- Fixed-Point Toolbox does not support the `divide` function for [Slope Bias] signals.

```
f = fimath('SumMode', 'SpecifyPrecision', ...
          'SumFractionLength', 16)
```

```
f =
```

```
        RoundMode: nearest
        OverflowMode: saturate
        ProductMode: FullPrecision
        MaxProductWordLength: 128
        SumMode: SpecifyPrecision
```



```
        SumWordLength: 32
        SumFractionLength: 16
        CastBeforeSum: true

a = fi(pi, 'fimath', f)

a =

    3.1416

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 13

        RoundMode: nearest
        OverflowMode: saturate
        ProductMode: FullPrecision
        MaxProductWordLength: 128
        SumMode: SpecifyPrecision
        SumWordLength: 32
        SumFractionLength: 16
        CastBeforeSum: true

b = fi(22, true, 16, 2^-8, 3, 'fimath', f)

b =

    22

        DataTypeMode: Fixed-point: slope and bias scaling
        Signedness: Signed
        WordLength: 16
        Slope: 0.00390625
        Bias: 3

        RoundMode: nearest
        OverflowMode: saturate
        ProductMode: FullPrecision
        MaxProductWordLength: 128
```

```
SumMode: SpecifyPrecision
SumWordLength: 32
SumFractionLength: 16
CastBeforeSum: true

a + b

ans =

    25.1416

DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 32
FractionLength: 16

RoundMode: nearest
OverflowMode: saturate
ProductMode: FullPrecision
MaxProductWordLength: 128
SumMode: SpecifyPrecision
SumWordLength: 32
SumFractionLength: 16
CastBeforeSum: true
```

Setting the `SumMode` and `ProductMode` properties to `SpecifyPrecision` are mutually exclusive except when performing the `*` operation between matrices. In this case, you must set both the `SumMode` and `ProductMode` properties to `SpecifyPrecision` for [Slope Bias] signals. Doing so is necessary because the `*` operation performs both sum and multiply operations to calculate the result.

Using fimath to Specify Rounding and Overflow Modes

Only rounding and overflow modes set prior to an operation with `fi` objects affect the outcome of those operations. Once you create a `fi` object in MATLAB, changing its rounding or overflow mode does not affect its value. For example, consider the `fi` objects `a` and `b`:

```
p = fipref('NumberDisplay', 'RealWorldValue',...
'NumericTypeDisplay', 'none', 'FimathDisplay', 'none');
T = numericType('WordLength',8,'FractionLength',7);
F = fimath('RoundMode', 'floor', 'OverflowMode', 'wrap');
a = fi(1,T,F)
```

```
a =
```

```
-1
```

```
b = fi(1,T)
```

```
b =
```

```
0.9922
```

Because you create `a` with a `fimath` object `F` that has `OverflowMode` set to `wrap`, the value of `a` wraps to `-1`. Conversely, because you create `b` with the default `OverflowMode` value of `saturate`, its value saturates to `0.9922`.

Now, assign the `fimath` object `F` to `b`:

```
b.fimath = F
```

```
b =
```

```
0.9922
```

Because the assignment operation and corresponding overflow and saturation happened when you created `b`, its value does not change when you assign it the new `fimath` object `F`.

Note `fi` objects that are associated with the global `fimath` and created from a floating-point value always get constructed with a `RoundMode` of `nearest` and an `OverflowMode` of `saturate`. To construct `fi` objects with different `RoundMode` and `OverflowMode` properties, specify the desired `RoundMode` and `OverflowMode` properties in the `fi` constructor.

Using fimath to Share Arithmetic Rules

There are two ways of sharing `fimath` properties in Fixed-Point Toolbox software:

- “Using the Global `fimath` to Share Arithmetic Rules” on page 4-27
- “Using `fimath` Objects to Share Arithmetic Rules” on page 4-28

Sharing `fimath` properties across `fi` objects ensures that the `fi` objects are using the same arithmetic rules and helps you avoid “mismatched `fimath`” errors.

Using the Global `fimath` to Share Arithmetic Rules

You can ensure that your `fi` objects are all using the same `fimath` properties by associating them with the global `fimath`. To associate a `fi` object with the global `fimath`, you can:

- Create a `fi` object using the `fi` constructor without specifying any `fimath` properties in the constructor call. For example:

```
a = fi(pi)
```

- Create a `fi` object using the `sfi` or `ufi` constructor. All `fi` objects created with these constructors associate themselves with the global `fimath`.

```
b = sfi(pi)
```

- Use dot notation to remove a local `fimath` object from an existing `fi` object. Doing so automatically associates the `fi` object with the global `fimath`. For example:

```
b = fi(pi, 'RoundMode', 'Fix')  
b.fimath = []
```

To learn how to configure the global `fimath`, see “Working with the Global `fimath`” on page 4-6.

Using fimath Objects to Share Arithmetic Rules

You can also use a `fimath` object to define common arithmetic rules that you would like to use for multiple `fi` objects. You can then create your `fi` objects, using the same `fimath` object for each. To do so, you must also create a `numerictype` object to define a common data type and scaling. Refer to Chapter 6, “Working with `numerictype` Objects” for more information on `numerictype` objects. The following example shows the creation of a `numerictype` object and `fimath` object, and then uses those objects to create two `fi` objects with the same `numerictype` and `fimath` attributes:

```
T = numerictype('WordLength', 32, 'FractionLength', 30)

T =

    DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 32
    FractionLength: 30

F = fimath('RoundMode', 'floor', 'OverflowMode', 'wrap')

F =

    RoundMode: floor
    OverflowMode: wrap
    ProductMode: FullPrecision
    MaxProductWordLength: 128
    SumMode: FullPrecision
    MaxSumWordLength: 128

a = fi(pi, T, F)

a =

    -0.8584

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

```
Signedness: Signed
WordLength: 32
FractionLength: 30
```

```
RoundMode: floor
OverflowMode: wrap
ProductMode: FullPrecision
MaxProductWordLength: 128
SumMode: FullPrecision
MaxSumWordLength: 128
```

```
b = fi(pi/2, T, F)
```

```
b =
```

```
1.5708
```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 32
FractionLength: 30
```

```
RoundMode: floor
OverflowMode: wrap
ProductMode: FullPrecision
MaxProductWordLength: 128
SumMode: FullPrecision
MaxSumWordLength: 128
```

Using fimath ProductMode and SumMode

In this section...

“Example Setup” on page 4-30

“FullPrecision” on page 4-31

“KeepLSB” on page 4-32

“KeepMSB” on page 4-33

“SpecifyPrecision” on page 4-34

Example Setup

The examples in the sections of this topic show the differences among the four settings of the ProductMode and SumMode properties:

- FullPrecision
- KeepLSB
- KeepMSB
- SpecifyPrecision

To follow along, first set the following preferences:

```
p = fipref;  
p.NumericTypeDisplay = 'short';  
p.FimathDisplay = 'none';  
p.LoggingMode = 'on';  
F = fimath('OverflowMode', 'wrap', 'RoundMode', 'floor', ...  
    'CastBeforeSum', false);  
warning off  
format compact
```

Next, define `fi` objects `a` and `b`. Both have signed 8-bit data types. The fraction length gets chosen automatically for each `fi` object to yield the best possible precision:

```
a = fi(pi, true, 8)  
a =
```



```

3.1563
s8,5

b = fi(exp(1), true, 8)
b =
2.7188
s8,5

```

FullPrecision

Now, set ProductMode and SumMode for a and b to FullPrecision and look at some results:

```

F.ProductMode = 'FullPrecision';
F.SumMode = 'FullPrecision';
a.fimath = F;
b.fimath = F;
a
a =
3.1563    %011.00101
s8,5

b
b =
2.7188    %010.10111
s8,5

a*b
ans =
8.5811    %001000.1001010011
s16,10

a+b
ans =
5.8750    %0101.11100
s9,5

```

In FullPrecision mode, the product word length grows to the sum of the word lengths of the operands. In this case, each operand has 8 bits, so the product word length is 16 bits. The product fraction length is the sum of the fraction lengths of the operands, in this case $5 + 5 = 10$ bits.

The sum word length grows by one most significant bit to accommodate the possibility of a carry bit. The sum fraction length aligns with the fraction lengths of the operands, and all fractional bits are kept for full precision. In this case, both operands have 5 fractional bits, so the sum has 5 fractional bits.

KeepLSB

Now, set ProductMode and SumMode for a and b to KeepLSB and look at some results:

```
F.ProductMode = 'KeepLSB';
F.ProductWordLength = 12;
F.SumMode = 'KeepLSB';
F.SumWordLength = 12;
a.fimath = F;
b.fimath = F;
a
a =
    3.1563    %011.00101
         s8,5

b
b =
    2.7188    %010.10111
         s8,5

a*b
ans =
    0.5811    %00.1001010011
         s12,10

a+b
ans =
    5.8750    %0000101.11100
         s12,5
```

In KeepLSB mode, you specify the word lengths and the least significant bits of results are automatically kept. This mode models the behavior of integer operations in the C language.

The product fraction length is the sum of the fraction lengths of the operands. In this case, each operand has 5 fractional bits, so the product fraction length is 10 bits. In this mode, all 10 fractional bits are kept. Overflow occurs because the full-precision result requires 6 integer bits, and only 2 integer bits remain in the product.

The sum fraction length aligns with the fraction lengths of the operands, and in this model all least significant bits are kept. In this case, both operands had 5 fractional bits, so the sum has 5 fractional bits. The full-precision result requires 4 integer bits, and 7 integer bits remain in the sum, so no overflow occurs in the sum.

KeepMSB

Now, set ProductMode and SumMode for a and b to KeepMSB and look at some results:

```
F.ProductMode = 'KeepMSB';
F.ProductWordLength = 12;
F.SumMode = 'KeepMSB';
F.SumWordLength = 12;
a.fimath = F;
b.fimath = F;
a
a =
    3.1563    %011.00101
         s8,5

b
b =
    2.7188    %010.10111
         s8,5

a*b
ans =
    8.5781    %001000.100101
         s12,6

a+b
ans =
```

```
5.8750    %0101.11100000
          s12,8
```

In `KeepMSB` mode, you specify the word lengths and the most significant bits of sum and product results are automatically kept. This mode models the behavior of many DSP devices where the product and sum are kept in double-wide registers, and the programmer chooses to transfer the most significant bits from the registers to memory after each operation.

The full-precision product requires 6 integer bits, and the fraction length of the product is adjusted to accommodate all 6 integer bits in this mode. No overflow occurs. However, the full-precision product requires 10 fractional bits, and only 6 are available. Therefore, precision is lost.

The full-precision sum requires 4 integer bits, and the fraction length of the sum is adjusted to accommodate all 4 integer bits in this mode. The full-precision sum requires only 5 fractional bits; in this case there are 8, so there is no loss of precision.

SpecifyPrecision

Now set `ProductMode` and `SumMode` for `a` and `b` to `SpecifyPrecision` and look at some results:

```
F.ProductMode = 'SpecifyPrecision';
F.ProductWordLength = 8;
F.ProductFractionLength = 7;
F.SumMode = 'SpecifyPrecision';
F.SumWordLength = 8;
F.SumFractionLength = 7;
a.fimath = F;
b.fimath = F;
a
a =
    3.1563    %011.00101
          s8,5

b
b =
    2.7188    %010.10111
```

```
                s8,5  
  
a*b  
ans =  
    0.5781    %0.1001010  
                s8,7  
  
a+b  
ans =  
   -0.1250    %1.1110000  
                s8,7
```

In `SpecifyPrecision` mode, you must specify both word length and fraction length for sums and products. This example unwisely uses fractional formats for the products and sums, with 8-bit word lengths and 7-bit fraction lengths.

The full-precision product requires 6 integer bits, and the example specifies only 1, so the product overflows. The full-precision product requires 10 fractional bits, and the example only specifies 7, so there is precision loss in the product.

The full-precision sum requires 2 integer bits, and the example specifies only 1, so the sum overflows. The full-precision sum requires 5 fractional bits, and the example specifies 7, so there is no loss of precision in the sum.

fimath Object Functions

You can learn about the functions associated with `fimath` objects in the Function Reference.

Working with fipref Objects

- “Constructing fipref Objects” on page 5-2
- “fipref Object Properties” on page 5-3
- “Using fipref Objects to Set Display Preferences” on page 5-5
- “Using fipref Objects to Set Logging Preferences” on page 5-7
- “Using fipref Objects to Set Data Type Override Preferences” on page 5-12
- “fipref Object Functions” on page 5-15

Constructing fipref Objects

The `fipref` object defines the display and logging attributes for all `fi` objects. You can use the `fipref` constructor function to create a new object.

To get started, type

```
P = fipref
```

to create a default `fipref` object.

```
P =  
    NumberDisplay: 'RealWorldValue'  
    NumericTypeDisplay: 'full'  
    FimathDisplay: 'full'  
    LoggingMode: 'Off'  
    DataTypeOverride: 'ForceOff'
```

The syntax

```
P = fipref(...'PropertyName','PropertyValue'...)
```

allows you to set properties for a `fipref` object at object creation with property name/property value pairs.

Your `fipref` settings persist throughout your MATLAB session. Use `reset(fipref)` to return to the default settings during your session. Use `savefipref` to save your display preferences for subsequent MATLAB sessions.

fipref Object Properties

In this section...

“Display, Data Type Override, and Logging Properties” on page 5-3

“Setting fipref Object Properties” on page 5-3

Display, Data Type Override, and Logging Properties

The following properties of `fipref` objects are always writable:

- `FimathDisplay` — Display options for the local `fimath` attributes of a `fi` object
- `DataTypeOverride` — Data type override options
- `LoggingMode` — Logging options for operations performed on `fi` objects
- `NumericTypeDisplay` — Display options for the numeric type attributes of a `fi` object
- `NumberDisplay` — Display options for the value of a `fi` object

These properties are described in detail in the Property Reference. To learn how to specify properties for `fipref` objects in Fixed-Point Toolbox software, refer to “Setting fipref Object Properties” on page 5-3.

Setting fipref Object Properties

Setting fipref Properties at Object Creation

You can set properties of `fipref` objects at the time of object creation by including properties after the arguments of the `fipref` constructor function. For example, to set `NumberDisplay` to `bin` and `NumericTypeDisplay` to `short`,

```
P = fipref('NumberDisplay', 'bin', ...  
         'NumericTypeDisplay', 'short')
```

```
P =  
    NumberDisplay: 'bin'  
 NumericTypeDisplay: 'short'
```

```
FimathDisplay: 'full'  
LoggingMode: 'Off'  
DataTypeOverride: 'ForceOff'
```

Using Direct Property Referencing with fipref

You can reference directly into a property for setting or retrieving `fipref` object property values using MATLAB structure-like referencing. You do this by using a period to index into a property by name.

For example, to get the `NumberDisplay` of `P`,

```
P.NumberDisplay
```

```
ans =
```

```
bin
```

To set the `NumericTypeDisplay` of `P`,

```
P.NumericTypeDisplay = 'full'
```

```
P =
```

```
NumberDisplay: 'bin'  
NumericTypeDisplay: 'full'  
FimathDisplay: 'full'  
LoggingMode: 'Off'  
DataTypeOverride: 'ForceOff'
```

Using fipref Objects to Set Display Preferences

You use the `fipref` object to specify three aspects of the display of `fi` objects: the object value, the local `fimath` properties, and the `numericType` properties.

For example, the following code shows the default `fipref` display for a `fi` object with a local `fimath` object:

```
a = fi(pi, 'RoundMode', 'floor', 'OverflowMode', 'wrap')

a =
    3.1415

        DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
            WordLength: 16
    FractionLength: 13

            RoundMode: floor
           OverflowMode: wrap
            ProductMode: FullPrecision
MaxProductWordLength: 128
            SumMode: FullPrecision
    MaxSumWordLength: 128
```

The default `fipref` display for a `fi` object associated with the global `fimath` is as follows:

```
a = fi(pi)

a =
    3.1416

        DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
            WordLength: 16
    FractionLength: 13
```

Next, change the `fipref` display properties:

```
P = fipref;
P.NumberDisplay = 'bin';
P.NumericTypeDisplay = 'short';
P.FimathDisplay = 'none'

P =
    NumberDisplay: 'bin'
  NumericTypeDisplay: 'short'
    FimathDisplay: 'none'
      LoggingMode: 'Off'
  DataTypeOverride: 'ForceOff'

a

a =
0110010010000111
    s16,13
```

For more information on the default `fipref` display, see “Display Settings” on page 1-7 in the Getting Started section of the Fixed-Point Toolbox User’s Guide.

Using fipref Objects to Set Logging Preferences

In this section...

“Logging Overflows and Underflows as Warnings” on page 5-7

“Accessing Logged Information with Functions” on page 5-9

Logging Overflows and Underflows as Warnings

Overflows and underflows are logged as warnings for all assignment, plus, minus, and multiplication operations when the fipref LoggingMode property is set to on. For example, try the following:

- 1 Create a signed fi object that is a vector of values from 1 to 5, with 8-bit word length and 6-bit fraction length.

```
a = fi(1:5,1,8,6);
```

- 2 Define the fimath object associated with a, and indicate that you will specify the sum and product word and fraction lengths.

```
F = a.fimath;  
F.SumMode = 'SpecifyPrecision';  
F.ProductMode = 'SpecifyPrecision';  
a.fimath = F;
```

- 3 Define the fipref object and turn on overflow and underflow logging.

```
P = fipref;  
P.LoggingMode = 'on';
```

- 4 Suppress the numericType and fimath displays.

```
P.NumericTypeDisplay = 'none';  
P.FimathDisplay = 'none';
```

- 5 Specify the sum and product word and fraction lengths.

```
a.SumWordLength = 16;  
a.SumFractionLength = 15;
```

```
a.ProductWordLength = 16;  
a.ProductFractionLength = 15;
```

- 6** Warnings are displayed for overflows and underflows in assignment operations. For example, try:

```
a(1) = pi  
Warning: 1 overflow occurred in the fi assignment operation.
```

```
a =
```

```
1.9844    1.9844    1.9844    1.9844    1.9844
```

```
a(1) = double(eps(a))/10
```

```
Warning: 1 underflow occurred in the fi assignment operation.
```

```
a =
```

```
0    1.9844    1.9844    1.9844    1.9844
```

- 7** Warnings are displayed for overflows and underflows in addition and subtraction operations. For example, try:

```
a+a
```

```
Warning: 12 overflows occurred in the fi + operation.
```

```
ans =
```

```
0    1.0000    1.0000    1.0000    1.0000
```

```
a-a
```

```
Warning: 8 overflows occurred in the fi - operation.
```

```
ans =
```

```
0    0    0    0    0
```

- 8** Warnings are displayed for overflows and underflows in multiplication operations. For example, try:

```

a.*a
Warning: 4 product overflows occurred in the fi .* operation.

ans =

           0    1.0000    1.0000    1.0000    1.0000

a*a'
Warning: 4 product overflows occurred in the fi * operation.
Warning: 3 sum overflows occurred in the fi * operation.

ans =

    1.0000

```

The final example above is a complex multiplication that requires both multiplication and addition operations. The warnings inform you of overflows and underflows in both.

Because overflows and underflows are logged as warnings, you can use the `dbstop` MATLAB function with the syntax

```
dbstop if warning
```

to find the exact lines in a file that are causing overflows or underflows.

Use

```
dbstop if warning fi:underflow
```

to stop only on lines that cause an underflow. Use

```
dbstop if warning fi:overflow
```

to stop only on lines that cause an overflow.

Accessing Logged Information with Functions

When the `fipref` `LoggingMode` property is set to on, you can use the following functions to return logged information about assignment and creation operations to the MATLAB command line:

- `maxlog` — Returns the maximum real-world value
- `minlog` — Returns the minimum value
- `noverflows` — Returns the number of overflows
- `nunderflows` — Returns the number of underflows

`LoggingMode` must be set to on before you perform any operation in order to log information about it. To clear the log, use the function `resetlog`.

For example, consider the following. First turn logging on, then perform operations, and then finally get information about the operations:

```
fipref('LoggingMode','on');
x = fi([-1.5 eps 0.5], true, 16, 15);
x(1) = 3.0;
maxlog(x)
```

```
ans =

    1.0000
```

```
minlog(x)
```

```
ans =

   -1
```

```
noverflows(x)
```

```
ans =

     2
```

```
nunderflows(x)
```

```
ans =

     1
```


Next, reset the log and request the same information again. Note that the functions return empty [], because logging has been reset since the operations were run:

```
resetlog(x)
```

```
maxlog(x)
```

```
ans =
```

```
    []
```

```
minlog(x)
```

```
ans =
```

```
    []
```

```
noverflows(x)
```

```
ans =
```

```
    []
```

```
nunderflows(x)
```

```
ans =
```

```
    []
```

Using fipref Objects to Set Data Type Override Preferences

In this section...

“Overriding the Data Type of fi Objects” on page 5-12

“Using Data Type Override to Help Set Fixed-Point Scaling” on page 5-13

Overriding the Data Type of fi Objects

Use the fipref `DataTypeOverride` property to override `fi` objects with singles, doubles, or scaled doubles. Data type override only occurs when the `fi` constructor function is called. Objects that are created while data type override is on have the overridden data type. They maintain that data type when data type override is later turned off. To obtain an object with a data type that is not the override data type, you must create an object when data type override is off:

```
p = fipref('DataTypeOverride', 'TrueDoubles')
```

```
p =
```

```
    NumberDisplay: 'RealWorldValue'  
NumericTypeDisplay: 'full'  
    FimathDisplay: 'full'  
    LoggingMode: 'Off'  
    DataTypeOverride: 'TrueDoubles'
```

```
a = fi(pi)
```

```
a =
```

```
    3.1416
```

```
    DataTypeMode: Double
```

```
p = fipref('DataTypeOverride', 'ForceOff')
```

```
p =
```

```
        NumberDisplay: 'RealWorldValue'  
NumericTypeDisplay: 'full'  
        FimathDisplay: 'full'  
        LoggingMode: 'Off'  
        DataTypeOverride: 'ForceOff'  
  
a  
  
a =  
  
    3.1416  
  
        DataTypeMode: Double  
  
b = fi(pi)  
  
b =  
  
    3.1416  
  
        DataTypeMode: Fixed-point: binary point scaling  
        Signedness: Signed  
        WordLength: 16  
        FractionLength: 13
```

Tip To reset the `fipref` object to its default values use `reset(fipref)` or `reset(p)`, where `p` is a `fipref` object. This is useful to ensure that data type override and logging are off.

Using Data Type Override to Help Set Fixed-Point Scaling

Choosing the scaling for the fixed-point variables in your algorithms can be difficult. In Fixed-Point Toolbox software, you can use a combination of data type override and min/max logging to help you discover the numerical ranges that your fixed-point data types need to cover. These ranges dictate the appropriate scalings for your fixed-point data types. In general, the procedure is

- 1** Implement your algorithm using fixed-point `fi` objects, using initial “best guesses” for word lengths and scalings.
- 2** Set the `fipref` `DataTypeOverride` property to `ScaledDoubles`, `TrueSingles`, or `TrueDoubles`.
- 3** Set the `fipref` `LoggingMode` property to `on`.
- 4** Use the `maxlog` and `minlog` functions to log the maximum and minimum values achieved by the variables in your algorithm in floating-point mode.
- 5** Set the `fipref` `DataTypeOverride` property to `ForceOff`.
- 6** Use the information obtained in step 4 to set the fixed-point scaling for each variable in your algorithm such that the full numerical range of each variable is representable by its data type and scaling.

A detailed example of this process is shown in the Fixed-Point Toolbox Fixed-Point Data Type Override, Min/Max Logging, and Scaling demo.

fipref Object Functions

You can learn about the functions associated with `fipref` objects in the Function Reference.

Working with numerictype Objects

- “Constructing numerictype Objects” on page 6-2
- “numerictype Object Properties” on page 6-7
- “The numerictype Structure” on page 6-11
- “Using numerictype Objects to Share Data Type and Scaling Settings of fi objects” on page 6-14
- “numerictype Object Functions” on page 6-17

Constructing numerictype Objects

In this section...

“numerictype Object Syntaxes” on page 6-2

“Example: Constructing a numerictype Object with Property Name and Property Value Pairs” on page 6-3

“Example: Copying a numerictype Object” on page 6-4

“Example: Building numerictype Object Constructors in a GUI” on page 6-5

numerictype Object Syntaxes

numerictype objects define the data type and scaling attributes of `fi` objects, as well as Simulink signals and model parameters. You can create numerictype objects in Fixed-Point Toolbox software in one of two ways:

- You can use the numerictype constructor function to create a new object.
- You can use the numerictype constructor function to copy an existing numerictype object.

To get started, type

```
T = numerictype
```

to create a default numerictype object.

```
T =
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Signed  
      WordLength: 16  
      FractionLength: 15
```

To see all of the numerictype object syntaxes, refer to the numerictype constructor function reference page.

The following examples show different ways of constructing `numerictype` objects. For more examples of constructing `numerictype` objects, see the “Examples” on the `numerictype` constructor function reference page.

Example: Constructing a numerictype Object with Property Name and Property Value Pairs

When you create a `numerictype` object using property name and property value pairs, Fixed-Point Toolbox software first creates a default `numerictype` object, and then, for each property name you specify in the constructor, assigns the corresponding value.

This behavior differs from the behavior that occurs when you use a syntax such as `T = numerictype(s,w)`, where you only specify the property values in the constructor. Using such a syntax results in no default `numerictype` object being created, and the `numerictype` object receives only the assigned property values that are specified in the constructor.

The following example shows how the property name/property value syntax creates a slightly different `numerictype` object than the property values syntax, even when you specify the same property values in both constructors.

To demonstrate this difference, suppose you want to create an unsigned `numerictype` object with a word length of 32 bits.

First, create the `numerictype` object using property name/property value pairs.

```
T1 = numerictype('signed',0,'wordlength',32)
```

```
T1 =
```

```
      DataTypeMode: Fixed-point: binary point scaling  
      Signedness: Unsigned  
      WordLength: 32  
      FractionLength: 15
```

The numerictype object T1 has the same DataTypeMode and FractionLength as a default numerictype object, but the WordLength and Signed properties are overwritten with the values you specified.

Now, create another unsigned 32 bit numerictype object, but this time specify only property values in the constructor.

```
T2 = numerictype(0,32)
```

```
T2 =
```

```
DataTypeMode: Fixed-point: unspecified scaling  
Signedness: Unsigned  
WordLength: 32
```

Unlike T1, T2 only has the property values you specified. The DataTypeMode of T2 is Fixed-Point: unspecified scaling, so no fraction length is assigned.

fi objects cannot have unspecified numerictype properties. Thus, all unspecified numerictype object properties become specified at the time of fi object creation.

Example: Copying a numerictype Object

To copy a numerictype object, simply use assignment as in the following example:

```
T = numerictype;  
U = T;  
isequal(T,U)
```

```
ans =
```

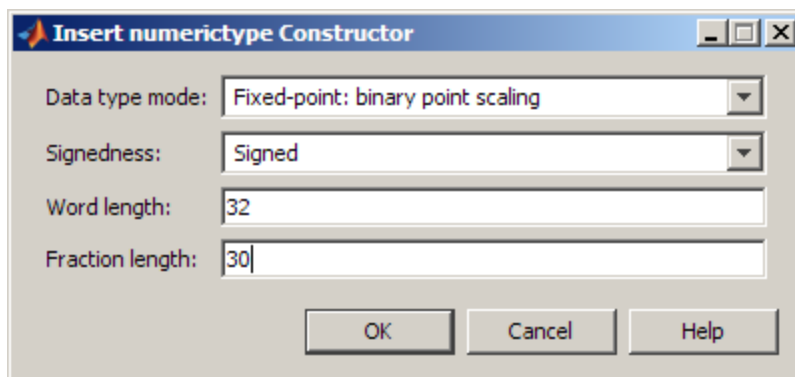
```
1
```

Example: Building numerictype Object Constructors in a GUI

When you are working with files in MATLAB, you can build your numerictype object constructors using the **Insert numerictype Constructor** dialog box. After specifying the properties of the numerictype object in the dialog box, you can insert the prepopulated numerictype object constructor string at a specific location in your file.

For example, to create a signed numerictype object with binary-point scaling, a word length of 32 bits and a fraction length of 30 bits, perform the following steps:

- 1 Open the **Insert numerictype Constructor** dialog box by selecting **Tools > Fixed-Point Toolbox > Insert numerictype Constructor** from the editor menu.
- 2 Use the edit boxes and drop-down menus to specify the following properties of the numerictype object:
 - **Data type mode** = Fixed-point: binary point scaling
 - **Signedness** = Signed
 - **Word length** = 32
 - **Fraction length** = 30



- 3 To insert the numerictype object constructor string in your file, place your cursor at the desired location in the file, and click **OK** on the **Insert**

numerictype Constructor dialog box. Clicking **OK** closes the **Insert numerictype Constructor** dialog box and automatically populates the numerictype object constructor string in your file:

```
5 T = numerictype(1, 32, 30)
```

numerictype Object Properties

In this section...
“Data Type and Scaling Properties” on page 6-7
“Setting numerictype Object Properties” on page 6-8

Data Type and Scaling Properties

All properties of a `numerictype` object are writable. However, the `numerictype` properties of a `fi` object become read only after the `fi` object has been created. Any `numerictype` properties of a `fi` object that are unspecified at the time of `fi` object creation are automatically set to their default values. The properties of a `numerictype` object are:

- `Bias` — Bias
- `DataType` — Data type category
- `DataTypeMode` — Data type and scaling mode
- `FixedExponent` — Fixed-point exponent
- `SlopeAdjustmentFactor` — Slope adjustment
- `FractionLength` — Fraction length of the stored integer value, in bits
- `Scaling` — Fixed-point scaling mode
- `Signed` — Signed or unsigned
- `Signedness` — Signed, unsigned, or auto
- `Slope` — Slope
- `WordLength` — Word length of the stored integer value, in bits

These properties are described in detail in the Property Reference. To learn how to specify properties for `numerictype` objects in Fixed-Point Toolbox software, refer to “Setting numerictype Object Properties” on page 6-8.

Setting numerictype Object Properties

Setting numerictype Properties at Object Creation

You can set properties of numerictype objects at the time of object creation by including properties after the arguments of the numerictype constructor function.

For example, to set the word length to 32 bits and the fraction length to 30 bits,

```
T = numerictype('WordLength', 32, 'FractionLength', 30)
```

```
T =
```

```
          DataTypeMode: Fixed-point: binary point scaling  
          Signedness: Signed  
          WordLength: 32  
          FractionLength: 30
```

Using Direct Property Referencing with numerictype Objects

You can reference directly into a property for setting or retrieving numerictype object property values using MATLAB structure-like referencing. You do this by using a period to index into a property by name.

For example, to get the word length of T,

```
T.WordLength
```

```
ans =
```

```
32
```

To set the fraction length of T,

```
T.FractionLength = 31
```

```
T =
```

```

        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
          WordLength: 32
    FractionLength: 31
    
```

Setting numerictype Properties in the Model Explorer

You can view and change the properties for any numerictype object defined in the MATLAB workspace in the Model Explorer. Open the Model Explorer by selecting **View > Model Explorer** in any Simulink model, or by typing `daexplr` at the MATLAB command line.

The figure below shows the Model Explorer when you define the following numerictype objects in the MATLAB workspace:

```
T = numerictype
```

```
T =
```

```

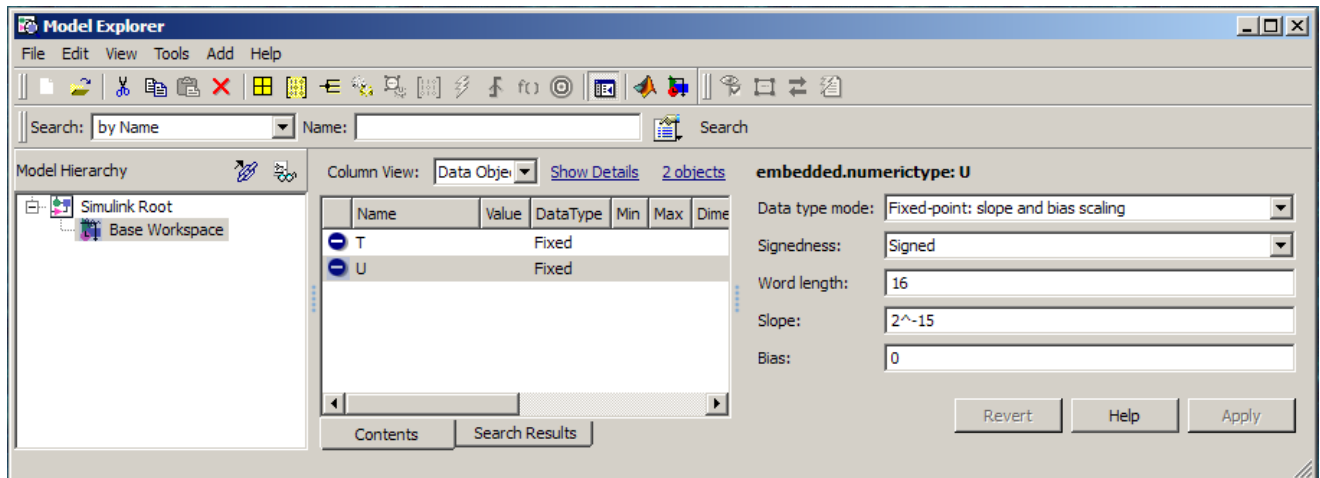
        DataTypeMode: Fixed-point: binary point scaling
          Signedness: Signed
          WordLength: 16
    FractionLength: 15
    
```

```
U = numerictype('DataTypeMode', 'Fixed-point: slope and bias')
```

```
U =
```

```

        DataTypeMode: Fixed-point: slope and bias scaling
          Signedness: Signed
          WordLength: 16
          Slope: 2^-15
          Bias: 0
    
```



Select the **Base Workspace** node in the **Model Hierarchy** pane to view the current objects in the **Contents** pane. When you select a **numerictype** object in the **Contents** pane, you can view and change its properties in the **Dialog** pane.

The numericType Structure

In this section...

“Valid Values for numericType Structure Properties” on page 6-11

“Properties That Affect the Slope” on page 6-13

“Stored Integer Value and Real World Value” on page 6-13

Valid Values for numericType Structure Properties

The numericType object contains all the data type and scaling attributes of a fixed-point object. The numericType object behaves like any MATLAB structure, except that it only lets you set valid values for defined fields. The following table shows the possible settings of each field of the structure.

Note When you create a `fi` object, any unspecified field of the numericType object reverts to its default value. Thus, if the `DataTypeMode` is set to unspecified scaling, it defaults to binary point scaling when the `fi` object is created. If the `Signedness` property of the numericType object is set to `Auto`, it defaults to `Signed` when the `fi` object is created.

DataTypeMode	DataType	Scaling	Signedness	Word- Length	Fraction- Length	Slope	Bias
<i>Fixed-point data types</i>							
Fixed-point: binary point scaling	Fixed	BinaryPoint	Signed Unsigned Auto	Positive integer from 1 to 65,536	Positive or negative integer	$2^{(-fraction\ length)}$	Any floating- point number
Fixed-point: slope and bias scaling	Fixed	SlopeBias	Signed Unsigned Auto	Positive integer from 1 to 65,536	N/A	Any floating- point number	Any floating- point number

DataTypeMode	DataType	Scaling	Signedness	Word- Length	Fraction- Length	Slope	Bias
Fixed-point: unspecified scaling	Fixed	Unspecified	Signed Unsigned Auto	Positive integer from 1 to 65,536	N/A	N/A	N/A
<i>Scaled double data types</i>							
Scaled double: binary point scaling	ScaledDouble	BinaryPoint	Signed Unsigned Auto	Positive integer from 1 to 65,536	Positive or negative integer	$2^{(-fractionlength)}$	
Scaled double: slope and bias scaling	ScaledDouble	SlopeBias	Signed Unsigned Auto	Positive integer from 1 to 65,536	N/A	Any floating- point number	Any floating- point number
Scaled double: unspecified scaling	ScaledDouble	Unspecified	Signed Unsigned Auto	Positive integer from 1 to 65,536	N/A	N/A	N/A
<i>Built-in data types</i>							
Double	double	N/A	1 true	64	0	1	0
Single	single	N/A	1 true	32	0	1	0
Boolean	boolean	N/A	0 false	1	0	1	0

You cannot change the numerictype properties of a fi object after fi object creation.

Properties That Affect the Slope

The **Slope** field of the numerictype structure is related to the SlopeAdjustmentFactor and FixedExponent properties by

$$\text{slope} = \text{slope adjustment factor} \times 2^{\text{fixed exponent}}$$

The FixedExponent and FractionLength properties are related by

$$\text{fixed exponent} = -\text{fraction length}$$

If you set the SlopeAdjustmentFactor, FixedExponent, or FractionLength property, the **Slope** field is modified.

Stored Integer Value and Real World Value

The numerictype StoredIntegerValue and RealWorldValue properties are related according to

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

which is equivalent to

$$\text{real-world value} = \text{stored integer value} \times (\text{slope adjustment factor} \times 2^{\text{fixed exponent}}) + \text{bias}$$

If any of these properties is updated, the others are modified accordingly.

Using numerictype Objects to Share Data Type and Scaling Settings of fi objects

You can use a numerictype object to define common data type and scaling rules that you would like to use for many fi objects. You can then create multiple fi objects, using the same numerictype object for each.

Example 1

In the following example, you create a numerictype object T with word length 32 and fraction length 28. Next, to ensure that your fi objects have the same numerictype attributes, create fi objects a and b using your numerictype object T.

```
format long g
T = numerictype('WordLength',32,'FractionLength',28)

T =

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 32
        FractionLength: 28

a = fi(pi,T)

a =

        3.1415926553309

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 32
        FractionLength: 28

b = fi(pi/2, T)
```

```
b =
```

```
1.5707963258028
```

```
DataTypeMode: Fixed-point: binary point scaling  
Signedness: Signed  
WordLength: 32  
FractionLength: 28
```

Example 2

In this example, start by creating a `numerictype` object `T` with [Slope Bias] scaling. Next, use that object to create two `fi` objects, `c` and `d` with the same `numerictype` attributes:

```
T = numerictype('scaling','slopebias','slope', 2^2, 'bias', 0)
```

```
T =
```

```
DataTypeMode: Fixed-point: slope and bias scaling  
Signedness: Signed  
WordLength: 16  
Slope: 2^2  
Bias: 0
```

```
c = fi(pi, T)
```

```
c =
```

```
4
```

```
DataTypeMode: Fixed-point: slope and bias scaling  
Signedness: Signed  
WordLength: 16  
Slope: 2^2  
Bias: 0
```

```
d = fi(pi/2, T)
```

d =

0

DataTypeMode: Fixed-point: slope and bias scaling
Signedness: Signed
WordLength: 16
Slope: 2²
Bias: 0

numerictype Object Functions

You can learn about the functions associated with `numerictype` objects in the Function Reference.

Working with quantizer Objects

- “Constructing quantizer Objects” on page 7-2
- “quantizer Object Properties” on page 7-3
- “Quantizing Data with quantizer Objects” on page 7-4
- “Transformations for Quantized Data” on page 7-6
- “quantizer Object Functions” on page 7-7

Constructing quantizer Objects

You can use quantizer objects to quantize data sets. You can create quantizer objects in Fixed-Point Toolbox software in one of two ways:

- You can use the `quantizer` constructor function to create a new object.
- You can use the `quantizer` constructor function to copy a quantizer object.

To create a quantizer object with default properties, type

```
q = quantizer

q =

    DataMode = fixed
    RoundMode = floor
    OverflowMode = saturate
    Format = [16 15]
```

To copy a quantizer object, simply use assignment as in the following example:

```
q = quantizer;
r = q;
isequal(q,r)

ans =

    1
```

A listing of all the properties of the quantizer object `q` you just created is displayed along with the associated property values. All property values are set to defaults when you construct a quantizer object this way. See “quantizer Object Properties” on page 7-3 for more details.

quantizer Object Properties

The following properties of quantizer objects are always writable:

- `DataMode` — Type of arithmetic used in quantization
- `Format` — Data format of a quantizer object
- `OverflowMode` — Overflow-handling mode
- `RoundMode` — Rounding mode

See the Property Reference for more details about these properties, including their possible values.

For example, to create a fixed-point quantizer object with

- The `Format` property value set to `[16,14]`
- The `OverflowMode` property value set to `'saturate'`
- The `RoundMode` property value set to `'ceil'`

type

```
q = quantizer('datamode','fixed','format',[16,14],...  
             'overflowmode','saturate','roundmode','ceil')
```

You do not have to include quantizer object property names when you set quantizer object property values.

For example, you can create quantizer object `q` from the previous example by typing

```
q = quantizer('fixed',[16,14],'saturate','ceil')
```

Note You do not have to include default property values when you construct a quantizer object. In this example, you could leave out `'fixed'` and `'saturate'`.

Quantizing Data with quantizer Objects

You construct a `quantizer` object to specify the quantization parameters to use when you quantize data sets. You can use the `quantize` function to quantize data according to a `quantizer` object's specifications.

Once you quantize data with a `quantizer` object, its state values might change.

The following example shows

- How you use `quantize` to quantize data
- How quantization affects `quantizer` object states
- How you reset `quantizer` object states to their default values using `reset`

1 Construct an example data set and a `quantizer` object.

```
format long g
randn('state',0);
x = randn(100,4);
q = quantizer([16,14]);
```

2 Retrieve the values of the `maxlog` and `noverflows` states.

```
q.maxlog
ans =
    -1.79769313486232e+308

q.noverflows
ans =
    0
```

Note that `maxlog` is equal to `-realmax`, which indicates that the `quantizer` `q` is in a reset state.

3 Quantize the data set according to the `quantizer` object's specifications.

```
y = quantize(q,x);  
Warning: 15 overflows.
```

4 Check the values of `maxlog` and `noverflows`.

```
q.maxlog  
  
ans =  
  
1.99993896484375  
  
q.noverflows  
  
ans =  
  
15
```

Note that the maximum logged value was taken after quantization, that is, `q.maxlog == max(y)`.

5 Reset the quantizer states and check them.

```
reset(q)  
q.maxlog  
  
ans =  
  
-1.79769313486232e+308  
  
q.noverflows  
  
ans =  
  
0
```

Transformations for Quantized Data

You can convert data values from numeric to hexadecimal or binary according to a quantizer object's specifications.

Use

- `num2bin` to convert data to binary
- `num2hex` to convert data to hexadecimal
- `hex2num` to convert hexadecimal data to numeric
- `bin2num` to convert binary data to numeric

For example,

```
q = quantizer([3 2]);  
x = [0.75  -0.25  
      0.50  -0.50  
      0.25  -0.75  
      0     -1  ];  
b = num2bin(q,x)
```

```
b =  
011  
010  
001  
000  
111  
110  
101  
100
```

produces all two's complement fractional representations of 3-bit fixed-point numbers.

quantizer Object Functions

You can learn about the functions associated with `quantizer` objects in the [Function Reference](#).

Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms

- “What Are Code Acceleration and Code Generation from MATLAB?” on page 8-3
- “Requirements for Generating MEX Files from MATLAB Algorithms” on page 8-4
- “Functions Supported for Code Acceleration and Code Generation from MATLAB” on page 8-5
- “Workflow for Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms” on page 8-15
- “Setting Up a Supported C Compiler to Generate MEX Functions” on page 8-16
- “Using fiaccel” on page 8-17
- “Setting Up File Infrastructure and Paths” on page 8-22
- “Preparing MATLAB Algorithms for Code Generation” on page 8-25
- “Setting MEX Compilation Options” on page 8-28
- “Specifying Properties of Primary Function Inputs” on page 8-36
- “Best Practices for Accelerating Fixed-Point MATLAB Code” on page 8-49
- “Working with Fixed-Point Code Generation Reports” on page 8-53
- “Generating MEX Functions from MATLAB Code That Uses Global Data” on page 8-58

- “Defining Input Properties Programmatically in the MATLAB File” on page 8-64
- “Controlling Run-Time Checks” on page 8-73
- “MATLAB® Coder” on page 8-76
- “MATLAB Function Block” on page 8-77

What Are Code Acceleration and Code Generation from MATLAB?

In many cases, you may want your code to run faster and more efficiently. *Code acceleration* provides optimizations for accelerating fixed-point algorithms through MEX file generation. In Fixed-Point Toolbox the `fiaccel` function converts your MATLAB code to a MEX function and can greatly accelerate the execution speed of your fixed-point algorithms.

Code generation creates efficient, production-quality C/C++ code for desktop and embedded applications. There are several ways to use Fixed-Point Toolbox software to generate C/C++ code.

Use...	To...	Requires...	See...
MATLAB® Coder™ (codegen) function	Automatically convert MATLAB code to C/C++ code	MATLAB Coder code generation software license	“Tutorial: Generating C Code from MATLAB Code at the Command Line” in the MATLAB Coder documentation
MATLAB Function	Use MATLAB code in your Simulink models that generate embeddable C/C++ code	Simulink license	“Using the MATLAB Function Block” in the Simulink documentation

MATLAB code generation supports variable-size arrays and matrices with known upper bounds. To learn more about using variable-size signals, see “What Is Variable-Size Data?” in the Code Generation for MATLAB documentation.

Requirements for Generating MEX Files from MATLAB Algorithms

You use the `fiaccel` function to generate MEX code from a MATLAB algorithm. The algorithm must meet these requirements:

- Must be a MATLAB function, not a script
- Must meet the requirements listed on the `fiaccel` reference page
- Does not call custom C code using any of the following constructs:
 - `coder.ceval`
 - `coder.ref`
 - `coder.rref`
 - `coder.wref`

Functions Supported for Code Acceleration and Code Generation from MATLAB

In addition to any function-specific limitations listed in the table, the following general limitations always apply to the use of Fixed-Point Toolbox functions in generated code or with `fiaccel`:

- `fipref` and quantizer objects are not supported.
- Dot notation is only supported for getting the values of `fimath` and `numericType` properties. Dot notation is not supported for `fi` objects, and it is not supported for setting properties.
- Word lengths greater than 128 bits are not supported.
- You cannot change the `fimath` or `numericType` of a given variable after that variable has been created.
- The boolean and `ScaledDouble` values of the `DataTypeMode` and `DataType` properties are not supported.
- For all `SumMode` property settings other than `FullPrecision`, the `CastBeforeSum` property must be set to `true`.
- The `numel` function returns the number of elements of `fi` objects in the generated code.
- When you compile code containing `fi` objects with nontrivial slope and bias scaling, you may see different results in generated code than you achieve by running the same code in MATLAB.
- All general limitations of C/C++ code generated from MATLAB apply. See “MATLAB Language Features Not Supported for Code Generation” for more information.

Function	Remarks/Limitations
<code>abs</code>	N/A
<code>add</code>	N/A
<code>all</code>	N/A
<code>any</code>	N/A

Function	Remarks/Limitations
bitand	Not supported for slope-bias scaled <code>fi</code> objects.
bitandreduce	N/A
bitcmp	N/A
bitconcat	N/A
bitget	N/A
bitor	Not supported for slope-bias scaled <code>fi</code> objects.
bitorreduce	N/A
bitreplicate	N/A
bitrol	N/A
bitror	N/A
bitset	N/A
bitshift	N/A
bitsliceget	N/A
bitsll	N/A
bitsra	N/A
bitsrl	N/A
bitxor	Not supported for slope-bias scaled <code>fi</code> objects.
bitxorreduce	N/A
ceil	N/A
complex	N/A
conj	N/A

Function	Remarks/Limitations
conv	<ul style="list-style-type: none"> • Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB. • For variable-sized signals, you may see different results between generated code and MATLAB. <ul style="list-style-type: none"> ▪ In the generated code, the output for variable-sized signals is always computed using the SumMode property of the governing fimath. ▪ In MATLAB, the output for variable-sized signals is computed using the SumMode property of the governing fimath when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the ProductMode of the governing fimath.
convergent	N/A
cordicexp	Variable-size signals are not supported.
cordiccos	Variable-size signals are not supported.
cordicpol2cart	Variable-size signals are not supported.
cordicrotate	Variable-size signals are not supported.
cordicsin	Variable-size signals are not supported.
cordicsincos	Variable-size signals are not supported.
ctranspose	N/A
diag	If supplied, the index, k , must be a real and scalar integer value that is not a <code>fi</code> object.
disp	—

Function	Remarks/Limitations
divide	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. Complex and imaginary divisors are not supported. Code generation in MATLAB does not support the syntax <code>T.divide(a,b)</code>.
double	N/A
end	N/A
eps	<ul style="list-style-type: none"> Supported for scalar fixed-point signals only. Supported for scalar, vector, and matrix, <code>fi</code> single and <code>fi</code> double signals.
eq	Not supported for fixed-point signals with different biases.
fi	<ul style="list-style-type: none"> Use to create a fixed-point constant or variable in the generated code. The default constructor syntax without any input arguments is not supported. The syntax <code>fi('PropertyName',PropertyValue...)</code> is not supported. To use property name/property value pairs, you must first specify the value <code>v</code> of the <code>fi</code> object as in <code>fi(v,'PropertyName',PropertyValue...)</code>. Works for all input values when complete <code>numericType</code> information of the <code>fi</code> object is provided. Works only for constant input values (value of input must be known at compile time) when complete <code>numericType</code> information of the <code>fi</code> object is not specified.

Function	Remarks/Limitations
	<ul style="list-style-type: none"> • <code>numericType</code> object information must be available for nonfixed-point Simulink inputs.
<code>filter</code>	<ul style="list-style-type: none"> • Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>.
<code>fimath</code>	<ul style="list-style-type: none"> • Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a <code>fimath</code> object. You define this object in the MATLAB Function block dialog in the Model Explorer. • Use to create <code>fimath</code> objects in the generated code.
<code>fix</code>	N/A
<code>floor</code>	N/A
<code>ge</code>	Not supported for fixed-point signals with different biases.
<code>get</code>	The syntax <code>structure = get(o)</code> is not supported.
<code>getlsb</code>	N/A
<code>getmsb</code>	N/A
<code>gt</code>	Not supported for fixed-point signals with different biases.
<code>horzcat</code>	N/A
<code>imag</code>	N/A
<code>int, int8, int16, int32</code>	N/A
<code>iscolumn</code>	N/A
<code>isempty</code>	N/A
<code>isequal</code>	N/A
<code>isfi</code>	N/A
<code>isfimath</code>	N/A
<code>isfimathlocal</code>	N/A

Function	Remarks/Limitations
<code>isfinite</code>	N/A
<code>isinf</code>	N/A
<code>isnan</code>	N/A
<code>isnumeric</code>	N/A
<code>isnumerictype</code>	N/A
<code>isreal</code>	N/A
<code>isrow</code>	N/A
<code>isscalar</code>	N/A
<code>assigned</code>	N/A
<code>isvector</code>	N/A
<code>le</code>	Not supported for fixed-point signals with different biases.
<code>length</code>	N/A
<code>logical</code>	N/A
<code>lowerbound</code>	N/A
<code>lsb</code>	<ul style="list-style-type: none"> • Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, <code>fi</code> single and double signals.
<code>lt</code>	Not supported for fixed-point signals with different biases.
<code>max</code>	N/A
<code>mean</code>	N/A
<code>median</code>	N/A
<code>min</code>	N/A
<code>minus</code>	Any non- <code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.

Function	Remarks/Limitations
mpower	<ul style="list-style-type: none"> • The exponent input, k, must be constant; that is, its value must be known at compile time. • Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. • For variable-sized signals, you may see different results between the generated code and MATLAB. <ul style="list-style-type: none"> ▪ In the generated code, the output for variable-sized signals is always computed using the <code>SumMode</code> property of the governing <code>fimath</code>. ▪ In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fimath</code> when the first input, a, is nonscalar. However, when a is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fimath</code>.
mpy	When you provide complex inputs to the <code>mpy</code> function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to <code>0n</code> .
mrdivide	N/A
mtimes	<ul style="list-style-type: none"> • Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. • Variable-sized inputs are only supported when the <code>SumMode</code> property of the governing <code>fimath</code> is set to <code>Specify precision</code> or <code>Keep LSB</code>. • For variable-sized signals, you may see different results between the generated code and MATLAB.

Function	Remarks/Limitations
	<ul style="list-style-type: none"> ▪ In the generated code, the output for variable-sized signals is always computed using the <code>SumMode</code> property of the governing <code>fi</code> math. ▪ In MATLAB, the output for variable-sized signals is computed using the <code>SumMode</code> property of the governing <code>fi</code> math when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the <code>ProductMode</code> of the governing <code>fi</code> math.
ndims	N/A
ne	Not supported for fixed-point signals with different biases.
nearest	N/A
numberofelements	numberofelements and numel both work the same as MATLAB numel for <code>fi</code> objects in the generated code.
numerictype	<ul style="list-style-type: none"> • Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a <code>numerictype</code> object that is populated with the signal's data type and scaling information. • Returns the data type when the input is a nonfixed-point signal. • Use to create <code>numerictype</code> objects in generated code.
permute	N/A
plus	Any non- <code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object.
pow2	N/A
power	The exponent input, k , must be constant; that is, its value must be known at compile time.
range	N/A

Function	Remarks/Limitations
rdivide	N/A
real	N/A
realmax	N/A
realmin	N/A
reinterpretpcast	N/A
repmat	N/A
rescale	N/A
reshape	N/A
round	N/A
sfi	N/A
sign	N/A
single	N/A
size	N/A
sort	N/A
sqrt	<ul style="list-style-type: none"> • Complex and [Slope Bias] inputs error out. • Negative inputs yield a 0 result.
sub	N/A
subsasgn	N/A
subsref	N/A
sum	Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.

Function	Remarks/Limitations
times	<ul style="list-style-type: none"> Any non-<code>fi</code> input must be constant; that is, its value must be known at compile time so that it can be cast to a <code>fi</code> object. When you provide complex inputs to the <code>times</code> function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Ports and data manager and set the Complexity parameter for all known complex inputs to <code>0n</code>.
transpose	N/A
tril	If supplied, the index, k , must be a real and scalar integer value that is not a <code>fi</code> object.
triu	If supplied, the index, k , must be a real and scalar integer value that is not a <code>fi</code> object.
ufi	N/A
uint8, uint16, uint32	N/A
uminus	N/A
uplus	N/A
upperbound	N/A
vertcat	N/A

Workflow for Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms

Step	Action	Details
1	Set up your C compiler.	See “Setting Up a Supported C Compiler to Generate MEX Functions” on page 8-16
2	Set up your file infrastructure.	See “Setting Up File Infrastructure and Paths” on page 8-22.
3	Make your MATLAB algorithm suitable for code generation	See “Best Practices for Accelerating Fixed-Point MATLAB Code” on page 8-49.
4	Set compilation options.	See “Setting MEX Compilation Options” on page 8-28
5	Specify properties of primary function inputs.	See “Specifying Properties of Primary Function Inputs” on page 8-36.
6	Run <code>fiaccel</code> with the appropriate command-line options.	See “Recommended Compilation Options for <code>fiaccel</code> ” on page 8-49

Setting Up a Supported C Compiler to Generate MEX Functions

Set up your C compiler by running `mex -setup`, as described in the documentation for `mex` in the MATLAB Function Reference. You must run this command even if you use the default C compiler that comes with MATLAB for Windows® platforms. You can also use `mex` to choose and configure a different C compiler, as described in “What You Need to Build MEX-Files” in the MATLAB External Interfaces documentation.

You can use the following compilers to generate MEX functions with `fiaccel`:

- Lcc-win32 C 2.4.1
- Microsoft® Visual C++® 2008 Express
- Microsoft Visual C++ 2005
- Microsoft Visual C++ 6.0
- Open WATCOM C++ 1.7
- GCC

Using fiaccel

In this section...

“Speeding Up Fixed-Point Execution with the fiaccel Function” on page 8-17

“Running fiaccel” on page 8-17

“Generated Files and Locations” on page 8-18

“Using Data Type Override with fiaccel” on page 8-21

Speeding Up Fixed-Point Execution with the fiaccel Function

You can convert fixed-point MATLAB code to MEX functions using `fiaccel`. The generated MEX functions contain optimizations to automatically accelerate fixed-point algorithms to compiled C/C++ code speed in MATLAB. The `fiaccel` function can greatly increase the execution speed of your algorithms.

Running fiaccel

The basic command is:

```
fiaccel M_fcn
```

By default, `fiaccel` performs the following actions:

- Searches for the function `M_fcn` stored in the file `M_fcn.m` as specified in “Compile Path Search Order” on page 8-22.
- Compiles `M_fcn` to MEX code.
- If there are no errors or warnings, generates a platform-specific MEX file in the current folder, using the naming conventions described in “File Naming Conventions” on page 8-52.
- If there are errors, does not generate a MEX file, but produces an error report in a default output folder, as described in “Generated Files and Locations” on page 8-18.

- If there are warnings, but no errors, generates a platform-specific MEX file in the current folder, but does report the warnings.

You can modify this default behavior by specifying one or more compiler options with `fiaccel`, separated by spaces on the command line.

Generated Files and Locations

`fiaccel` generates files in the following locations:

Generates:	In:
Platform-specific MEX files	Current folder
HTML reports (if errors or warnings occur during compilation)	Default output folder: <code>fiaccel/mex/M_fcn_name/html</code>

You can change the name and location of generated files by using the options `-o` and `-d` when you run `fiaccel`.

In this example, you will use the `fiaccel` function to compile different parts of a simple algorithm. By comparing the run times of the two cases, you will see the benefits and best use of the `fiaccel` function.

Example: Comparing Run Times When Accelerating Different Algorithm Parts

The algorithm used throughout this example replicates the functionality of the MATLAB `sum` function, which sums the columns of a matrix. To see the algorithm, type `open fi_matrix_column_sum.m` at the MATLAB command line.

```
function B = fi_matrix_column_sum(A)
% Sum the columns of matrix A.
%#codegen
    [m,n] = size(A);
    w = get(A,'WordLength') + ceil(log2(m));
    f = get(A,'FractionLength');
    B = fi(zeros(1,n),true,w,f);
```

```
for j = 1:n
    for i = 1:m
        B(j) = B(j) + A(i,j);
    end
end
```

Trial 1: Best Performance

The best way to speed up the execution of the algorithm is to compile the entire algorithm using the `fiaccel` function. To evaluate the performance improvement provided by the `fiaccel` function when the entire algorithm is compiled, run the following code.

The first portion of code executes the algorithm using only MATLAB functions. The second portion of the code compiles the entire algorithm using the `fiaccel` function. The MATLAB `tic` and `toc` functions keep track of the run times for each method of execution.

```
% MATLAB
fipref('NumericTypeDisplay','short');
A = fi(randn(1000,10));
tic
B = fi_matrix_column_sum(A)
t_matrix_column_sum_m = toc

% fiaccel
fiaccel fi_matrix_column_sum -args {A} ...
-I [matlabroot '/toolbox/fixedpoint/fidemos']
tic
B = fi_matrix_column_sum_mex(A);
t_matrix_column_sum_mex = toc
```

Trial 2: Worst Performance

Compiling only the smallest unit of computation using the `fiaccel` function leads to much slower execution. In some cases, the overhead that results from calling the `mex` function inside a nested loop can cause even slower execution than using MATLAB functions alone. To evaluate the performance of the `mex` function when only the smallest unit of computation is compiled, run the following code.

The first portion of code executes the algorithm using only MATLAB functions. The second portion of the code compiles the smallest unit of computation with the `fiaccel` function, leaving the rest of the computations to MATLAB.

```
% MATLAB
tic
[m,n] = size(A);
w = get(A,'WordLength') + ceil(log2(m));
f = get(A,'FractionLength');
B = fi(zeros(1,n),true,w,f);
for j = 1:n
    for i = 1:m
        B(j) = fi_scalar_sum(B(j),A(i,j));
        % B(j) = B(j) + A(i,j);
    end
end
t_scalar_sum_m = toc

% fiaccel
fiaccel fi_scalar_sum -args {B(1),A(1,1)} ...
-I [matlabroot '/toolbox/fixedpoint/fidemos']
tic
[m,n] = size(A);
w = get(A,'WordLength') + ceil(log2(m));
f = get(A,'FractionLength');
B = fi(zeros(1,n),true,w,f);
for j = 1:n
    for i = 1:m
        B(j) = fi_scalar_sum_mex(B(j),A(i,j));
        % B(j) = B(j) + A(i,j);
    end
end
t_scalar_sum_mex = toc
```

Ratio of Times

A comparison of Trial 1 and Trial 2 appears in the following table. Your computer may record different times than the ones the table shows, but the ratios should be approximately the same. There is an extreme difference in ratios between the trial where the entire algorithm was compiled using

fiaccel (t_matrix_column_sum_mex.m) and where only the scalar sum was compiled (t_scalar_sum_mex.m). Even the file with no fiaccel compilation (t_matrix_column_sum_m) did better than when only the smallest unit of computation was compiled using fiaccel (t_scalar_sum_mex).

X (Overall Performance Rank)	Time	X/Best	X_m/X_{mex}
Trial 1: Best Performance			
t_matrix_column_sum_m (2)	1.99759	84.4917	84.4917
t_matrix_column_sum_mex (1)	0.0236424	1	
Trial 2: Worst Performance			
t_scalar_sum_m (4)	10.2067	431.71	2.08017
t_scalar_sum_mex (3)	4.90664	207.536	

Using Data Type Override with fiaccel

Fixed-Point Toolbox software ships with a demonstration of how to generate a MEX function from MATLAB code. The code in the demo takes the weighted average of a signal to create a lowpass filter. To run the demo in the Help browser select Demos under Fixed-Point Toolbox, and then select the Fixed-Point Lowpass Filtering Using MATLAB for Code Generation demo.

You can specify data type override in this demo by typing an extra command at the MATLAB prompt in the “Define Fixed-Point Parameters” section of the demo. To turn data type override on, type the following command at the MATLAB prompt after running the `reset(fipref)` demo command in that section:

```
fipref('DataTypeOverride','TrueDoubles')
```

This command tells Fixed-Point Toolbox software to create all `fi` objects with type `fi double`. When you compile the code using the `fiaccel` command in the “Compile the M-File into a MEX File” section of the demo, the resulting MEX-function uses floating-point data.

Setting Up File Infrastructure and Paths

In this section...

“Compile Path Search Order” on page 8-22

“When to Use the Code Generation Path” on page 8-23

“Add Files to the Code Generation Path” on page 8-23

“Adding Folders to Search Paths” on page 8-23

“Naming Conventions” on page 8-23

Compile Path Search Order

`fiaccl` resolves function calls by searching first on the code generation path and then on the MATLAB path. By default, `fiaccl` tries to compile and generate code for functions it finds on the path unless you explicitly declare the function to be extrinsic. An *extrinsic function* is a function on the MATLAB path that is dispatched to MATLAB software for execution. `fiaccl` does not compile extrinsic functions, but rather dispatches them to MATLAB for execution.

When to Use the Code Generation Path

Use the code generation path to override a MATLAB function with a customized version. Because `fiaccl` searches the code generation path first, a MATLAB file on that path always shadows a MATLAB file of the same name on the MATLAB path.

To override a MATLAB function with a customized version:

- 1 Create each version of the MATLAB function in identically named files.
- 2 Add the MATLAB version to the MATLAB path.
- 3 Add the customized version to the code generation path.

See “Adding Folders to Search Paths” on page 8-23.

Add Files to the Code Generation Path

With `fiaccel`, you can prepend folders and files to the code generation path, as described in “Adding Folders to Search Paths” on page 8-23. By default, the code generation path contains the current folder and the toolbox functions supported for code generation.

Adding Folders to Search Paths

To add folders to:	Do this:
Code generation path	Prepend folders to the code generation path by using the <code>fiaccel -I</code> option.
MATLAB path	Follow the instructions in “Adding a Folder to the Search Path” in the MATLAB Programming documentation.

Naming Conventions

MATLAB enforces naming conventions for functions and generated files.

- “Reserved Prefixes” on page 8-23
- “Conventions for Naming Generated files” on page 8-23

Reserved Prefixes

MATLAB reserves the prefix `eml` for global C functions and variables in generated code. For example, run-time library function names all begin with the prefix `emlrt`, such as `emlrtCallMATLAB`. To avoid naming conflicts, do not name C functions or primary MATLAB functions with the prefix `eml`.

Conventions for Naming Generated files

MATLAB provides platform-specific extensions for MEX files.

Platform	MEX File Extension
Linux® (32-bit)	<code>.mexglx</code>
Linux x86-64	<code>.mexa64</code>

Platform	MEX File Extension
Windows (32-bit)	.mexw32
Windows x64	.mexw64

Preparing MATLAB Algorithms for Code Generation

In this section...

“Debugging Strategies” on page 8-25

“Detecting Errors at Design Time” on page 8-26

“Detecting Errors at Compile Time” on page 8-26

Debugging Strategies

To prepare your algorithms for code generation, MathWorks recommends that you choose a debugging strategy for detecting and correcting violations in your MATLAB applications, especially if they consist of a large number of MATLAB files that call each other’s functions. Here are two best practices:

Debugging Strategy	What to Do	Pros	Cons
Bottom-up verification	<ol style="list-style-type: none"> 1 Verify that your lowest-level (leaf) functions are suitable for code generation. 2 Work your way up the function hierarchy incrementally to compile and verify each function, ending with the top-level function. 	<ul style="list-style-type: none"> • Efficient • Safe • Easy to isolate syntax violations 	Requires application tests that work from the bottom up

Debugging Strategy	What to Do	Pros	Cons
Top-down verification	<ol style="list-style-type: none"> 1 Declare all functions called by the top-level function to be extrinsic so <code>fiaccl</code> does not compile them. 2 Verify that your top-level function is suitable for code generation. 3 Work downward in the function hierarchy to: <ol style="list-style-type: none"> a. Remove extrinsic declarations one by one b. Compile and verify each function, ending with the leaf functions. 	Lets you retain your top-level tests	Introduces extraneous code that you must remove after code verification, including: <ul style="list-style-type: none"> • Extrinsic declarations • Additional assignment statements as necessary to convert opaque values returned by extrinsic functions to nonopaque values.

Detecting Errors at Design Time

To detect potential issues for MEX generation as you write your MATLAB algorithm, add the `%#codegen` directive to the code that you want `fiaccl` to compile. Adding this directive indicates that you intend to generate code from the algorithm and turns on detailed diagnostics during MATLAB code analysis (see “Code Analysis Options” in the MATLAB Desktop Tools and Development Environment documentation).

Detecting Errors at Compile Time

Before you can successfully generate code from a MATLAB algorithm, you must verify that the algorithm does not contain syntax and semantics violations that would cause compile-time errors, as described in “Preparing MATLAB Algorithms for Code Generation” on page 8-25.

`fiaccl` checks for all potential syntax violations at compile time. When `fiaccl` detects errors or warnings, it automatically produces a code generation report that describes the issues and provides links to the offending code. See “Working with Fixed-Point Code Generation Reports” on page 8-53.

If your MATLAB code calls functions on the MATLAB path, `fiaccl` attempts to compile these functions unless you declare them to be extrinsic.

Setting MEX Compilation Options

In this section...

“Working with the MEX Compiler Configuration Object” on page 8-28

“Modifying Compilation Options at the Command Line Using Dot Notation” on page 8-28

“MEX Compiler Options Parameters” on page 8-29

“How fiaccel Resolves Conflicting Options” on page 8-35

Working with the MEX Compiler Configuration Object

For MEX code generation, MATLAB provides a configuration object `coder.MEXConfig` for fine-tuning the compilation. To set MEX compilation options:

- 1 Define the compiler configuration object in the MATLAB workspace by issuing a constructor command:

```
comp_cfg = coder.mexconfig
```

MATLAB displays the list of compiler options and their current values in the command window.

- 2 Modify the compilation options as necessary. See “Modifying Compilation Options at the Command Line Using Dot Notation” on page 8-28
- 3 Invoke `fiaccel` with the `-config` option and specify the configuration object as its argument:

```
fiaccel -config comp_cfg myMfile
```

The `-config` option instructs `fiaccel` to convert `myFile.m` to a MEX function, based on the compilation settings in `comp_cfg`.

Modifying Compilation Options at the Command Line Using Dot Notation

Use dot notation to modify the value of compilation options, using this syntax:

```
configuration_object.property = value
```

Dot notation uses assignment statements to modify configuration object properties. For example, to change the maximum size function to inline and the stack size limit for inlined functions during MEX generation, enter this code at the command line:

```
co_cfg = coder.MEXConfig
co_cfg.InlineThreshold = 25;
co_cfg.InlineStackLimit = 4096;
fiaccl -config co_cfg myFun
```

MEX Compiler Options Parameters

The following table describes parameters for fine-tuning the behavior of the compiler for converting MATLAB files to MEX:

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Report		
Create code generation report	GenerateReport true, false	Document generated code in an HTML report.
Launch report automatically	LaunchReport true, false	Specify whether to automatically display HTML reports after code generation completes. Note Requires that you enable Create code generation report
Debugging		
Echo expressions without semicolons	EchoExpressions true , false	Specify whether or not actions that do not terminate with a semicolon appear in the MATLAB Command Window.

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Enable debug build	EnableDebugging true, false	Compile the generated code in debug mode.
Language and Semantics		
Constant Folding Timeout	ConstantFoldingTimeout <i>integer</i> , 10000	Specify the maximum number of instructions to be executed by the constant folder.

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Dynamic memory allocation	DynamicMemoryAllocation off , 'AllVariableSizeArrays'	Enable dynamic memory allocation for variable-size data. By default, dynamic memory allocation is disabled and <code>fiaccel</code> allocates memory statically on the stack. When you select dynamic memory allocation, <code>fiaccel</code> allocates memory for all variable-size data dynamically on the heap. You <i>must</i> use dynamic memory allocation for all unbounded variable-size data.
Enable variable sizing	EnableVariableSizing true , false	Enable support for variable-size arrays.

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Extrinsic calls	ExtrinsicCalls true , false	<p>Allow calls to extrinsic functions.</p> <p>When enabled (true), the compiler generates code for the call to a MATLAB function, but does not generate the function's internal code.</p> <p>When disabled (false), the compiler ignores the extrinsic function. Does not generate code for the call to the MATLAB function—as long as the extrinsic function does not affect the output of the caller function. Otherwise, the compiler issues a compiler error.</p>

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Global Data Synchronization Mode	GlobalDataSyncMethod <i>string</i> , SyncAlways , SyncAtEntryAndExits, NoSync	<p>Controls when global data is synchronized with the MATLAB global workspace. By default, (SyncAlways), synchronizes global data at MEX function entry and exit and for all extrinsic calls. This synchronization ensures maximum consistency between MATLAB and generated code. If the extrinsic calls do not affect global data, use this option with the <code>coder.extrinsic -sync:off</code> option to turn off synchronization for these calls.</p> <p>SyncAtEntryAndExits synchronizes global data at MEX function entry and exit only. If only a few extrinsic calls affect global data, use this option with the <code>coder.extrinsic -sync:on</code> option to turn on synchronization for these calls.</p> <p>NoSync disables synchronization. Ensure that your generated code does not interact with MATLAB before disabling synchronization. Otherwise, inconsistencies might occur.</p>

Parameter	Equivalent Command-Line Property and Values (default in bold)	Description
Saturate on integer overflow	SaturateOnIntegerOverflow true , false	Add checks in the generated code to detect integer overflow or underflow.
Safety (disable for faster MEX)		
Ensure memory integrity	IntegrityChecks true , false	Detects violations of memory integrity in code generated from MATLAB algorithms and stops execution with a diagnostic message. Setting IntegrityChecks to false also disables the run-time stack.
Ensure responsiveness	ResponsivenessChecks true , false	Enables responsiveness checks in code generated from MATLAB algorithms.
Function Inlining and Stack Allocation		
Inline Stack Limit	InlineStackLimit <i>integer</i> , 4000	Specify the stack size limit on inlined functions.
Inline Threshold	InlineThreshold <i>integer</i> , 10	Specify the maximum size of functions to be inlined.
Inline Threshold Max	InlineThresholdMax <i>integer</i> , 200	Specify the maximum size of functions after inlining.
Stack Usage Max	StackUsageMax <i>integer</i> , 200000	Specify the maximum stack usage per function.
Optimizations		
Use BLAS library if possible	EnableBLAS true , false	Speed up low-level matrix operations during simulation by calling the Basic Linear Algebra Subprograms (BLAS) library.

See Also

- “Controlling Run-Time Checks” on page 8-73
- “How Working with Variable-Size Data is Different for Code Generation”
- “When to Use Dynamic Allocation for Variable-Size Data”
- “Generating MEX Functions from MATLAB Code That Uses Global Data” on page 8-58

How `fiaccl` Resolves Conflicting Options

`fiaccl` takes the union of all options, including those specified using configuration objects, so that you can specify options in any order.

Specifying Properties of Primary Function Inputs

In this section...
“Why You Must Specify Input Properties” on page 8-36
“Properties to Specify” on page 8-36
“Rules for Specifying Properties of Primary Inputs” on page 8-40
“Methods for Defining Properties of Primary Inputs” on page 8-40
“Defining Input Properties by Example at the Command Line” on page 8-41

Why You Must Specify Input Properties

To generate code in a statically typed language, `fiaccel` must determine the properties of all variables in the MATLAB code at compile time. Therefore, you must specify the class, size, and complexity of inputs to the primary function (also known as the *top-level* or *entry-point* function). If your primary function has no input parameters, `fiaccel` can compile your MATLAB algorithm without modification. You do not need to specify properties of inputs to subfunctions or external functions called by the primary function. For `fiaccel` requirements, refer to its reference page.

Properties to Specify

If your primary function has inputs, you must specify the following properties for each input:

For:	Specify Properties:				
	Class	Size	Complexity	numerictype	fimath
Fixed-point inputs	✓	✓	✓	✓	✓

For:	Specify Properties:				
	Class	Size	Complexity	numerictype	fimath
Structure inputs*	✓	✓	✓	✓ (if structure field is fixed-point)	✓ (if structure field is fixed-point)
All other inputs	✓	✓	✓		

* When a primary input is a structure, `fiaccl` treats each field as a separate input.

Default Property Values

`fiaccl` assigns the following default values for properties of primary function inputs:

Property	Default
class	double
size	scalar
complexity	real
numerictype	No default
fimath	MATLAB default <code>fimath</code> object

Specifying Default Values for Structure Fields. In most cases, `fiaccl` uses defaults when you don't explicitly specify values for properties—except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore, you may need to specify default values for properties of structure fields. For examples, see “Example: Specifying Class and Size of Scalar Structure” on page 8-70 and “Example: Specifying Class and Size of Structure Array” on page 8-71.

Specifying Default `fimath` Values for MEX Functions. MEX functions generated with `fiaccel` use the MATLAB default global `fimath`. The MATLAB factory default global `fimath` has the following properties:

```
RoundMode: nearest
OverflowMode: saturate
ProductMode: FullPrecision
MaxProductWordLength: 128
SumMode: FullPrecision
MaxSumWordLength: 128
```

For more information, see “Working with the Global `fimath`” on page 4-6.

When running MEX functions that depend on the MATLAB default `fimath` value, do not change this value during your MATLAB session. Otherwise, you receive a run-time error, alerting you to a mismatch between the compile-time and run-time `fimath` values.

For example, suppose you define the following MATLAB function `test`:

```
function y = test %#codegen
y = fi(0);
```

The function `test` constructs a `fi` object without explicitly specifying a `fimath` object. Therefore, `test` will rely on the default `fimath` object in effect at compile time. At the MATLAB prompt, generate the MEX function `test_mex` to use the factory setting of the MATLAB default `fimath`:

```
resetglobalfimath;
fiaccel test
% fiaccel generates a MEX function, test_mex,
% in the current folder
```

Next, run `test_mex` to display the MATLAB default `fimath` value:

```
test_mex

ans =

    0
```

```

        DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
        FractionLength: 15

```

Now, modify the MATLAB default `fimath` value so it no longer matches the factory setting used at compile time:

```

F = fimath('RoundMode','Floor');
globalfimath(F);

```

Finally, clear the MEX function from memory and rerun it:

```

clear test_mex
test_mex

```

The mismatch is detected and causes an error:

```

??? This function was generated with a different default
fimath than the current default.

```

```

Error in ==> test_mex

```

Supported Classes

The following table presents the class names supported by `fiaccl`:

Class Name	Description
<code>logical</code>	Logical array of true and false values
<code>char</code>	Character array
<code>int8</code>	8-bit signed integer array
<code>uint8</code>	8-bit unsigned integer array
<code>int16</code>	16-bit signed integer array
<code>uint16</code>	16-bit unsigned integer array
<code>int32</code>	32-bit signed integer array
<code>uint32</code>	32-bit unsigned integer array

Class Name	Description
single	Single-precision floating-point or fixed-point number array
double	Double-precision floating-point or fixed-point number array
struct	Structure array
embedded.fi	Fixed-point number array

Rules for Specifying Properties of Primary Inputs

Follow these rules when specifying the properties of primary inputs:

- For each primary function input whose class is fixed point (fi), you must specify the input's `numericType` and `fiMath` properties.
- For each primary function input whose class is `struct`, you must specify the properties of each of its fields in the order that they appear in the structure definition.

Methods for Defining Properties of Primary Inputs

You can use any of the following methods to define the properties of primary function inputs:

Method	Pros	Cons
“Defining Input Properties by Example at the Command Line” on page 8-41	<ul style="list-style-type: none"> • Easy to use • Does not alter original MATLAB code • Designed for prototyping a function that has a small number of primary inputs 	<ul style="list-style-type: none"> • Must be specified at the command line every time you invoke <code>fiaccel</code> (unless you use a script) • Not efficient for specifying memory-intensive inputs such as large structures and arrays
“Defining Input Properties Programmatically in the MATLAB File” on page 8-64	<ul style="list-style-type: none"> • Integrated with MATLAB code so you do not need to redefine properties each time you invoke <code>fiaccel</code> 	<ul style="list-style-type: none"> • Uses complex syntax

Method	Pros	Cons
	<ul style="list-style-type: none"> • Provides documentation of property specifications in the MATLAB code • Efficient for specifying memory-intensive inputs such as large structures 	

Note To specify the properties of inputs for any given primary function, use one of these methods or the other, but not both.

Defining Input Properties by Example at the Command Line

- “Command Line Option -args” on page 8-41
- “Rules for using the -args option” on page 8-43
- “Specifying Constant Inputs” on page 8-43
- “Specifying Variable-Size Inputs” on page 8-45

Command Line Option -args

`fiaccl` provides a command-line option `-args` for specifying the properties of primary function inputs as a cell array of example values. The cell array can be a variable or literal array of constant values.

Example: Specifying Properties of Primary Inputs by Example.

Consider a function that adds its two inputs:

```
function y = emcf(u,v) %#codegen
% The directive %#codegen indicates that you
% intend to generate code for this algorithm
y = u + v;
```

The following examples show how to specify different properties of the primary inputs `u` and `v` by example at the command line:

- Use a literal cell array of constants to specify that both inputs are real, scalar, fixed-point values:

```
fiaccel -o emcfx emcf ...
    -args {fi(0,1,16,15),fi(0,1,16,15)}
```

- Use a literal cell array of constants to specify that input *u* is an unsigned 16-bit, 1-by-4 vector and input *v* is a scalar, fixed-point value:

```
fiaccel -o emcfx emcf ...
    -args {zeros(1,4,'uint16'),fi(0,1,16,15)}
```

- Assign sample values to a cell array variable to specify that both inputs are real, unsigned 8-bit integer vectors:

```
a = fi([1;2;3;4],0,8,0)
b = fi([5;6;7;8],0,8,0)
ex = {a,b}
fiaccel -o emcfx emcf -args ex
```

Example: Specifying Properties of Primary Fixed-Point Inputs by

Example. Consider a function that calculates the square root of a fixed-point number:

```
function y = sqrtfi(x) %#codegen
y = sqrt(x);
```

To specify the properties of the primary fixed-point input *x* by example on the MATLAB command line, follow these steps:

- 1 Define the `numericType` properties for *x*, as in this example:

```
T = numericType('WordLength',32,...
    'FractionLength',23,'Signed',true);
```

- 2 Define the `fimath` properties for *x*, as in this example:

```
F = fimath('SumMode','SpecifyPrecision',...
    'SumWordLength',32,'SumFractionLength',23,...
    'ProductMode','SpecifyPrecision',...
    'ProductWordLength',32,'ProductFractionLength',23);
```

- 3 Create a fixed-point variable with the `numericType` and `fimath` properties you just defined, as in this example:

```
myeg = { fi(4.0,T,F) };
```

- 4 Compile the function `sqrtfi` using the `fiaccel` command, passing the variable `myeg` as the argument to the `-args` option, as in this example:

```
fiaccel sqrtfi -args myeg;
```

Rules for using the `-args` option

Follow these rules when using the `-args` command-line option to define properties by example:

- The cell array of sample values must contain the same number of elements as primary function inputs.
- The order of elements in the cell array must correspond to the order in which inputs appear in the primary function signature — for example, the first element in the cell array defines the properties of the first primary function input.

Specifying Constant Inputs

In cases where you know your primary inputs will not change at run time, you can specify them as constant values than as variables to eliminate unnecessary overhead in generated code. Common uses of constant inputs are for flags that control how an algorithm executes and values that specify the sizes or types of data.

You can define inputs to be constants using this command-line option:

```
-args {coder.Constant(constant_input)}
```

This expression specifies that an input will be a constant with the size, class, complexity, and value of *constant_input*.

Calling Functions with Constant Inputs. `fiaccl` compiles constant function inputs into the generated code. As a result, the MEX function signature differs from the MATLAB function signature. At run time you supply the constant argument to the MATLAB function, but not to the MEX function.

For example, consider the following function `identity` which copies its input to its output:

```
function y = identity(u) %#codegen
y = u;
```

To generate a MEX function `identity_mex` with a constant input, type the following command at the MATLAB prompt:

```
fiaccl -o identity_mex identity -args {coder.Constant(fi(0.1,1,16,15))}
```

To run the MATLAB function, supply the constant argument as follows:

```
identity(fi(0.1,1,16,15))
```

You get the following result:

```
ans =
    0.1000
```

Now, try running the MEX function with this command:

```
identity_mex
```

You should get the same answer.

Example: Specifying a Structure as a Constant Input. Suppose you define a structure `tmp` in the MATLAB workspace to specify the dimensions of a matrix, as follows:

```
tmp = struct('rows', 2, 'cols', 3);
```

The following MATLAB function `rowcol` accepts a structure input `p` to define matrix `y`:

```
function y = rowcol(u,p) %#codegen
```

```
y = fi(zeros(p.rows,p.cols),1,16,15) + u;
```

The following example shows how to specify that primary input `u` is a double scalar variable and primary input `p` is a constant structure:

```
fiaccel rowcol ...
    -args {fi(0,1,16,15),coder.Constant(tmp)}
```

To run this code, use

```
u = fi(0.5,1,16,15)
y_m = rowcol(u,tmp)

y_mex = rowcol_mex(u)
```

Specifying Variable-Size Inputs

Variable-size data is data whose size might change at run time. MATLAB supports bounded and unbounded variable-size data for code generation.

- *Bounded variable-size data* has fixed upper bounds; this data can be allocated statically on the stack or dynamically on the heap.
- *Unbounded variable-size data* does not have fixed upper bounds; this data must be allocated on the heap.

You can define inputs to have one or more variable-size dimensions and specify their upper bounds using the `-args` option:

Expression	Description
<code>-args {coder.typeof(example_value)}</code>	Specifies a variable-size input with the same size, class, and complexity as <i>example_value</i> . Upper bounds equal the size of each dimension specified in <i>example_value</i> . Assumes that each dimension is variable unless its upper bound is one.

Expression	Description
<code>-args {coder.typeof (<i>example_value</i>, <i>size_vector</i>)}</code>	<p>Specifies a variable-size input with:</p> <ul style="list-style-type: none">• Same class and complexity as <i>example_value</i>• Same size and upper bounds as <i>size_vector</i> <p>Assumes that each dimension is variable unless its upper bound is one.</p> <p>When you enable dynamic memory allocation, you can specify <code>Inf</code> in the size vector for dimensions with unknown upper bounds at compile time.</p>

Expression	Description
<pre>-args {coder.typeof (example_value, size_vector, dim_vector)}</pre>	<p>Specifies a variable-size input with:</p> <ul style="list-style-type: none"> • Same class and complexity as <i>example_value</i> • Same size and upper bounds as <i>size_vector</i> <p><i>dim_vector</i> specifies which dimensions are variable. A value of <code>true</code> or <code>one</code> means that the corresponding dimension is variable. A value of <code>false</code> or <code>zero</code> means that the corresponding dimension is fixed.</p> <p>The <i>dim_vector</i> value must be <code>true</code> for a dimension with unknown upper bounds (specified as <code>Inf</code> in the <i>size_vector</i>).</p>

Example: Specifying a Variable-Size Vector Input.

1 First, set the global `fimath`.

```
Foriginal = fimath; % Preserve your current fimath
F = fimath('SumMode','SpecifyPrecision',...
'SumFractionLength',24)
globalfimath(F);
```

2 Write a function that computes the sum of every `n` elements of a vector `A` and stores them in a vector `B`:

```
function B = nway(A,n) %#codegen
% Compute sum of every N elements of A and put them in B.

coder.extrinsic('error');
Tb = numerictype(1,32,24);
```

```
if ((mod(numberofelements(A),n) == 0) && ...
    (n>=1 && n<=numberofelements(A)))
    B = fi(zeros(1,numberofelements(A)/n),Tb);
    k = 1;
    for i = 1 : numberofelements(A)/n
        B(i) = sum(A(k + (0:n-1)));
        k = k + n;
    end
else
    B = fi(zeros(1,0),Tb);
    error('n<=0 or does not divide number of elements evenly');
end
```

- 3** Specify the first input A as a `fi` object. Its first dimension stays fixed in size and its second dimension can grow to an upper bound of 100. Specify the second input n as a double scalar.

```
fiaccel nway ...
    -args {coder.typeof(fi(0,1,16,15),[1 100]),0}...
    -report
```

Note You do not need to explicitly cast these inputs as `double` because `fiaccel` assumes the default properties of inputs are real, double scalars.

- 4** Reset the global `fimath`.

```
globalfimath(Foriginal)
```

- 5** As an alternative, assign the `coder.typeof` expression to a MATLAB variable, then pass the variable as an argument to `-args`:

```
vareg = coder.typeof(fi(0,1,16,15),[1 100])
fiaccel nway -args {vareg, double(0)}
```

Note For comparison, this command does explicitly cast the inputs to `double`.

Best Practices for Accelerating Fixed-Point MATLAB Code

In this section...

“Recommended Compilation Options for `fiaccel`” on page 8-49

“Using Build Scripts” on page 8-50

“Using the MATLAB Code Analyzer to Check Code Interactively at Design Time” on page 8-51

“Separating Your Test Bench from Your Function Code” on page 8-52

“Preserving Your Code” on page 8-52

“File Naming Conventions” on page 8-52

Recommended Compilation Options for `fiaccel`

- `-args` – Specify input parameters by example

Use the `-args` option to specify the properties of primary function inputs as a cell array of example values at the same time as you generate code for the MATLAB file with `fiaccel`. The cell array can be a variable or literal array of constant values. The cell array should provide the same number and order of inputs as the primary function.

When you use the `-args` option you are specifying the data types and array dimensions of these parameters, not the values of the variables. For more information, see “Defining Input Properties by Example at the Command Line” in the MATLAB Coder documentation.

Note Alternatively, you can use the `assert` function to define properties of primary function inputs directly in your MATLAB file. For more information, see “Defining Input Properties Programmatically in the MATLAB File” on page 8-64.

- `-report` – Generate code generation report

Use the `-report` option to generate a report in HTML format at code generation time to help you debug your MATLAB code and verify that it

is suitable for code generation. If you do not specify the `-report` option, `fiaccl` generates a report only if build errors or warnings occur.

The code generation report contains the following information:

- Summary of code generation results, including type of target and number of warnings or errors
- Target build log that records build and linking activities
- Links to generated files
- Error and warning messages (if any)

For more information, see `fiaccl`.

Using Build Scripts

Use build scripts to call `fiaccl` to generate MEX functions from your MATLAB function.

A build script automates a series of MATLAB commands that you want to perform repeatedly from the command line, saving you time and eliminating input errors. For instance, you can use a build script to clear your workspace before each build and to specify code generation options.

This example shows a build script to run `fiaccl` to process `lms_02.m`:

```
close all;
clear all;
clc;

N = 73113;

fiaccl -report lms_02.m ...
      -args { zeros(N,1) zeros(N,1) }
```

In this example, the following actions occur:

- `close all` deletes all figures whose handles are not hidden. See `close` in the MATLAB Graphics function reference for more information.

- `clear all` removes all variables, functions, and MEX-files from memory, leaving the workspace empty. This command also clears all breakpoints.

Note Remove the `clear all` command from the build scripts if you want to preserve breakpoints for debugging.

- `clc` clears all input and output from the Command Window display, giving you a “clean screen.”
- `N = 73113` sets the value of the variable `N`, which represents the number of samples in each of the two input parameters for the function `lms_02`
- `fiaccel -report lms_02.m -args { zeros(N,1) zeros(N,1) }` calls `fiaccel` to accelerate simulation of the file `lms_02.m` using the following options:
 - `-report` generates a code generation report
 - `-args { zeros(N,1) zeros(N,1) }` specifies the properties of the function inputs as a cell array of example values. In this case, the input parameters are `N`-by-1 vectors of real doubles.

Using the MATLAB Code Analyzer to Check Code Interactively at Design Time

The code analyzer checks your code for problems and recommends modifications to maximize performance and maintainability. You can use the code analyzer to check your code continuously in the MATLAB Editor while you work.

To ensure that continuous code checking is enabled:

- 1** From the MATLAB menu, select **File > Preferences > Code Analyzer**.

The list of code analyzer preferences appears.

- 2** Select the **Enable integrated warning and error messages** check box.

Separating Your Test Bench from Your Function Code

Separate your core algorithm from your test bench. Create a separate test script to do all the pre- and post-processing such as loading inputs, setting up input values, calling the function under test, and outputting test results. See the example on the `fiaccl` reference page.

Preserving Your Code

Preserve your code before making further modifications. This practice provides a fallback in case of error and a baseline for testing and validation. Use a consistent file naming convention, as described in “File Naming Conventions” on page 8-52. For example, add a 2-digit suffix to the file name for each file in a sequence. Alternatively, use a version control system.

File Naming Conventions

Use a consistent file naming convention to identify different types and versions of your MATLAB files. This approach keeps your files organized and minimizes the risk of overwriting existing files or creating two files with the same name in different folders.

For example, the file naming convention in the Generating MEX Functions getting started tutorial is:

- The suffix `_build` identifies a build script.
- The suffix `_test` identifies a test script.
- A numerical suffix, for example, `_01` identifies the version of a file. These numbers are typically two-digit sequential integers, beginning with 01, 02, 03, and so on.

For example:

- The file `build_01.m` is the first version of the build script for this tutorial.
- The file `test_03.m` is the third version of the test script for this tutorial.

Working with Fixed-Point Code Generation Reports

In this section...

“Generating the Code Generation Report” on page 8-53

“Opening the Code Generation Report” on page 8-54

“Viewing Your MATLAB Code” on page 8-54

“Viewing Variables in the Variables Tab” on page 8-56

“See Also” on page 8-57

Generating the Code Generation Report

When you compile your code with the `fiaccel` function or the MATLAB Coder `codegen` function, you can use the `-report` option to generate a code generation report. This report allows you to examine the data types of the variables and expressions in your code.

To see an example of the code generation report generated by the `fiaccel` function, compile `cordic_atan_kernel.m`. This file ships as a part of the Fixed-Point ATAN2 Calculation demo. You can open the file by typing the following at the MATLAB command line:

```
open cordic_atan_kernel
```

To compile the `cordic_atan_kernel` file, you must provide inputs `x`, `y`, `N`, and `angleLUT`. This example uses the following input values:

```
x = fi(0.23);  
y = x;  
N = 12;  
Tz = numerictype(1,16,13);  
angleLUT = fi(atan(2.^-(0:N-1)), 'NumericType', Tz);
```

After you define the input variables in the MATLAB workspace, change your working folder to a local folder and compile the file using `fiaccel`. Use the `-report` option to generate the code generation report:

```
fiaccel cordic_atan_kernel -args {x,y,N,angleLUT} -report
```

Opening the Code Generation Report

If the compilation is successful, you receive the following message:

Code generation successful: [View report](#)

Click the **View report** link to view the report.

If the compilation fails, a link to the error report appears:

Code generation failed: [View report](#)

Click the **View report** link to view the error report and debug your code. For more information on working with error reports, see “Code Generation Reports” in the MATLAB Coder documentation.

Viewing Your MATLAB Code

When the code generation report opens, you can hover your cursor over the variables and expressions in your MATLAB code to see their data type information. The code generation report provides color-coded data type information according to the following legend.

Color	Meaning
Green	Data type information is available for the selected variable at this location in the code.
Orange	There is a warning message associated with the selected variable or expression.
Pink	No data type information is available for the selected variable.
Purple	Data type information is available for the selected expression at this location in the code.
Red	There is an error message associated with the selected variable or expression.

Variables in your code that have data type information available appear highlighted in green, as shown in the following figure.

Function: **cordic_atan_kernel** Callers: Select a function call-site:

```

1 function [z,x,y] = cordic_atan_kernel(y,x,N,angleLUT) %#codegen
2 % Calculate arctangent in range [-pi/2, pi/2] using Vectoring mode CORDIC
3 % algorithm. Both x and y inputs must be real scalar, x must >= 0.
4 % Full precision Fimath is used in all fixed-point operations
5 %
6 % Inputs:
7 % y : y coordinate or imaginary part of the input vector
8 % x : x coordinate or real part of the input vector
9 % N : total number of iterations, must be a non-negative integer
10 % angleLUT : the angle look-up table, has same numerictype as the output
11 %     angle
12 % Output:
13 % z : angle that equals atan2(y,x), in radians
14 %     the output angle range is within [-pi/2, +pi/2]
15 % x : x coordinate of the last vector at the end of the iterations
16 % y : y coordinate of the last vector at the end of the iterations
17 %
18 % Copyright 1984-2010 The MathWorks, Inc.
19 % $Revision: 1.1.6.2 $ $Date: 2010/10/15 14:11:18 $
20
21 % initialization
22 z = angleLUT(1); z(:) = 0; % assume z_{0} is 0 and the same data type as angleLUT
23 for i = 0:N-1,
24     x0 = x;
25     if y < 0 % negative y leads to counter clock-wise rotation
26         x(:) = x0 - bitsra(y,i) * % x_{i+1} = x_{i} - y_{i}>>i
27         y(:) = x0 + bitsra(y,i) * % y_{i+1} = x_{i} + y_{i}>>i
28         z(:) = z0 + atan(2^{-i})
29     else
30         x(:) = x0 + bitsra(y,i) * % x_{i+1} = x_{i} + y_{i}>>i
31         y(:) = x0 - bitsra(y,i) * % y_{i+1} = x_{i} - y_{i}>>i
32         z(:) = z0 + atan(2^{-i})
33     end
34 end
35

```

Information for the selected variable:

Size	1 x 1
Complex	No
Class	embedded.fi
Signedness	Signed
WL	16
FL	17

Expressions in your code that have data type information available appear highlighted in purple, as the next figure shows.

```

21 % initialization
22 z = angleLUT(1); z(:) = 0; % assume z_{0} is 0 and the same data
23 for i = 0:N-1
24     x = x - y; % clock-wise rotation
25     i = i + 1; % counter-clockwise rotation
26     y = x_{i} - y_{i}>>i
27     x = y_{i} + x_{i}>>i
28     z = z_{i} + atan(2^{-i})
29     e = e - atan(2^{-i}) % rotation
30     y = x_{i} + y_{i}>>i
31     y(:) = y - bitsra(x0,i); % y_{i+1} = y_{i} - x_{i}>>i
32     z(:) = z + angleLUT(i+1); % z_{i+1} = z_{i} - atan(2^{-i})
33 end
34 end
35

```

Information for the selected expression:

Size	1 x 1
Complex	No
Class	embedded.fi
Signedness	Signed
WL	16
FL	13

Viewing Variables in the Variables Tab

To see the data type information for all the variables in your file, click the **Variables** tab of the code generation report. You can expand all **fi** and **fi** objects listed in the **Variables** tab to display the **fi** properties. When you expand a **fi** object in the **Variables** tab, the report indicates whether the **fi** object has a local **fi** object or is associated with the global **fi**.

The following figure shows the information displayed for a **fi** object that is associated with the global **fi**.

Generating MEX Functions from MATLAB Code That Uses Global Data

In this section...

“Workflow Overview” on page 8-58

“Declaring Global Variables” on page 8-58

“Defining Global Data” on page 8-59

“Synchronizing Global Data with MATLAB” on page 8-60

“Limitations of Using Global Data” on page 8-63

Workflow Overview

To generate MEX functions from MATLAB code that uses global data:

- 1 Declare the variables as global in your code.
- 2 Define and initialize the global data before using it.

For more information, see “Defining Global Data” on page 8-59.

- 3 Compile your code using `fiaccel`.

If you use global data, you must also specify whether you want to synchronize this data between MATLAB and the generated code. If there is no interaction between MATLAB and the generated code, it is safe to disable synchronization. Otherwise, you should enable synchronization. For more information, see “Synchronizing Global Data with MATLAB” on page 8-60.

Declaring Global Variables

For code generation, you must declare global variables before using them in your MATLAB code. Consider the `use_globals` function that uses two global variables AR and B.

```
function y = use_globals()  
%#codegen  
% Turn off inlining to make
```

```

% generated code easier to read
coder.inline('never');
% Declare AR and B as global variables
global AR;
global B;
AR(1) = B(1);
y = AR * 2;

```

Defining Global Data

You can define global data either in the MATLAB global workspace or at the command line. If you do not initialize global data at the command line, `fiaccl` looks for the variable in the MATLAB global workspace. If the variable does not exist, `fiaccl` generates an error.

Defining Global Data in the MATLAB Global Workspace

To compile the `use_globals` function described in “Declaring Global Variables” on page 8-58 using `fiaccl`:

- 1 Define the global data in the MATLAB workspace. At the MATLAB prompt, enter:

```

global AR B;
AR = fi(ones(4),1,16,14);
B = fi([1 2 3],1,16,13);

```

- 2 Compile the function to generate a MEX file named `use_globalsx`.

```

fiaccl -o use_globalsx use_globals

```

Defining Global Data at the Command Line

To define global data at the command line, use the `fiaccl -global` option. For example, to compile the `use_globals` function described in “Declaring Global Variables” on page 8-58, specify two global inputs `AR` and `B` at the command line.

```

fiaccl -o use_globalsx ...
    -global {'AR',fi(ones(4)),'B',fi([1 2 3])} use_globals

```

Alternatively, specify the type and initial value with the `-globals` flag using the format `-globals {'g', {type, initial_value}}`.

Defining Variable-Sized Global Data. To provide initial values for variable-sized global data, specify the type and initial value with the `-globals` flag using the format `-globals {'g', {type, initial_value}}`. For example, to specify a global variable `g1` that has an initial value `[1 1]` and upper bound `[2 2]`, enter:

```
fiaccel foo -globals {'g1',{coder.typeof(0,[2 2],true),[1 1]}}
```

For a detailed explanation of `coder.typeof` syntax, see `coder.typeof`.

Synchronizing Global Data with MATLAB

Why Synchronize Global Data?

The generated code and MATLAB each have their own copies of global data. To ensure consistency, you must synchronize their global data whenever the two interact. If you do not synchronize the data, their global variables might differ. The level of interaction determines when to synchronize global data.

When to Synchronize Global Data

By default, synchronization between global data in MATLAB and generated code occurs at MEX function entry and exit and for all *extrinsic* calls, which are calls to MATLAB functions on the MATLAB path that `fiaccel` dispatches to MATLAB for execution. This behavior ensures maximum consistency between generated code and MATLAB.

To improve performance, you can:

- Select to synchronize only at MEX function entry and exit points.
- Disable synchronization when the global data does not interact.
- Choose whether to synchronize before and after each extrinsic call.

The following table summarizes which global data synchronization options to use. To learn how to set these options, see “How to Synchronize Global Data” on page 8-61.

Global Data Synchronization Options

If you want to...	Set the global data synchronization mode to:	Synchronize before and after extrinsic calls?
Ensure maximum consistency when all extrinsic calls modify global data.	At MEX-function entry, exit and extrinsic calls (default)	Yes. Default behavior.
Ensure maximum consistency when most extrinsic calls modify global data, but a few do not.	At MEX-function entry, exit and extrinsic calls (default)	Yes. Use the <code>coder.extrinsic-sync:off</code> option to turn off synchronization for the extrinsic calls that do not affect global data.
Ensure maximum consistency when most extrinsic calls do not modify global data, but a few do.	At MEX-function entry and exit	Yes. Use the <code>coder.extrinsic-sync:on</code> option to synchronize only the calls that modify global data
Maximize performance when synchronizing global data, and none of your extrinsic calls modify global data.	At MEX-function entry and exit	No.
Communicate between generated code files only. No interaction between global data in MATLAB and generated code.	Disabled	No.

How to Synchronize Global Data

To control global data synchronization, set the global data synchronization mode and select whether to synchronize extrinsic functions. For guidelines on which options to use, see “When to Synchronize Global Data” on page 8-60.

You control the synchronization of global data with extrinsic functions using the `coder.extrinsic -sync:on` and `-sync:off` options.

Controlling the Global Data Synchronization Mode from the Command Line.

- 1 Define the compiler options object in the MATLAB workspace by issuing a constructor command:

```
comp_cfg = coder.mexconfig
```

- 2 From the command line, set the `GlobalDataSyncMethod` property to `Always`, `SyncAtEntryAndExits` or `NoSync`, as applicable. For example:

```
comp_cfg.GlobalDataSyncMethod = 'SyncAtEntryAndExits';
```

- 3 Use the `comp_cfg` configuration object when compiling your code by specifying it using the `-config` compilation option. For example,

```
fiaccel -config comp_cfg myFile
```

Controlling Synchronization for Extrinsic Function Calls. You can control whether synchronization between global data in MATLAB and generated code occurs before and after you call an extrinsic function. To do so, use the `coder.extrinsic -sync:on` and `-sync:off` options.

By default, global data is:

- Synchronized before and after each extrinsic call if the global data synchronization mode is `At MEX-function entry, exit and extrinsic calls`. If you are sure that certain extrinsic calls do not affect global data, turn off synchronization for these calls using the `-sync:off` option. Turning off synchronization improves performance. For example, if functions `foo1` and `foo2` *do not* affect global data, turn off synchronization for these functions:

```
coder.extrinsic('-sync:off', 'foo1', 'foo2');
```

- Not synchronized if the global data synchronization mode is `At MEX-function entry and exit`. If the code has a few extrinsic calls that affect global data, turn on synchronization for these calls using the

-sync:on option. For example, if functions `foo1` and `foo2` *do* affect global data, turn on synchronization for these functions:

```
coder.extrinsic('-sync:on', 'foo1', 'foo2');
```

- Not synchronized if the global data synchronization mode is `Disabled`. When synchronization is disabled, you cannot control the synchronization for specific extrinsic calls. The `-sync:on` option has no effect.

Limitations of Using Global Data

You cannot use global data with

- The `coder.cstructname` function. This function does not support global variables.
- The `coder.varsize` function. Instead, use a `coder.typeof` object to define variable-sized global data as described in “Defining Variable-Sized Global Data” on page 8-60.

Defining Input Properties Programmatically in the MATLAB File

In this section...

“How to Use `assert` with `fiaccel`” on page 8-64

“Rules for Using `assert` Function” on page 8-69

“Example: Specifying Properties of Primary Fixed-Point Inputs” on page 8-69

“Example: Specifying Class and Size of Scalar Structure” on page 8-70

“Example: Specifying Class and Size of Structure Array” on page 8-71

How to Use `assert` with `fiaccel`

You can use the MATLAB `assert` function to define properties of primary function inputs directly in your MATLAB file.

Use the `assert` function to invoke standard MATLAB functions for specifying the class, size, and complexity of primary function inputs.

Specify Any Class

```
assert ( isa ( param, 'class_name' ) )
```

Sets the input parameter *param* to the MATLAB class *class_name*. For example, to set the class of input `U` to a 32-bit signed integer, call:

```
...  
assert(isa(U, 'embedded.fi'));  
...
```

Note If you set the class of an input parameter to `fi`, you must also set its `numericType`, see “Specify `numericType` of Fixed-Point Input” on page 8-67. You can also set its `fimath` properties, see “Specify `fimath` of Fixed-Point Input” on page 8-68. If you do not set the `fimath` properties, `fiaccel` uses the MATLAB default `fimath` value.

If you set the class of an input parameter to `struct`, you must specify the properties of each field in the structure in the order in which you define the fields in the structure definition.

Specify `fi` Class

```
assert ( isfi ( param ) )
assert ( isa ( param, 'embedded.fi' ) )
```

Sets the input parameter `param` to the MATLAB class `fi` (fixed-point numeric object). For example, to set the class of input `U` to `fi`, call:

```
...
assert(isfi(U));
...
```

or

```
...
assert(isa(U, 'embedded.fi'));
...
```

Note If you set the class of an input parameter to `fi`, you must also set its `numericType`, see “Specify `numericType` of Fixed-Point Input” on page 8-67. You can also set its `fimath` properties, see “Specify `fimath` of Fixed-Point Input” on page 8-68. If you do not set the `fimath` properties, `fiaccel` uses the MATLAB default `fimath` value.

Specify Structure Class

```
assert ( isstruct ( param ) )
```

Sets the input parameter *param* to the MATLAB class `struct` (structure). For example, to set the class of input `U` to a `struct`, call:

```
...  
assert(isstruct(U));  
...
```

or

```
...  
assert(isa(U, 'struct'));  
...
```

Note If you set the class of an input parameter to `struct`, you must specify the properties of each field in the structure in the order in which you define the fields in the structure definition.

Specify Any Size

```
assert ( all ( size ( param == [ dims ] ) ) )
```

Sets the input parameter *param* to the size specified by dimensions *dims*. For example, to set the size of input `U` to a 3-by-2 matrix, call:

```
...  
assert(all(size(U)== [3 2]));  
...
```

Specify Scalar Size

```
assert ( isscalar ( param ) )  
assert ( all ( size ( param == [ 1 ] ) ) )
```

Sets the size of input parameter *param* to scalar. For example, to set the size of input `U` to scalar, call:

```
...
assert(isscalar(U));
...
```

or

```
...
assert(all(size(U)== [1]));
...
```

Specify Real Input

```
assert ( isreal ( param ) )
```

Specifies that the input parameter *param* is real. For example, to specify that input *U* is real, call:

```
...
assert(isreal(U));
...
```

Specify Complex Input

```
assert ( ~isreal ( param ) )
```

Specifies that the input parameter *param* is complex. For example, to specify that input *U* is complex, call:

```
...
assert(~isreal(U));
...
```

Specify numerictype of Fixed-Point Input

```
assert ( isequal ( numerictype ( fiparam ), T ) )
```

Sets the numerictype properties of fi input parameter *fiparam* to the numerictype object *T*. For example, to specify the numerictype property of

fixed-point input *U* as a signed `numericType` object *T* with 32-bit word length and 30-bit fraction length, use the following code:

```
...
% Define the numericType object.
T = numericType(1, 32, 30);

% Set the numericType property of input U to T.
assert(isequal(numericType(U),T));
...
```

Specify `fimath` of Fixed-Point Input

```
assert ( isequal ( fimath ( fiparam ), F ) )
```

Sets the `fimath` properties of `fi` input parameter *fiparam* to the `fimath` object *F*. For example, to specify the `fimath` property of fixed-point input *U* so that it saturates on integer overflow, use the following code:

```
...
% Define the fimath object.
F = fimath('OverflowMode','saturate');

% Set the fimath property of input U to F.
assert(isequal(fimath(U),F));
...
```

Note If you do not specify the `fimath` properties using `assert`, `fiaccel` uses the MATLAB default `fimath` value.

Specify Multiple Properties of Input

```
assert ( function1 ( params ) && function2 ( params ) && function3 ( params ) && ... )
```

Specifies the class, size, and complexity of one or more inputs using a single `assert` function call. For example, the following code specifies that input *U* is a double, complex, 3-by-3 matrix, and input *V* is a 16-bit unsigned integer:

```

...
assert(isa(U,'double') && ~isreal(U) && all(size(U) == [3 3]) && isa(V,'uint16'));
...

```

Rules for Using assert Function

Follow these rules when using the `assert` function to specify the properties of primary function inputs:

- Call `assert` functions at the beginning of the primary function, before any flow-control operations such as `if` statements or subroutine calls.
- Do not call `assert` functions inside conditional constructs, such as `if`, `for`, `while`, and `switch` statements.
- Use the `assert` function with `fiaccel` only for specifying properties of primary function inputs before converting your MATLAB code to MEX code.
- If you set the class of an input parameter to `fi`:
 - You must also set its `numerictype`, see “Specify `numerictype` of Fixed-Point Input” on page 8-67.
 - You can also set its `fimath` properties, see “Specify `fimath` of Fixed-Point Input” on page 8-68. If you do not set the `fimath` properties, `fiaccel` uses the MATLAB default `fimath` value.
- If you set the class of an input parameter to `struct`, you must specify the class, size, and complexity of each field in the structure in the order in which you define the fields in the structure definition.

Example: Specifying Properties of Primary Fixed-Point Inputs

In the following example, the primary MATLAB function `emcsqrtfi` takes one fixed-point input: `x`. The code specifies the following properties for this input:

Property	Value
<code>class</code>	<code>fi</code>
<code>numerictype</code>	<code>numerictype</code> object <code>T</code> , as specified in the primary function

Property	Value
fimath	fimath object F, as specified in the primary function
size	scalar (by default)
complexity	real (by default)

```
function y = emcsqrtfi(x)
T = numericity('WordLength',32,'FractionLength',23,...
    'Signed',true);
F = fimath('SumMode','SpecifyPrecision',...
    'SumWordLength',32,'SumFractionLength',23,...
    'ProductMode','SpecifyPrecision',...
    'ProductWordLength',32,'ProductFractionLength',23);
assert(isfi(x));
assert(isequal(numericity(x),T));
assert(isequal(fimath(x),F));

y = sqrt(x);
```

Example: Specifying Class and Size of Scalar Structure

Assume you have defined S as the following scalar MATLAB structure:

```
S = struct('r',double(1),'i',fi(4,true,8,0));
```

This code specifies the class and size of S and its fields when passed as an input to your MATLAB function:

```
function y = fcn(S)

% Specify the class of the input as struct.
assert(isstruct(S));

% Specify the size of the fields r and i
% in the order in which you defined them.
T = numericity('Wordlength', 8,'FractionLength', ...
    0,'signed',true);
assert(isa(S.r,'double'));
```

```

assert(isfi(S.i) && isequal(numericType(S.i),T));

y = S;

```

Note In most cases, `fiaccel` uses defaults when you do not explicitly specify values for properties—except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore in the preceding example, an `assert` function specifies that field `S.r` is of type `double`, even though `double` is the default.

Example: Specifying Class and Size of Structure Array

For structure arrays, you must choose a representative element of the array for specifying the properties of each field. For example, assume you have defined `S` as the following 1-by-2 array of MATLAB structures:

```

S = struct('r',{double(1), double(2)},'i',...
          {fi(4,1,8,0), fi(5,1,8,0)});

```

The following code specifies the class and size of each field of structure input `S` using the first element of the array:

```

function y = fcn(S)

% Specify the class of the input S as struct.
assert(isstruct(S));
T = numericType('Wordlength', 8,'FractionLength', ...
               0,'signed',true);

% Specify the size of the fields r and i
% based on the first element of the array.
assert(all(size(S) == [1 2]));
assert(isa(S(1).r,'double'));
assert(isfi(S(1).i) && isequal(numericType(S(1).i),T));

y = S;

```

Note In most cases, `fiaccel` uses defaults when you don't explicitly specify values for properties — except for structure fields. The only way to name a field in a structure is to set at least one of its properties. Therefore in the example above, an `assert` function specifies that field `S(1).r` is of type `double`, even though `double` is the default.

Controlling Run-Time Checks

In this section...
“Types of Run-Time Checks” on page 8-73
“When to Disable Run-Time Checks” on page 8-74
“How to Disable Run-Time Checks” on page 8-74

Types of Run-Time Checks

In simulation, the code generated for your MATLAB functions includes the following run-time checks and external function calls.

- Memory integrity checks

These checks detect violations of memory integrity in code generated for MATLAB functions and stop execution with a diagnostic message.

Caution For safety, these checks are enabled by default. Without memory integrity checks, violations will result in unpredictable behavior.

- Responsiveness checks in code generated for MATLAB functions

These checks enable periodic checks for Ctrl+C breaks in code generated for MATLAB functions. Enabling responsiveness checks also enables graphics refreshing.

Caution For safety, these checks are enabled by default. Without these checks the only way to end a long-running execution might be to terminate MATLAB.

- Extrinsic calls to MATLAB functions

Extrinsic calls to MATLAB functions, for example to display results, are enabled by default for debugging purposes. For more information

about extrinsic functions, see “Declaring MATLAB Functions as Extrinsic Functions”.

When to Disable Run-Time Checks

Generally, generating code with run-time checks enabled results in more generated code and slower simulation than generating code with the checks disabled. Similarly, extrinsic calls are time consuming and have an adverse effect on performance. Disabling run-time checks and extrinsic calls usually results in streamlined generated code and faster simulation, with these caveats:

Consider disabling...	Only if...
Memory integrity checks	You are sure that your code is safe and that all array bounds and dimension checking is unnecessary.
Responsiveness checks	You are sure that you will not need to stop execution of your application using Ctrl+C .
Extrinsic calls	You are only using extrinsic calls to functions that do not affect application results.

How to Disable Run-Time Checks

To disable run-time checks:

- 1 Define the compiler options object in the MATLAB workspace by issuing a constructor command:

```
comp_cfg = coder.MEXConfig
```

- 2 From the command line set the `IntegrityChecks`, `ExtrinsicCalls`, or `ResponsivenessChecks` properties false, as applicable:

```
comp_cfg.IntegrityChecks = false;
comp_cfg.ExtrinsicCalls = false;
```

```
comp_cfg.ResponsivenessChecks = false;
```

MATLAB Coder

MATLAB Coder codegen automatically converts MATLAB code directly to C code. It generates standalone C code that is bit-true to fixed-point MATLAB code. Using Fixed-Point Toolbox software you can generate C code with algorithms containing integer math only (i.e., without any floating-point math). For more information, refer to the *MATLAB Coder User's Guide*.

MATLAB Function Block

In this section...

“Composing a MATLAB Language Function in a Simulink Model” on page 8-77

“Using the MATLAB Function Block with Data Type Override” on page 8-77

“Using Fixed-Point Data Types with the MATLAB Function Block” on page 8-79

“Example: Implementing a Fixed-Point Direct Form FIR Using the MATLAB Function Block” on page 8-86

Composing a MATLAB Language Function in a Simulink Model

The MATLAB Function block lets you compose a MATLAB language function in a Simulink model that generates embeddable code. When you simulate the model or generate code for a target environment, a function in a MATLAB Function block generates efficient C/C++ code. This code meets the strict memory and data type requirements of embedded target environments. In this way, the MATLAB Function blocks bring the power of MATLAB for the embedded environment into Simulink.

For more information about the MATLAB Function block and code generation, refer to the following:

- MATLAB Function block reference page in the Simulink documentation
- “Using the MATLAB Function Block” in the Simulink documentation
- “About Code Generation from MATLAB Algorithms” in the Code Generation from MATLAB documentation

Using the MATLAB Function Block with Data Type Override

When you use the MATLAB Function block in a Simulink model that specifies data type override, the block determines the data type override equivalents of the input signal and parameter types. The block then uses these equivalent

values to run the simulation. The following table shows how the MATLAB Function block determines the data type override equivalent using

- The data type of the input signal or parameter
- The data type override setting in the Simulink model

Note The MATLAB Function block does not support the Scaled double data type override setting.

Input Signal or Parameter Type	Data Type Override Setting	Data Type Override Equivalent
Inherited single	Double	fi double
	Single	fi single
Specified single	Double	Built-in double
	Single	Built-in single
Inherited double	Double	fi double
	Single	fi single
Specified double	Double	Built-in double
	Single	Built-in single
Inherited Fixed	Double	fi double
	Single	fi single
Specified Fixed	Double	fi double
	Single	fi single

For more information about using the MATLAB Function block with data type override, see the following section of the Simulink documentation:

“Using Data Type Override with the MATLAB Function Block”

Using Fixed-Point Data Types with the MATLAB Function Block

Code generation from MATLAB supports a significant number of Fixed-Point Toolbox functions. Refer to “Functions Supported for Code Acceleration and Code Generation from MATLAB” on page 8-5 for information about which Fixed-Point Toolbox functions are supported.

For more information on working with fixed-point MATLAB Function blocks, see:

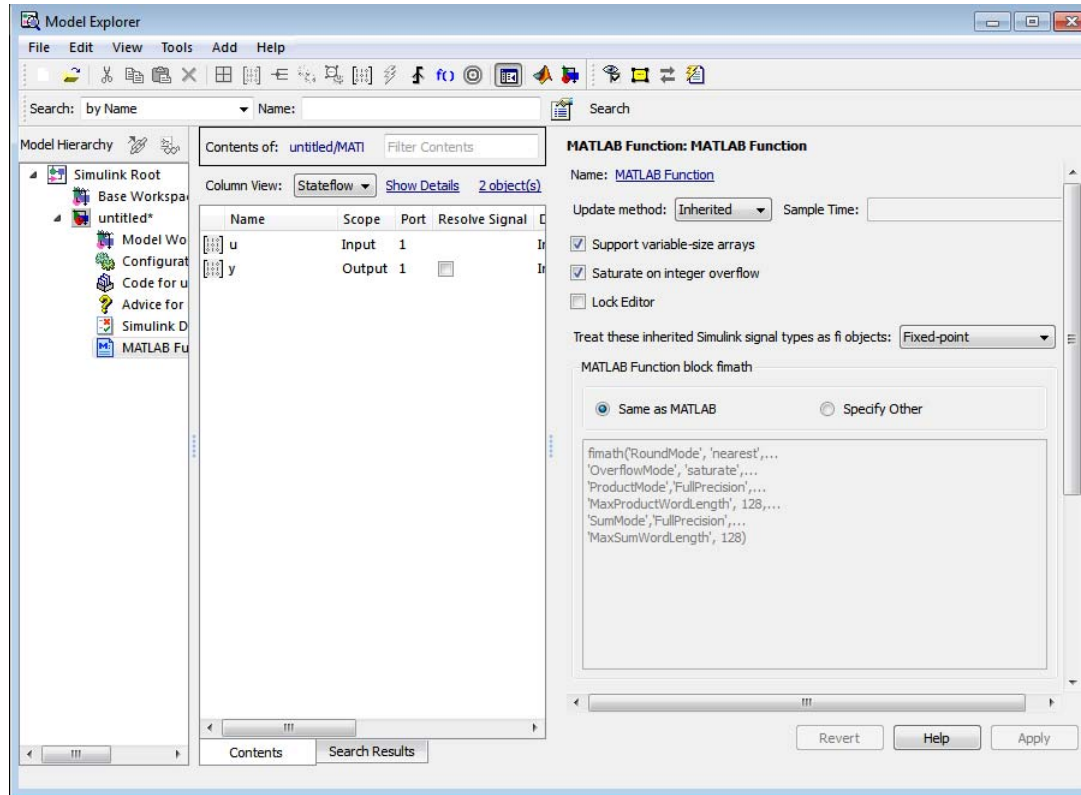
- “Specifying Fixed-Point Parameters in the Model Explorer” on page 8-79
- “Using fimath Objects in MATLAB Function Blocks” on page 8-81
- “Sharing Models with Fixed-Point MATLAB Function Blocks” on page 8-84

Note To simulate models using fixed-point data types in Simulink, you must have a Simulink Fixed Point license.

Specifying Fixed-Point Parameters in the Model Explorer

You can specify parameters for an MATLAB Function block in a fixed-point model using the Model Explorer. Try the following exercise:

- 1** Place a MATLAB Function block in a new model. You can find the block in the Simulink User-Defined Functions library.
- 2** Open the Model Explorer by selecting **View > Model Explorer** from your model.
- 3** Expand the **untitled*** node in the **Model Hierarchy** pane of the Model Explorer. Then, select the **MATLAB Function** node. The Model Explorer now appears as shown in the following figure.



The following parameters in the **Dialog** pane apply to MATLAB Function blocks in models that use fixed-point and integer data types:

Treat these inherited Simulink signal types as fi objects

Choose whether to treat inherited fixed-point and integer signals as *fi* objects.

- When you select **Fixed-point**, the MATLAB Function block treats all fixed-point inputs as Fixed-Point Toolbox *fi* objects.
- When you select **Fixed-Point & Integer**, the MATLAB Function block treats all fixed-point and integer inputs as Fixed-Point Toolbox *fi* objects.

MATLAB Function block `fimath`

Specify the `fimath` properties for the block to associate with the following objects:

- All fixed-point and integer input signals to the MATLAB Function block that you choose to treat as `fi` objects.
- All `fi` and `fimath` objects constructed in the MATLAB Function block.

You can select one of the following options for the **MATLAB Function block `fimath`**:

- **Same as MATLAB** — When you select this option, the block uses the same `fimath` properties as the current global `fimath`. The edit box appears dimmed and displays the current global `fimath` in read-only form.
- **Specify other** — When you select this option, you can specify your own `fimath` object in the edit box.

For more information on these parameters, see “Using `fimath` Objects in MATLAB Function Blocks” on page 8-81.

Using `fimath` Objects in MATLAB Function Blocks

The **MATLAB Function block `fimath`** parameter enables you to specify one set of `fimath` object properties for the MATLAB Function block. The block associates the `fimath` properties you specify with the following objects:

- All fixed-point and integer input signals to the MATLAB Function block that you choose to treat as `fi` objects.
- All `fi` and `fimath` objects constructed in the MATLAB Function block.

You can set these parameters on the following dialog box, which you can access through either the Model Explorer or the “Ports and Data Manager”.

MATLAB Function: MATLAB Function

Name: [MATLAB Function](#)

Update method: Sample Time:

Support variable-size arrays

Saturate on integer overflow

Lock Editor

Treat these inherited Simulink signal types as fi objects:

MATLAB Function block fimath

Same as MATLAB Specify Other

```
fimath('RoundMode', 'nearest', ...
'OverflowMode', 'saturate', ...
'ProductMode', 'FullPrecision', ...
'MaxProductWordLength', 128, ...
'SumMode', 'FullPrecision', ...
'MaxSumWordLength', 128)
```

Description:

[Document link:](#)

- To access this pane through the Model Explorer:
 - Select **View > Model Explorer** from your model menu.
 - Then, select the MATLAB Function block from the Model Hierarchy pane on the left side of the Model Explorer.
- To access this pane through the Ports and Data Manager, select **Tools > Edit Data/Ports** from the MATLAB Editor menu.

When you select **Same as MATLAB** for the **MATLAB Function block `fimath`**, the MATLAB Function block uses the current global `fimath`. The current global `fimath` appears dimmed and in read-only form in the edit box.

When you select **Specify other** the block allows you to specify your own `fimath` object in the edit box. You can do so in one of two ways:

- Constructing the `fimath` object inside the edit box.
- Constructing the `fimath` object in the MATLAB or model workspace and then entering its variable name in the edit box.

Note If you use this option and plan to share your model with others, make sure you define the variable in the model workspace. See “Sharing Models with Fixed-Point MATLAB Function Blocks” on page 8-84 for more information on sharing models.

The Fixed-Point Toolbox `isfimathlocal` function supports code generation for MATLAB. You cannot generate code for the other Fixed-Point Toolbox functions related to the global `fimath` because they change the MATLAB preferences. These unsupported functions include:

- `globalfimath`
- `saveglobalfimathpref`
- `removeglobalfimathpref`
- `resetglobalfimath`

Sharing Models with Fixed-Point MATLAB Function Blocks

When you collaborate with a coworker, you can share a fixed-point model using the MATLAB Function block. To share a model, make sure that you move any variables you define in the MATLAB workspace, including `fimath` objects, to the model workspace. For example, try the following:

- 1 Place a MATLAB Function block in a new model. You can find the block in the Simulink User-Defined Functions library.
- 2 Define a `fimath` object in the MATLAB workspace that you want to use for any Simulink fixed-point signal entering the MATLAB Function block as an input:

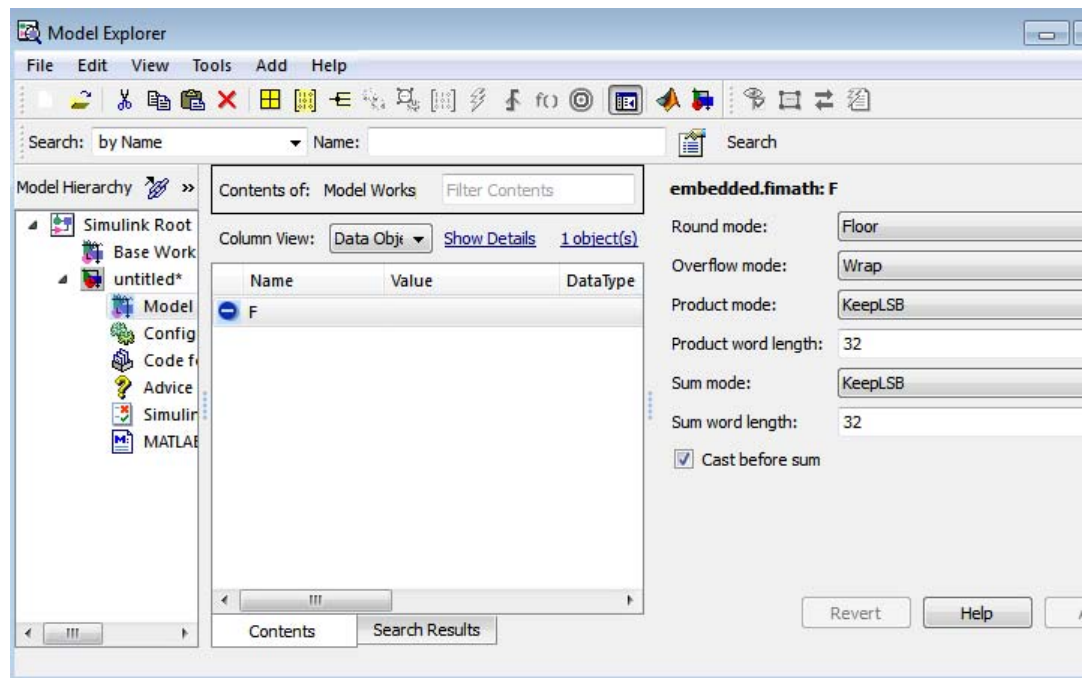
```
F = fimath('RoundMode','Floor','OverflowMode','Wrap',...
          'ProductMode','KeepLSB','ProductWordLength',32,...
          'SumMode','KeepLSB','SumWordLength',32)
```

```
F =
      RoundMode: floor
      OverflowMode: wrap
      ProductMode: KeepLSB
ProductWordLength: 32
      SumMode: KeepLSB
SumWordLength: 32
CastBeforeSum: true
```

- 3 Open the Model Explorer by selecting **View > Model Explorer** from your model.
- 4 Expand the **untitled*** node in the **Model Hierarchy** pane of the Model Explorer, and select the **MATLAB Function** node.
- 5 Select **Specify other** for the **MATLAB Function block fimath** parameter and enter the variable `F` into the edit box on the **Dialog** pane. Click **Apply** to save your changes.

You have now defined the `fimath` properties to be associated with all Simulink fixed-point input signals and all `fi` and `fimath` objects constructed within the block.

- 6 Select the **Base Workspace** node in the **Model Hierarchy** pane. You can see the variable **F** that you have defined in the MATLAB workspace listed in the **Contents** pane. If you send this model to a coworker, that coworker must first define that same variable in the MATLAB workspace to get the same results.
- 7 Cut the variable **F** from the base workspace, and paste it into the model workspace listed under the node for your model, in this case, **untitled***. The Model Explorer now appears as shown in the following figure.



You can now e-mail your model to a coworker. Because you included the required variables in the workspace of the model itself, your coworker can simply run the model and get the correct results. Receiving and running the model does not require any extra steps.

Example: Implementing a Fixed-Point Direct Form FIR Using the MATLAB Function Block

The following sections lead you through creating a fixed-point, low-pass, direct form FIR filter in Simulink. To create the FIR filter, you use Fixed-Point Toolbox software and the MATLAB Function block. In this example, you perform the following tasks in the sequence shown:

- “Program the MATLAB Function Block” on page 8-86
- “Prepare the Inputs” on page 8-88
- “Create the Model” on page 8-89
- “Define the fimath Object Using the Model Explorer” on page 8-93
- “Run the Simulation” on page 8-94

Program the MATLAB Function Block

- 1 Place a MATLAB Function block in a new model. You can find the block in the Simulink User-Defined Functions library.
- 2 Save your model as `cgen_fi.mdl`.
- 3 Double-click the MATLAB Function block in your model to open the MATLAB Function Block Editor. Type or copy and paste the following MATLAB code, including comments, into the Editor:

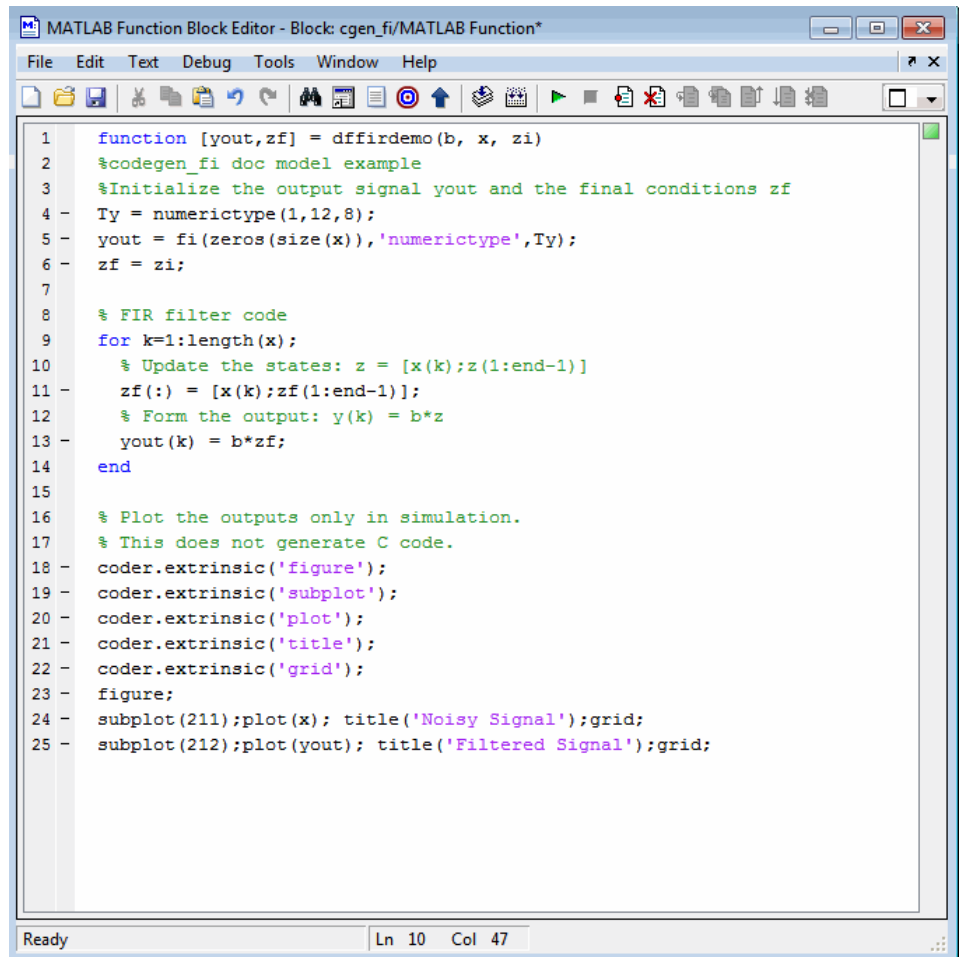
```
function [yout,zf] = dffirdemo(b, x, zi)
%codegen_fi doc model example
%Initialize the output signal yout and the final conditions zf
Ty = numericity(1,12,8);
yout = fi(zeros(size(x)), 'numericity', Ty);
zf = zi;

% FIR filter code
for k=1:length(x);
    % Update the states: z = [x(k);z(1:end-1)]
    zf(:) = [x(k);zf(1:end-1)];
    % Form the output: y(k) = b*z
    yout(k) = b*zf;
```

```
end

% Plot the outputs only in simulation.
% This does not generate C code.
coder.extrinsic('figure');
coder.extrinsic('subplot');
coder.extrinsic('plot');
coder.extrinsic('title');
coder.extrinsic('grid');
figure;
subplot(211);plot(x); title('Noisy Signal');grid;
subplot(212);plot(yout); title('Filtered Signal');grid;
```

The Editor now appears as shown in the following figure.



```
1 function [yout,zf] = dffirdemo(b, x, zi)
2 %codegen_fi doc model example
3 %Initialize the output signal yout and the final conditions zf
4 - Ty = numerictype(1,12,8);
5 - yout = fi(zeros(size(x)), 'numerictype', Ty);
6 - zf = zi;
7
8 % FIR filter code
9 for k=1:length(x);
10     % Update the states: z = [x(k);z(1:end-1)]
11     zf(:) = [x(k);zf(1:end-1)];
12     % Form the output: y(k) = b*z
13     yout(k) = b*zf;
14 end
15
16 % Plot the outputs only in simulation.
17 % This does not generate C code.
18 - coder.extrinsic('figure');
19 - coder.extrinsic('subplot');
20 - coder.extrinsic('plot');
21 - coder.extrinsic('title');
22 - coder.extrinsic('grid');
23 - figure;
24 - subplot(211);plot(x); title('Noisy Signal');grid;
25 - subplot(212);plot(yout); title('Filtered Signal');grid;
```

Prepare the Inputs

Define the filter coefficients b , noise x , and initial conditions zi by typing the following code at the MATLAB command line:

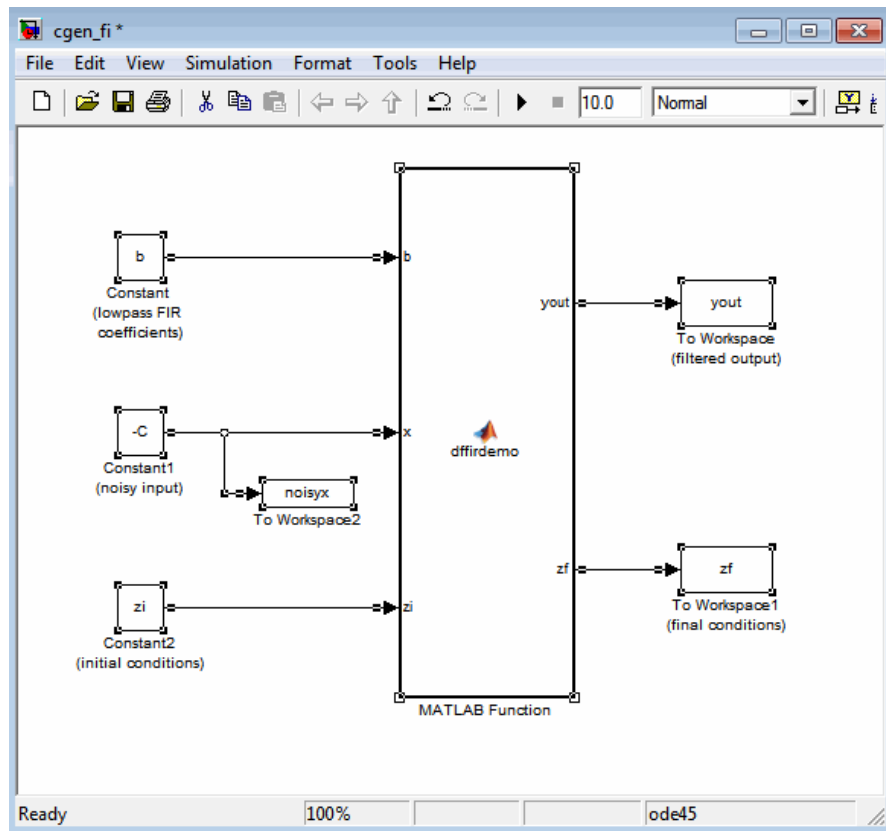
```
b = fi_fir_coefficients;
load mtlb
x = mtlb;
n = length(x);
```



```
noise = sin(2*pi*2140*(0:n-1)'./Fs);  
x = x + noise;  
zi = zeros(length(b),1);
```

Create the Model

- 1 Add blocks to your model to create the following system.



2 Set the block parameters in the model to the following values.

Block	Parameter	Value
Constant	Constant value	b
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	Sample based
	Sample time	inf
	Mode	Fixed point
	Signedness	Signed
	Scaling	Slope and bias
	Word length	12
	Slope	2^{-12}
	Bias	0
Constant1	Constant value	x+noise
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	Sample based
	Sample time	1
	Mode	Fixed point
	Signedness	Signed
	Scaling	Slope and bias
	Word length	12
	Slope	2^{-8}
	Bias	0

Block	Parameter	Value
Constant2	Constant value	zi
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	Sample based
	Sample time	inf
	Mode	Fixed point
	Signedness	Signed
	Scaling	Slope and bias
	Word length	12
	Slope	2^{-8}
	Bias	0
To Workspace	Variable name	yout
	Limit data points to last	inf
	Decimation	1
	Sample time	-1
	Save format	Array
	Log fixed-point data as a fi object	Selected

Block	Parameter	Value
To Workspace1	Variable name	zf
	Limit data points to last	inf
	Decimation	1
	Sample time	-1
	Save format	Array
	Log fixed-point data as a fi object	Selected
To Workspace2	Variable name	noisyx
	Limit data points to last	inf
	Decimation	1
	Sample time	-1
	Save format	Array
	Log fixed-point data as a fi object	Selected

- 3 From the model menu, select **Simulation > Configuration Parameters** and set the following parameters.

Parameter	Value
Stop time	0
Type	Fixed-step
Solver	discrete (no continuous states)

Click **Apply** to save your changes.

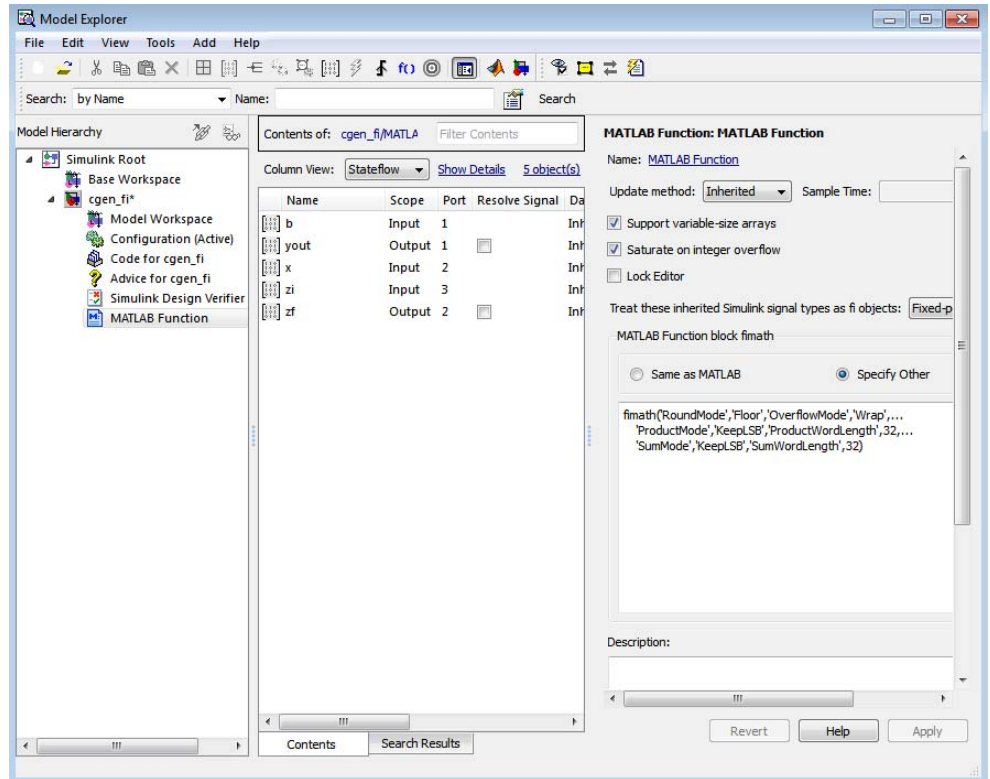
Define the fimath Object Using the Model Explorer

- 1 Open the Model Explorer for the model by selecting **View > Model Explorer**.
- 2 Click the **cgen_fi > MATLAB Function** node in the **Model Hierarchy** pane. The dialog box for the MATLAB Function block appears in the **Dialog** pane of the Model Explorer.
- 3 Select **Specify other** for the **MATLAB Function block fimath** parameter on the MATLAB Function block dialog box. You can then create the following fimath object in the edit box:

```
fimath('RoundMode','Floor','OverflowMode','Wrap',...  
      'ProductMode','KeepLSB','ProductWordLength',32,...  
      'SumMode','KeepLSB','SumWordLength',32)
```

The `fimath` object you define here is associated with fixed-point inputs to the MATLAB Function block as well as the `fi` object you construct within the block.

Click **Apply** to save your changes. The model explorer now appears as shown in the following figure.



By selecting **Specify other** for the **MATLAB Function block fimath**, you ensure that your model always uses the `fimath` properties you specified. Thus, the model does not depend on the global `fimath` and always produces the same results.

Run the Simulation

- 1 Run the simulation by selecting your model and typing **Ctrl+T**. While the simulation is running, information outputs to the MATLAB command line. You can look at the plots of the noisy signal and the filtered signal.
- 2 Next, build embeddable C code for your model by selecting the model and typing **Ctrl+B**. While the code is building, information outputs to the

MATLAB command line. A folder called `coder_fi_grt_rtw` is created in your current working folder.

- 3 Navigate to `coder_fi_grt_rtw > coder_fi.c`. In this file, you can see the code generated from your model. Search for the following comment in your code:

```
/* coder_fi doc model example */
```

This search brings you to the beginning of the section of the code that your MATLAB Function block generated.

Interoperability with Other Products

- “Using fi Objects with Simulink” on page 9-2
- “Using fi Objects with DSP System Toolbox ” on page 9-7
- “Using fiaccel, MATLAB® Coder, or Simulink to Generate Code” on page 9-12

Using fi Objects with Simulink

In this section...
“Reading Fixed-Point Data from the Workspace” on page 9-2
“Writing Fixed-Point Data to the Workspace” on page 9-2
“Setting the Value and Data Type of Block Parameters” on page 9-6
“Logging Fixed-Point Signals” on page 9-6
“Accessing Fixed-Point Block Data During Simulation” on page 9-6

Reading Fixed-Point Data from the Workspace

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

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

Writing Fixed-Point Data to the Workspace

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

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

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

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

```
a =
```

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

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 15
```

```
s.signals.values = a
```

```
s =
```

```
      signals: [1x1 struct]
```

```
s.signals.dimensions = 2
```

```
s =
```

```
      signals: [1x1 struct]
```

```
s.time = [0:10]'
```

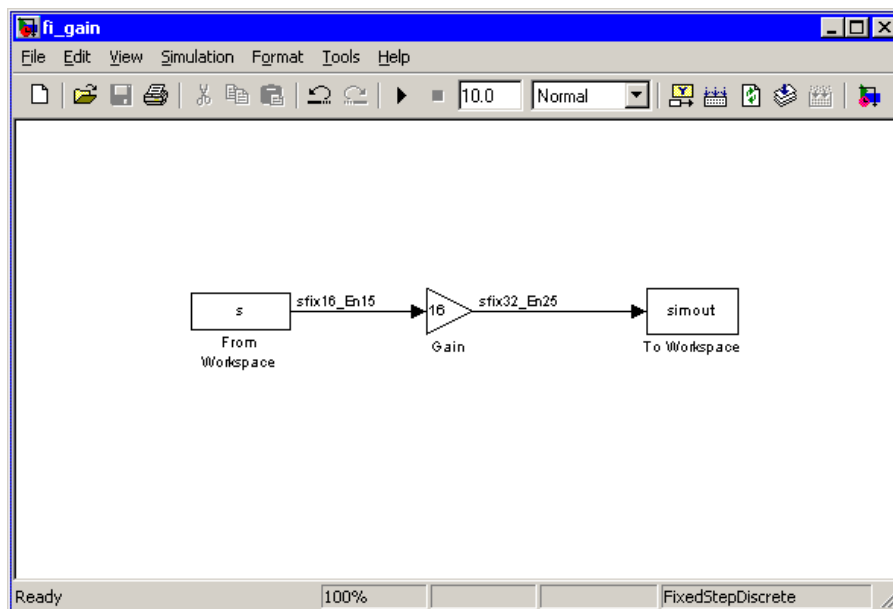
```
s =  
  
    signals: [1x1 struct]  
    time: [11x1 double]
```

The From Workspace block in the following model has the `fi` structure `s` in the **Data** parameter.

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

In the model, the following parameters in the **Solver** pane of the **Configuration Parameters** dialog have the indicated settings:

- **Start time** — 0.0
- **Stop time** — 10.0
- **Type** — Fixed-step
- **Solver** — Discrete (no continuous states)
- **Fixed step size (fundamental sample time)** — 1.0



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

```
simout.signals.values
```

```
ans =
```

```

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

```

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 32
FractionLength: 25
```

Setting the Value and Data Type of Block Parameters

You can use Fixed-Point Toolbox expressions to specify the value and data type of block parameters in Simulink. Refer to “Block Support for Data and Numeric Signal Types” in the Simulink documentation for more information.

Logging Fixed-Point Signals

When fixed-point signals are logged to the MATLAB workspace via signal logging, they are always logged as `fi` objects. To enable signal logging for a signal, select the **Log signal data** option in the signal’s **Signal Properties** dialog box. For more information, refer to “Exporting Signal Data Using Signal Logging” in the Simulink documentation.

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

Accessing Fixed-Point Block Data During Simulation

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

Using fi Objects with DSP System Toolbox

In this section...

“Reading Fixed-Point Signals from the Workspace” on page 9-7

“Writing Fixed-Point Signals to the Workspace” on page 9-7

“Using fi Objects with dfilt Objects” on page 9-11

Reading Fixed-Point Signals from the Workspace

You can read fixed-point data from the MATLAB workspace into a Simulink model using the Signal From Workspace and Triggered Signal From Workspace blocks from DSP System Toolbox™ software. Enter the name of the defined `fi` variable in the **Signal** parameter of the Signal From Workspace or Triggered Signal From Workspace block.

Writing Fixed-Point Signals to the Workspace

Fixed-point output from a model can be written to the MATLAB workspace via the Signal To Workspace or Triggered To Workspace block from the blockset. The fixed-point data is always written as a 2-D or 3-D array.

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

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

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

```
a =
```

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

```
      DataTypeMode: Fixed-point: binary point scaling
      Signedness: Signed
      WordLength: 16
      FractionLength: 15
```

The Signal From Workspace block in the following model has these settings:

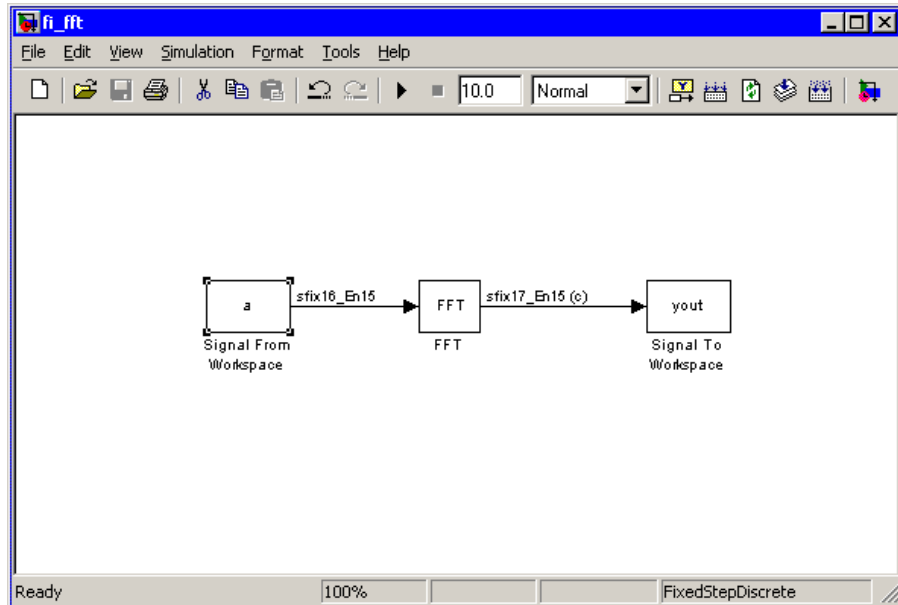
- **Signal** — a
- **Sample time** — 1
- **Samples per frame** — 2
- **Form output after final data value by** — Setting to zero

The following parameters in the **Solver** pane of the **Configuration Parameters** dialog have these settings:

- **Start time** — 0.0

- **Stop time** — 10.0
- **Type** — Fixed-step
- **Solver** — Discrete (no continuous states)
- **Fixed step size (fundamental sample time)** — 1.0

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



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

```
yout =
```

```
(:, :, 1) =
```

0.8415 -0.1319
-0.8415 -0.9561

(:, :, 2) =

1.0504 1.6463
0.7682 0.3324

(:, :, 3) =

-1.7157 -1.2383
0.2021 0.6795

(:, :, 4) =

0.3776 -0.6157
-0.9364 -0.8979

(:, :, 5) =

1.4015 1.7508
0.5772 0.0678

(:, :, 6) =

-0.5440 0
-0.5440 0

```
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 17
FractionLength: 15
```

Using `fi` Objects with `dfilt` Objects

When the `Arithmetic` property is set to `'fixed'`, you can use an existing `fi` object as the input, states, or coefficients of a `dfilt` object in DSP System Toolbox software. Also, fixed-point filters in the toolbox return `fi` objects as outputs. Refer to the DSP System Toolbox software documentation for more information.

Using `fiaccel`, MATLAB Coder, or Simulink to Generate Code

There are several ways to use Fixed-Point Toolbox software to generate code:

- The Fixed-Point Toolbox `fiaccel` function converts your fixed-point MATLAB code to a MEX function and can greatly accelerate the execution speed of your fixed-point algorithms.
- The MATLAB Coder `codegen` function automatically converts MATLAB code to C/C++ code. Using the MATLAB Coder software allows you to accelerate your MATLAB code that uses Fixed-Point Toolbox software. To use the `codegen` function with Fixed-Point Toolbox software, you also need to have a MATLAB Coder license. For more information, see “Tutorial: Generating C Code from MATLAB Code at the Command Line” in the MATLAB Coder documentation.
- The MATLAB Function block allows you to use MATLAB code in your Simulink models that generate embeddable C/C++ code. To use the MATLAB Function block with Fixed-Point Toolbox software, you also need a Simulink license. For more information on the MATLAB Function block, see “Using the MATLAB Function Block” in the Simulink documentation.

For more information on generating code with Fixed-Point Toolbox software, see Chapter 8, “Code Acceleration and Code Generation from MATLAB for Fixed-Point Algorithms”.

A

ANSI C

- compared with `fi` objects 2-22

arithmetic

- fixed-point 4-16

- with [Slope Bias] signals 4-22

arithmetic operations

- fixed-point 2-10

B

- binary conversions 2-25

C

casts

- fixed-point 2-19

Code generation

- fixed-point 8-1

code generation from MATLAB

- best practices

 - generate code generation report 8-49

 - preserving your code 8-52

 - separating test bench from function code 8-52

 - specifying input properties 8-49

 - using build scripts 8-50

 - using file naming convention 8-52

 - using the MATLAB code analyzer 8-51

- compiler options for MEX code generation 8-29

- controlling run-time checks 8-73

- how to disable run-time checks 8-74

- recommended options for `fiaccel` 8-49

- using Code Analyzer 8-26

- when to disable run-time checks 8-74

compiler options parameters

- for MEX code generation from MATLAB 8-29

compilers

- supported for generating MEX functions

 - with `fiaccel` 8-16

complex multiplication

- fixed-point 2-13

controlling run-time checks

- code generation from MATLAB 8-73

D

- data type override 5-12

- demos 1-11

display preferences

- setting 5-5

- display settings 1-7

documentation

- installing 1-3

F

`fi` objects

- constructing 3-2

`fiaccel`

- recommended options 8-49

- supported compilers 8-16

`fimath` objects 2-16

- constructing 4-2

- properties

 - setting in the Model Explorer 4-14

 - setting properties in the Model Explorer 4-14

`fipref` objects

- constructing 5-2

- fixed-point arithmetic 4-16

fixed-point data

- reading from workspace 9-2

- writing to workspace 9-2

fixed-point data types

- addition 2-12

- arithmetic operations 2-10

- casts 2-19

- complex multiplication 2-13

- modular arithmetic 2-10
- multiplication 2-13
- overflow handling 2-5
- precision 2-5
- range 2-5
- rounding 2-6
- saturation 2-5
- scaling 2-4
- subtraction 2-12
- two's complement 2-11
- wrapping 2-5

fixed-point math 4-16

Fixed-Point MATLAB code 8-1

fixed-point run-time API 9-6

fixed-point signal logging 9-6

H

help

- getting 1-5

how to disable run-time checks

- code generation from MATLAB 8-74

I

installation

- documentation 1-3
- Fixed-Point Toolbox 1-3

interoperability

- fi objects with DSP System Toolbox 9-7
- fi objects with Filter Design Toolbox 9-11
- fi objects with Simulink 9-2

L

licensing 1-4

logging

- overflows and underflows 5-7

logging modes

- setting 5-7

M

math

- with [Slope Bias] signals 4-22

MATLAB Function block

- using with Model Explorer and fixed-point models 8-79

Model Explorer

- setting `embedded.fimath` properties 4-14
- setting `embedded.numericitytype` properties 6-9
- using with fixed-point code generation for MATLAB 8-79

modular arithmetic 2-10

multiplication

- fixed-point 2-13

N

`numericitytype` objects

- constructing 6-2
- properties

 - setting in the Model Explorer 6-9
 - setting properties in the Model Explorer 6-9

O

one's complement 2-11

overflow handling 2-5

- compared with ANSI C 2-28

overflows

- logging 5-7

P

padding 2-19

precision

- fixed-point data types 2-5

property values

- quantizer objects 7-3

Q

quantizer objects
 constructing 7-2
 property values 7-3

R

range
 fixed-point data types 2-5
reading fixed-point data from workspace 9-2
rounding
 fixed-point data types 2-6
run-time API
 fixed-point data 9-6

S

saturation 2-5
scaling 2-4
signal logging

 fixed-point 9-6
 [Slope Bias] arithmetic 4-22

T

two's complement 2-11

U

unary conversions 2-24
underflows
 logging 5-7

W

when to disable run-time checks
 code generation from MATLAB 8-74
wrapping
 fixed-point data types 2-5
writing fixed-point data to workspace 9-2