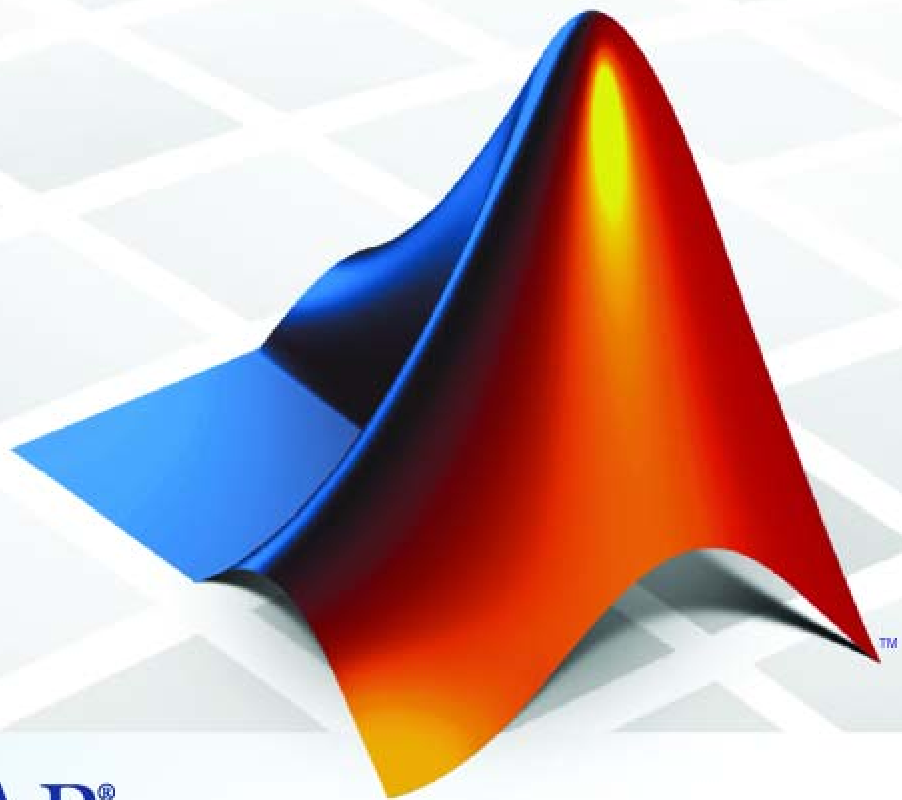
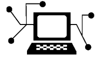
Fixed-Point Toolbox™ 2
User's Guide



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

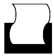
How to Contact The MathWorks



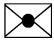
`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

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

508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

Fixed-Point Toolbox™ User's Guide

© COPYRIGHT 2004–2008 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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	Version 2.1 (R2007b)
March 2008	Online only	Version 2.2 (R2008a)

Contents

1 Getting Started

Product Overview 1-2

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

Demos 1-9

2 Fixed-Point Concepts

Fixed-Point Data Types 2-2

Scaling 2-4

Precision and Range 2-5
- Range 2-5
- Precision 2-6

Arithmetic Operations 2-8
- Modulo Arithmetic 2-8
- Two's Complement 2-9
- Addition and Subtraction 2-10
- Multiplication 2-11
- Casts 2-16

fi Objects Compared to C Integer Data Types **2-20**
Integer Data Types **2-20**
Unary Conversions **2-22**
Binary Conversions **2-23**
Overflow Handling **2-25**

3 Working with fi Objects

Constructing fi Objects **3-2**
fi Object Syntaxes **3-2**
Examples of Constructing fi Objects **3-4**

Casting fi Objects **3-14**
Overwriting by Assignment **3-14**
Ways to Cast in MATLAB® Software **3-14**

fi Object Properties **3-18**
Data Properties **3-18**
fimath Properties **3-18**
numerictype Properties **3-19**
Setting fi Object Properties **3-20**

fi Object Functions **3-23**

4 Working with fimath Objects

Constructing fimath Objects **4-2**

fimath Object Properties **4-4**
Math, Rounding, and Overflow Properties **4-4**
Setting fimath Object Properties **4-5**

Using fimath Objects to Perform Fixed-Point Arithmetic . . . 4-8
Binary Point Arithmetic . . . 4-8
[Slope Bias] Arithmetic . . . 4-12

Using fimath to Specify Rounding and Overflow Modes . . . 4-16

Using fimath to Share Arithmetic Rules . . . 4-17

Using fimath ProductMode and SumMode . . . 4-19
Example Setup . . . 4-19
FullPrecision . . . 4-20
KeepLSB . . . 4-21
KeepMSB . . . 4-22
SpecifyPrecision . . . 4-23

fimath Object Functions . . . 4-25

5 Working with fipref Objects

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

6 Working with numerictype Objects

Constructing numerictype Objects 6-2
numerictype Object Syntaxes 6-2
Examples of Constructing numerictype Objects 6-3

numerictype Object Properties 6-6
Data Type and Scaling Properties 6-6
Setting numerictype Object Properties 6-7

The numerictype Structure 6-10
Possible Values of the numerictype Structure Properties . . 6-10
Properties That Affect the Slope 6-11
Stored Integer Value and Real World Value 6-12

Using numerictype Objects to Share Data Type and Scaling Settings 6-13

numerictype Object Functions 6-16

7 Working with quantizer Objects

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

8 Working with the Fixed-Point Embedded MATLAB™ Subset

Supported Functions and Limitations of Fixed-Point Embedded MATLAB™ Subset 8-2

Fixed-Point Embedded MATLAB™ Subset Features 8-9
Embedded MATLAB™ MEX 8-9
Embedded MATLAB™ Function Block 8-12

9 Interoperability with Other Products

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 the Simulink® Embedded MATLAB™ Function Block 9-8
Using Fixed-Point Data Types with the Embedded MATLAB™ Function Block 9-8
Using the Embedded MATLAB™ Function Block with Data Type Override 9-9
Using the Model Explorer with a Fixed-Point Embedded MATLAB™ Function Block 9-10
Example: Implementing a Fixed-Point Direct Form FIR Using the Embedded MATLAB™ Function Block 9-14

Using Embedded MATLAB™ Coder **9-24**

Using fi Objects with Signal Processing Blockset™ Software **9-25**
Reading Fixed-Point Signals from the Workspace **9-25**
Writing Fixed-Point Signals to the Workspace **9-25**

Using fi Objects with Filter Design Toolbox™ Software **9-30**

Index

Getting Started

Product Overview (p. 1-2)	Describes Fixed-Point Toolbox™ software and its major features
Licensing (p. 1-4)	Describes the Fixed-Point Toolbox licensing model
Getting Help (p. 1-5)	Tells you how to get help on Fixed-Point Toolbox objects, properties, and functions
Display Settings (p. 1-7)	Describes the `fi` object display settings used in the code examples in this User's Guide
Demos (p. 1-9)	Lists the Fixed-Point Toolbox demos

Product Overview

Fixed-Point Toolbox™ software provides fixed-point data types in MATLAB® technical computing software and enables algorithm development by providing fixed-point arithmetic. The toolbox enables you to create the following types of objects:

- `fi` — Defines a fixed-point numeric object in the MATLAB workspace. Each `fi` object is composed of value data, a `fimath` object, and a `numerictype` object.
- `fimath` — Governs how overloaded arithmetic operators work with `fi` objects
- `fipref` — Defines the display, logging, and data type override preferences of `fi` objects
- `numerictype` — Defines the data type and scaling attributes of `fi` objects
- `quantizer` — Quantizes data sets

Fixed-Point Toolbox™ software provides you with

- The ability to define fixed-point data types, scaling, and rounding and overflow methods in the MATLAB workspace
- Bit-true real and complex simulation
- Basic fixed-point arithmetic
  - Arithmetic operators `+`, `-`, `*`, `.*` for binary point-only and real [Slope Bias] signals
  - Division using the `divide` function for binary point-only signals
- Arbitrary word length up to `intmax('uint16')` bits
- Logging of minimums, maximums, overflows, and underflows
- Data type override with singles, doubles, or scaled doubles
- Conversions between binary, hex, double, and built-in integers
- Relational, logical, and bitwise operators
- Matrix functions such as `ctranspose` and `horzcat`

- Statistics functions such as `max` and `min`
- Interoperability with Simulink®, Signal Processing Blockset™ software, Embedded MATLAB™ subset, and Filter Design Toolbox™ software
- Compatibility with the Simulink To Workspace and From Workspace blocks

Licensing

You can use `fi` objects with the `DataType` property set to `double` *without* a Fixed-Point Toolbox™ license when the `fipref` `LoggingMode` property is set 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 when the `fipref` `LoggingMode` property is 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 objects of Fixed-Point Toolbox™ software are discussed in the following chapters:

- 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 display of `fi` objects is determined by the `fipref` object. Throughout this User's Guide, code examples of `fi` objects are usually shown as they appear when the `fipref` properties are set as follows:

- `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'

a = fi(pi)

a =

    3.1416


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

In other cases, it makes sense to also show the `fimath` object display:

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

For example,

```
p = fipref('NumberDisplay', 'RealWorldValue',...
'NumericTypeDisplay', 'full', 'FimathDisplay', 'full')

p =

         NumberDisplay: 'RealWorldValue'
    NumericTypeDisplay: 'full'
         FimathDisplay: 'full'
           LoggingMode: 'Off'
      DataTypeOverride: 'ForceOff'

a = fi(pi)

a =

    3.1416

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

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

For more information, refer to Chapter 5, “Working with fipref Objects”.

Demos

You can access demos in the **Demos** tab of the **Help Navigator** window. 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 Embedded MATLAB™ MEX — Steps through generating a C-MEX function from M code, running the generated C-MEX function, and displaying the results

Fixed-Point Concepts

Fixed-Point Data Types (p. 2-2)	Defines fixed-point data types
Scaling (p. 2-4)	Discusses the types of scaling used in Fixed-Point Toolbox; binary point-only and [Slope Bias]
Precision and Range (p. 2-5)	Discusses the concepts behind arithmetic operations in Fixed-Point Toolbox
Arithmetic Operations (p. 2-8)	Introduces the concepts behind arithmetic operations in Fixed-Point Toolbox
fi Objects Compared to C Integer Data Types (p. 2-20)	Compares ANSI C integer data type ranges, conversions, and exception handling with those of `fi` objects

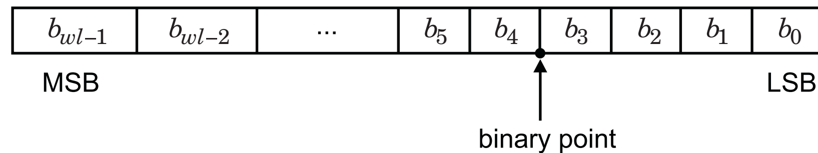
Fixed-Point Data Types

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 ith 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 2-9 for more information.

Scaling

Fixed-point numbers can be encoded according to the scheme

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

where the slope can be expressed as

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

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 fractional slope equal to one. This is referred to as binary point-only scaling or power-of-two scaling:

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

or

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

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 demo “`fi` Binary Point Scaling.”

Precision and Range

In this section…

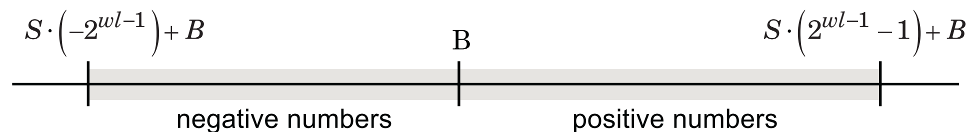
"Range" on page 2-5

"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.

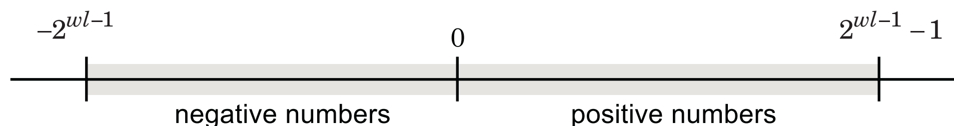
Range

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}:



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 2-8 for more information.

When you create a `fi` object, any overflows are saturated. The `OverflowMode` property of the default `fimath` object 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

One of the limitations of representing numbers with finite precision is that not every number in the available range can be represented exactly. When the result of a fixed-point calculation is a number that cannot be represented exactly by the data type and scaling being used, precision is lost. A rounding method must be used to cast the result to a representable number. 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 integer. In the case of a tie, it rounds to the nearest even stored integer. This is the least biased rounding method provided by the toolbox.
- `fix` rounds to the closest representable integer 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 integer. In the case of a tie, it rounds to the closest representable integer in the direction of positive infinity. This is the default rounding method for `fi` object creation and `fi` arithmetic.
- `round` rounds to the closest representable integer. In the case of a tie, it rounds positive numbers to the closest representable integer in the direction of positive infinity, and it rounds negative numbers to the closest representable integer in the direction of negative infinity.

Arithmetic Operations

In this section...

"Modulo Arithmetic" on page 2-8

"Two's Complement" on page 2-9

"Addition and Subtraction" on page 2-10

"Multiplication" on page 2-11

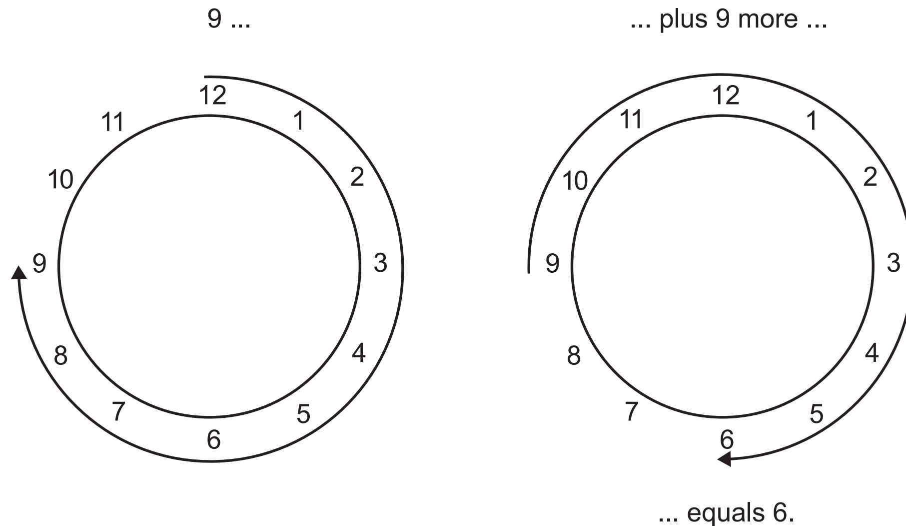
"Casts" on page 2-16

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,

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

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

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 1 using binary math.

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:

```
00101
   +1
-----
00110   (6)
```

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):

```
010010.1     (18.5)
 +0110.110   (6.75)
----------
011001.010   (25.25)
```

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):

```
010010.100   (18.5)                             010010.100   (18.5)
 -0110.110   (6.75)   two's complement  →      +111001.010   (-6.75)
----------            and sign extension       -----------
                                               1001011.110   (11.75)
```

Carry bit is discarded

The default `fimath` object 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):

```
      10.11  (–1.25)
        011  (3)
     ------
     11011
     1011
     ------
    1100.01  (–3.75)
```

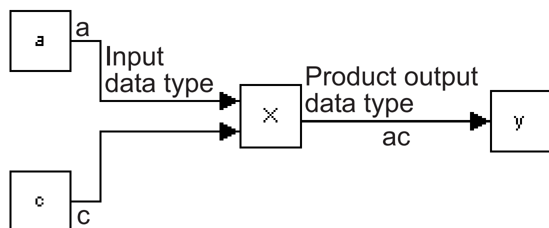
The extra 1 is the result of necessary sign extension.

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

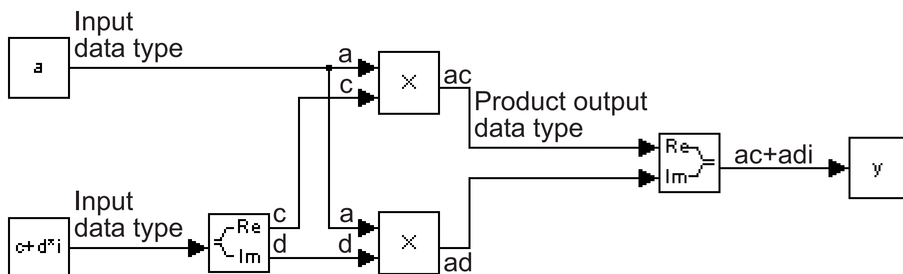
Multiplication Data Types

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.

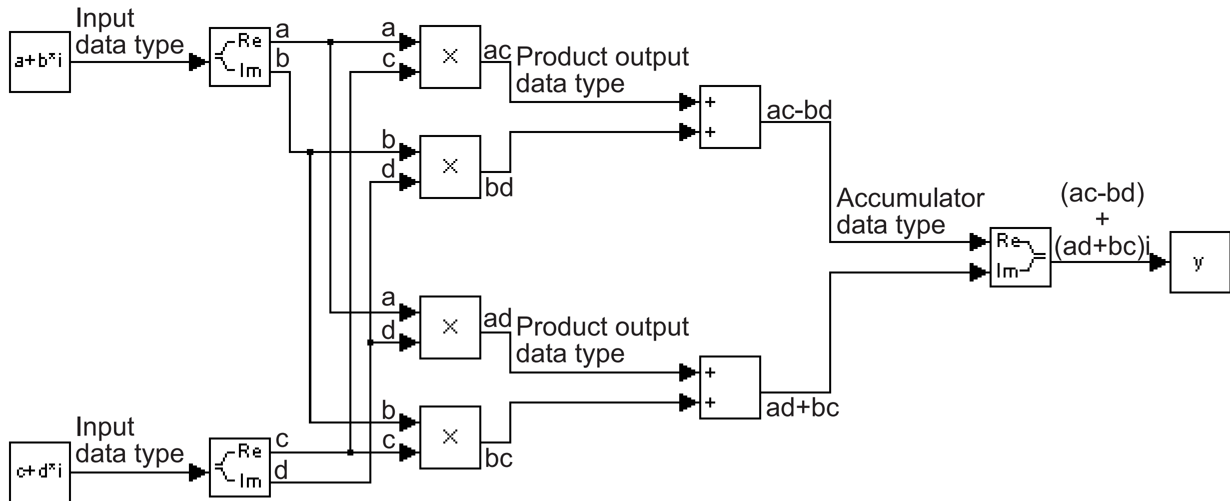
Real-Real Multiplication. The following diagram shows the data types used by the toolbox in the multiplication of two real numbers . The output of this multiplication is in the product data type, which is governed by the `fimath` `ProductMode` property:



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 output of this multiplication is in the product data type, which is governed by the `fimath` `ProductMode` property:



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



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
                Signed: true
            WordLength: 24
        FractionLength: 20

 y = fi(10, T2, F)

y =

    10


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

 z = x*y

z =

    50


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

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

y =

  10.0000 + 2.0000i


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

z = x*y

z =

  50.0000 +10.0000i


          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            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 = fi(5+6i,T1,F)

x =

   5.0000 + 6.0000i
```

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

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

y =

  10.0000 + 2.0000i


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

z = x*y

z =

  38.0000 +70.0000i


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

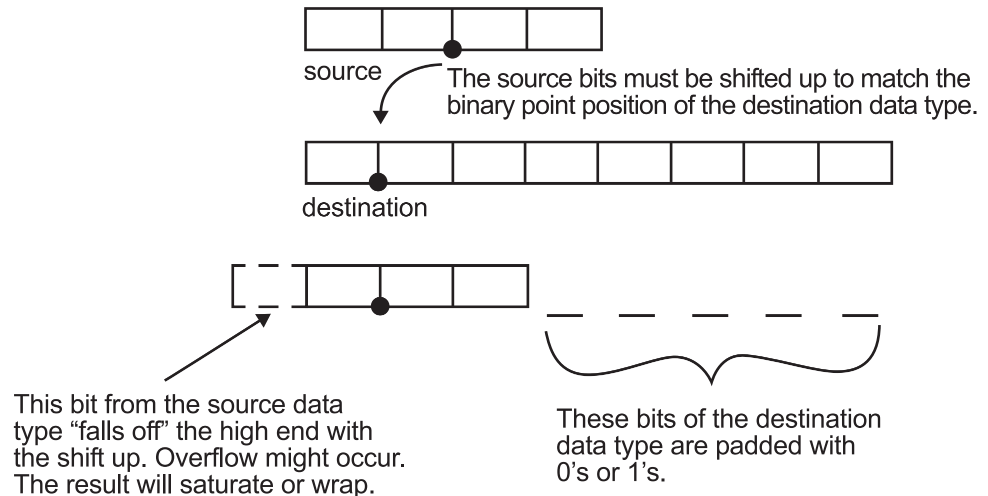
Casts

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 3-14.

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:



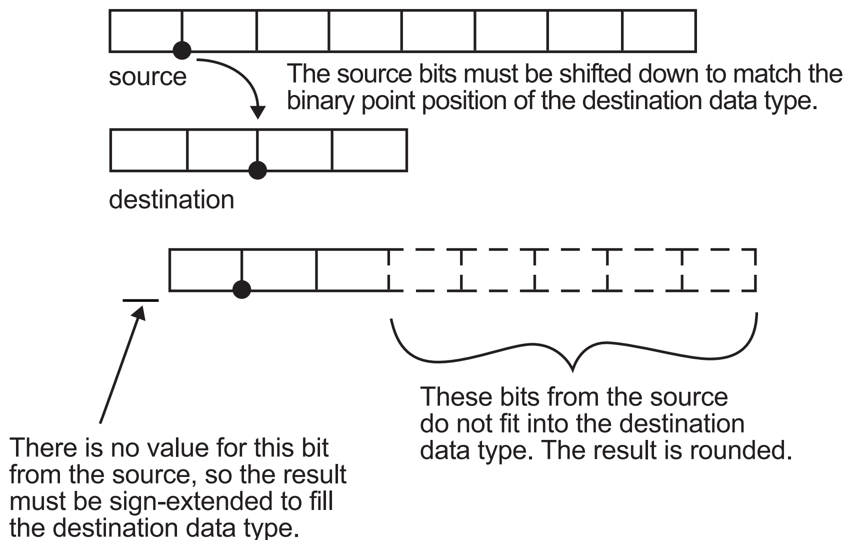
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 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 the result is sign-extended 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 2-20
"Unary Conversions" on page 2-22
"Binary Conversions" on page 2-23
"Overflow Handling" on page 2-25

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 ranges of ANSI C integer data types.

ANSI C Integer Data Types

The following table shows the minimum ranges of ANSI C integer data types. 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`.

Note that the minimum ANSI C ranges are large enough to accommodate one's complement or sign/magnitude representation, but not two's complement representation. In the one's complement and sign/magnitude representations, a signed integer with n bits has a range from $-2^{n-1}+1$ to $2^{n-1}-1$, inclusive. In both of these representations, an equal number of positive and negative numbers are represented, and zero is represented twice.

Integer Type	Minimum	Maximum
`signed char`	–127	127

Integer Type	Minimum	Maximum
`unsigned char`	0	255
`short int`	–32,767	32,767
`unsigned short`	0	65,535
`int`	–32,767	32,767
`unsigned int`	0	65,535
`long int`	–2,147,483,647	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. 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.

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

Constructor	Signed	Word Length	Fraction Length	Minimum	Maximum	Closest ANSI C Equivalent
`fi(x,0,16,0)`	No	16	0	0	65,535	`unsigned short`
`fi(x,1,32,0)`	Yes	32	0	–2,147,483,648	2,147,483,647	`long int`
`fi(x,0,32,0)`	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

[1]If 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`	`~x = 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. Overflow mode is obeyed: wrap or saturate if 1-valued bits are shifted off the left, or into the sign bit if the operand is signed. 0-valued bits are shifted in on the right.
`x>>n`	`bitshift(x,-n)`	Result is same numeric type as operand. Round mode is obeyed if 1-valued bits are shifted off the right. 0-valued bits are shifted in on the left if the operand is either signed and positive or unsigned. 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

[1]Type 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
fi	double or single	• Signed = same as the original fi operand • WordLength = same as the original fi operand • FractionLength = set to best precision possible
fi	int8	• 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	• Signed = 0 • WordLength = 8 • FractionLength = 0
fi	int16	• Signed = 1 • WordLength = 16 • FractionLength = 0
fi	uint16	• Signed = 0 • WordLength = 16 • FractionLength = 0
fi	int32	• Signed = 1 • WordLength = 32 • FractionLength = 0
fi	uint32	• Signed = 0 • WordLength = 32 • 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.
`ProductMode`	`'FullPrecision'`	Full-precision results are kept. Overflow does not occur. An error is thrown if the resulting word length is greater than `MaxProductWordLength`. The rules for computing the resulting product word and fraction lengths are given in "ProductMode" in the Property Reference.

fimath Object Properties Related to Overflow Handling	Property Value	Description
	'KeepLSB'	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. 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. The rule for computing the resulting product fraction length is given in “ProductMode” in the Property Reference.
	'KeepMSB'	The most significant bits of the product are kept. Overflow is prevented, but precision may be lost. 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. The rule for computing the resulting product fraction length is given in “ProductMode” in the Property Reference.
	'SpecifyPrecision'	You can specify both the word length and the fraction length of the resulting product.

fimath Object Properties Related to Overflow Handling	Property Value	Description
`ProductWordLength`	Positive integer	The word length of product results when `ProductMode` is `'KeepLSB'`, `'KeepMSB'`, or `'SpecifyPrecision'`.
`MaxProductWordLength`	Positive integer	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.
`ProductFractionLength`	Integer	The fraction length of product results when `ProductMode` is `'Specify Precision'`.
`SumMode`	`'FullPrecision'`	Full-precision results are kept. Overflow does not occur. An error is thrown if the resulting word length is greater than `MaxSumWordLength`. The rules for computing the resulting sum word and fraction lengths are given in "SumMode" in the Property Reference.

fimath Object Properties Related to Overflow Handling	Property Value	Description
	'KeepLSB'	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. 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. The rule for computing the resulting sum fraction length is given in "SumMode" in the Property Reference.
	'KeepMSB'	The most significant bits of the sum are kept. Overflow is prevented, but precision may be lost. 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. 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.

fimath Object Properties Related to Overflow Handling	Property Value	Description
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 (p. 3-2)	Teaches you how to create `fi` objects
Casting fi Objects (p. 3-14)	Shows you how to cast `fi` objects
fi Object Properties (p. 3-18)	Tells you how to find more information about the properties associated with `fi` objects, and shows you how to set these properties
fi Object Functions (p. 3-23)	Introduces the functions in the toolbox that operate directly on `fi` objects

Constructing fi Objects

In this section...

"fi Object Syntaxes" on page 3-2

"Examples of Constructing fi Objects" on page 3-4

fi Object Syntaxes

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

- You can use the `fi` constructor function to create a new object.
- You can use the `fi` constructor function 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
                Signed: true
            WordLength: 16
        FractionLength: 15
```

A signed `fi` object is created with a value of 0, word length of 16 bits, and fraction length of 15 bits.

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

You can use the `fi` constructor function in the following ways:

- `a = fi` is the default constructor and returns a `fi` object with no value, 16-bit word length, and 15-bit fraction length.
- `a = fi(v)` returns a signed fixed-point object with value `v`, 16-bit word length, and best-precision fraction length.
- `a = fi(v,s)` returns a fixed-point object with value `v`, signedness `s`, 16-bit word length, and best-precision fraction length. `s` can be `0` (false) for unsigned or `1` (true) for signed.
- `a = fi(v,s,w)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, and best-precision fraction length.
- `a = fi(v,s,w,f)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, and fraction length `f`.
- `a = fi(v,s,w,slope,bias)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, `slope`, and `bias`.
- `a = fi(v,s,w,slopeadjustmentfactor,fixedexponent,bias)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.
- `a = fi(v,T)` returns a fixed-point object with value `v` and `embedded.numerictype` `T`. Refer to Chapter 6, "Working with numerictype Objects" for more information on `numerictype` objects.
- `a = fi(v,F)` returns a fixed-point object with value `v`, `embedded.fimath` `F`, 16-bit word length, and best-precision fraction length. Refer to Chapter 4, "Working with fimath Objects" for more information on `fimath` objects.
- `b = fi(a,F)` allows you to maintain the value and `numerictype` object of `fi` object `a`, while changing its `fimath` object to `F`.
- `a = fi(v,T,F)` returns a fixed-point object with value `v`, `embedded.numerictype` `T`, and `embedded.fimath` `F`. The syntax `a = fi(v,T,F)` is equivalent to `a = fi(v,F,T)`.
- `a = fi(v,s,F)` returns a fixed-point object with value `v`, signedness `s`, 16-bit word length, best-precision fraction length, and `embedded.fimath` `F`.
- `a = fi(v,s,w,F)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, best-precision fraction length, and `embedded.fimath` `F`.

- `a = fi(v,s,w,f,F)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, fraction length `f`, and `embedded.fimath` `F`.
- `a = fi(v,s,w,slope,bias,F)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, `slope`, `bias`, and `embedded.fimath` `F`.
- `a = fi(v,s,w,slopeadjustmentfactor,fixedexponent,bias,F)` returns a fixed-point object with value `v`, signedness `s`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, `bias`, and `embedded.fimath` `F`.
- `a = fi(...'PropertyName',PropertyValue...)` and `a = fi('PropertyName',PropertyValue...)` allow you to set fixed-point objects for a `fi` object by property name/property value pairs.

Examples of Constructing fi Objects

For example, the following creates a `fi` object with a value of `pi`, a word length of 8 bits, and a fraction length of 3 bits:

```
a = fi(pi, 1, 8, 3)

a =

    3.1250

          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 8
        FractionLength: 3
```

The value `v` can also be an array:

```
a = fi((magic(3)/10), 1, 16, 12)

a =

    0.8000    0.1001    0.6001
    0.3000    0.5000    0.7000
    0.3999    0.8999    0.2000


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

```
              Signed: true
          WordLength: 16
      FractionLength: 12
```

If you omit the argument `f`, it is set automatically to the best precision possible:

```
a = fi(pi, 1, 8)

a =

    3.1563


         DataTypeMode: Fixed-point: binary point scaling
               Signed: true
           WordLength: 8
       FractionLength: 5
```

If you omit `w` and `f`, they are set automatically to 16 bits and the best precision possible, respectively:

```
a = fi(pi, 1)

a =

    3.1416


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

Constructing a fi Object with Property Name/Property Value Pairs

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

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

a =

    3.1415

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

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

Constructing a fi Object Using a numerictype Object

You can use a `numerictype` object to define a `fi` object:

```
T = numerictype

T =


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

a = fi(pi, T)

a =

    1.0000
```

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

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

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

```
F = fimath

F =

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

a = fi(pi, T, F)

a =

    1.0000


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

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

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. By default, the word length is 16 bits, and the scaling is best precision:

```
F = fimath

F =


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

F.OverflowMode = 'wrap'

F =


            RoundMode: nearest
         OverflowMode: wrap
          ProductMode: FullPrecision
```

```
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true

a = fi(pi, F)

a =

    3.1416

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

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

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

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

b =

    3.1416

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

             RoundMode: nearest
          OverflowMode: wrap
```

```
             ProductMode: FullPrecision
    MaxProductWordLength: 128
                 SumMode: FullPrecision
        MaxSumWordLength: 128
           CastBeforeSum: true

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

c =

    3.1406

            DataTypeMode: Fixed-point: binary point scaling
                  Signed: false
              WordLength: 8
          FractionLength: 6

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

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

d =

    3.1406

            DataTypeMode: Fixed-point: binary point scaling
                  Signed: false
              WordLength: 8
          FractionLength: 6

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

```
             SumMode: FullPrecision
    MaxSumWordLength: 128
       CastBeforeSum: true
```

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 fraction length of `14`:

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

T =

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

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

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

a =

    1.0000

          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 16
        FractionLength: 14
             RoundMode: nearest
          OverflowMode: saturate
           ProductMode: FullPrecision
  MaxProductWordLength: 128
               SumMode: FullPrecision
      MaxSumWordLength: 128
         CastBeforeSum: true
```

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
                Signed: false
            WordLength: 16
        FractionLength: 14

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

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
                Signed: true
            WordLength: 16
        FractionLength: 13
b = a

b =
```

```
3.1416

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

Casting fi Objects

In this section...

"Overwriting by Assignment" on page 3-14
"Ways to Cast in MATLAB® Software" on page 3-14

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 in 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 two ways to cast data in MATLAB:

- Casting by Subscripted Assignment

- Casting by Conversion 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', 'FimathDisplay', 'none');
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
                Signed: true
            WordLength: 8
        FractionLength: 0

T16 = numerictype(1,16,0)

T16 =


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

B = fi(32767,T16)

B =

       32767
      s16,0

A = fi(B, T8)

A =

   127
      s8,0
```

fi Object Properties

In this section...

"Data Properties" on page 3-18

"fimath Properties" on page 3-18

"numerictype 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`, `uint8`, `uint16`, and `uint32` to get the stored integer value of a `fi` object in these formats
- `oct` — Stored integer value of a `fi` object in octal

fimath Properties

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

- `fimath` — `fimath` object associated with a `fi` object

The following `fimath` properties are, by transitivity, also properties of a `fi` object. The properties of the `fimath` object listed below are always writable:

- `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 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

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 numeric type attributes of a `fi` object

The following `numerictype` properties are, by transitivity, also properties of a `fi` object. The properties of the `numerictype` object listed below are not writable once the `fi` object has been created. 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
- `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

These properties are described in detail in 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
                Signed: true
            WordLength: 16
        FractionLength: 13

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

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 `DataTypeMode` of `a`,

```
a.DataTypeMode

ans =

Fixed-point: binary point scaling
```

To set the `OverflowMode` of `a`,

```
a.OverflowMode = 'wrap'

a =

    3.1416


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

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

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 (p. 4-2)	Teaches you how to create `fimath` objects
fimath Object Properties (p. 4-4)	Tells you how to find more information about the properties associated with `fimath` objects, and shows you how to set these properties
Using fimath Objects to Perform Fixed-Point Arithmetic (p. 4-8)	Gives examples of using `fimath` objects to control the results of fixed-point arithmetic with `fi` objects
Using fimath to Specify Rounding and Overflow Modes (p. 4-16)	Gives an example that shows that the order in which you set overflow and rounding modes matters
Using fimath to Share Arithmetic Rules (p. 4-17)	Gives an example of using a `fimath` object to share modular arithmetic information among multiple `fi` objects
Using fimath ProductMode and SumMode (p. 4-19)	Shows the differences among the different settings of the `ProductMode` and `SumMode` properties
fimath Object Functions (p. 4-25)	Introduces the functions in the toolbox that operate directly on `fimath` objects

Constructing fimath Objects

`fimath` objects define the arithmetic attributes of `fi` objects. You can create `fimath` objects in Fixed-Point Toolbox™ software in one of two ways:

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

To get started, type

```
F = fimath
```

to create a default `fimath` object.

```
F = fimath

F =


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

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 4-5.

fimath Object Properties

In this section...
"Math, Rounding, and Overflow Properties" on page 4-4
"Setting fimath Object Properties" on page 4-5

Math, Rounding, and Overflow Properties

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

- `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
- `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

These properties are described in detail in 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 4-5.

Setting fimath Object Properties

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
         CastBeforeSum: true
```

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 this 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
         CastBeforeSum: true
```

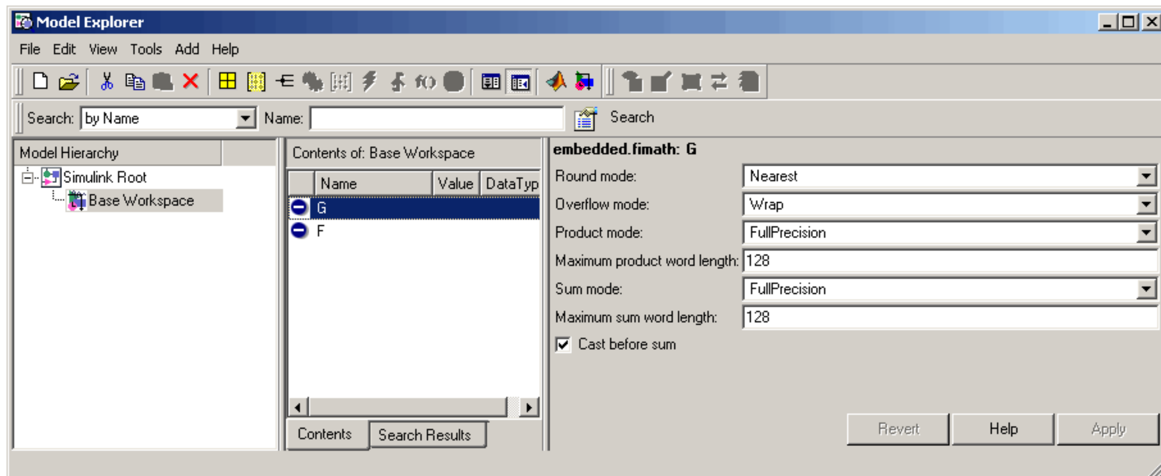
Setting fimath Properties in the Model Explorer

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 figure below 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
            CastBeforeSum: true

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

G =


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



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.

Using fimath Objects to Perform Fixed-Point Arithmetic

In this section...

"Binary Point Arithmetic" on page 4-8
"[Slope Bias] Arithmetic" on page 4-12

Binary Point Arithmetic

The `fimath` object encapsulates the math properties of Fixed-Point Toolbox™ software, and is itself a property of the `fi` object.

Every `fi` object has a `fimath` object as a property.

```
a = fi(pi)

a =

    3.1416


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

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

a.fimath

ans =
```

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

To perform arithmetic with `+`, `-`, `.*`, or `*`, two `fi` operands must have the same `fimath` properties.

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

ans =

     1

a + b

ans =

   11.1416


          DataTypeMode: Fixed-point: binary point scaling
                Signed: true
            WordLength: 19
        FractionLength: 13

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

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. Math is permitted between fixed-point and scaled doubles fi objects. 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
                Signed: true
            WordLength: 16
        FractionLength: 13

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

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

b =

     1     3
     4     2


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

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

```
    MaxProductWordLength: 128
                 SumMode: FullPrecision
        MaxSumWordLength: 128
           CastBeforeSum: true

a + b

ans =

    4.1416    6.1416
    7.1416    5.1416


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

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

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

[Slope Bias] Arithmetic

Fixed-point arithmetic using the `fimath` object is supported for all binary point-only signals. Arithmetic is also supported for [Slope Bias] signals with the following restrictions:

- [Slope Bias] signals must be real.
- The `fimath` object `SumMode` and `ProductMode` properties must be set to `'SpecifyPrecision'` for sum and multiply operations, respectively.
- The `fimath` object `CastBeforeSum` property must be set to `'true'`.

- The divide function is not supported 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
                        Signed: true
                    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
                        Signed: true
                    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
                        Signed: true
                    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, both the SumMode and ProductMode properties must be set to SpecifyPrecision for [Slope Bias] signals, because both sum and multiply operations are performed while calculating 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. Changing the rounding or overflow mode of a `fi` object after it has been created 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 `a` is created with `fimath` object `F` that has `OverflowMode` set to `wrap`, its value wrapped to -1. On the other hand, `b` is created with the default `OverflowMode` value of `saturate`, so its value is saturated 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 already happened when `b` was created, no change happens to `b` when it is assigned the new `fimath` object `F`, even though its rounding mode changed from `saturate` to `wrap`.

Using fimath to Share Arithmetic Rules

You can use a `fimath` object to define common arithmetic rules that you would like to use for many `fi` objects. You can then create multiple `fi` objects, using the same `fimath` object for each. To do so, you also need to 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, which are then used 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
                Signed: true
            WordLength: 32
        FractionLength: 30

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

F =


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

a = fi(pi, T, F)

a =

   -0.8584
```

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

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

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

b =

    1.5708


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

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

Using fimath ProductMode and SumMode

In this section...

"Example Setup" on page 4-19

"FullPrecision" on page 4-20

"KeepLSB" on page 4-21

"KeepMSB" on page 4-22

"SpecifyPrecision" on page 4-23

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 is automatically chosen 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 is aligned 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 is aligned 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 (p. 5-2)	Teaches you how to create `fipref` objects
fipref Object Properties (p. 5-3)	Tells you how to find more information about the properties associated with `fipref` objects, and shows you how to set these properties
Using fipref Objects to Set Display Preferences (p. 5-5)	Gives examples of using `fipref` objects to set display preferences for `fi` objects
Using fipref Objects to Set Logging Preferences (p. 5-7)	Gives examples of using `fipref` objects to set logging preferences for `fi` objects
Using fipref Objects to Set Data Type Override Preferences (p. 5-12)	Describes how to use the `fipref` object to perform data type override
fipref Object Functions (p. 5-15)	Introduces the functions in the toolbox that operate directly on `fipref` objects

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 `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 dictate three aspects of the display of `fi` objects: how the value of a `fi` object is displayed, how the `fimath` properties are displayed, and how the `numerictype` properties are displayed.

For example, the following shows the default `fipref` display for a `fi` object:

```
a = fi(pi)

a =

    3.1416


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

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

Now, 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 =
0110010010001000
      s16,13
```

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 an M-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 =

     3

minlog(x)

ans =

   -1.5000

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
                   Signed: true
               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.

6

Working with numerictype Objects

Constructing numerictype Objects (p. 6-2)	Teaches you how to create `numerictype` objects
numerictype Object Properties (p. 6-6)	Tells you how to find more information about the properties associated with `numerictype` objects, and shows you how to set these properties
The numerictype Structure (p. 6-10)	Presents the `numerictype` object as a MATLAB® structure, and gives the valid fields and settings for those fields
Using numerictype Objects to Share Data Type and Scaling Settings (p. 6-13)	Gives an example of using a `numerictype` object to share modular data type and scaling information among multiple `fi` objects
numerictype Object Functions (p. 6-16)	Introduces the functions in the toolbox that operate directly on `numerictype` objects

Constructing numerictype Objects

In this section...

"numerictype Object Syntaxes" on page 6-2

"Examples of Constructing numerictype Objects" on page 6-3

numerictype Object Syntaxes

`numerictype` objects define the data type and scaling attributes of `fi` objects. 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
                Signed: true
            WordLength: 16
        FractionLength: 15
```

You can use the `numerictype` constructor function in the following ways:

- `T = numerictype` creates a default `numerictype` object.
- `T = numerictype(s)` creates a `numerictype` object with `Fixed-point: unspecified scaling`, signedness `s`, and 16-bit word length.
- `T = numerictype(s,w)` creates a `numerictype` object with `Fixed-point: unspecified scaling`, signedness `s`, and word length `w`.

- `T = numerictype(s,w,f)` creates a `numerictype` object with `Fixed-point: binary point scaling`, signedness `s`, word length `w`, and fraction length `f`.
- `T = numerictype(s,w,slope,bias)` creates a `numerictype` object with `Fixed-point: slope and bias scaling`, signedness `s`, word length `w`, `slope`, and `bias`.
- `T = numerictype(s,w,slopeadjustmentfactor,fixedexponent,bias)` creates a `numerictype` object with `Fixed-point: slope and bias scaling`, signedness `s`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.
- `T = numerictype(property1,value1, ...)` allows you to set properties for a `numerictype` object using property name/property value pairs.
- `T = numerictype(T1, property1, value1, ...)` allows you to make a copy of an existing `numerictype` object, while modifying any or all of the property values.
- `T = numerictype('double')` creates a `double numerictype`.
- `T = numerictype('single')` creates a `single numerictype`.
- `T = numerictype('boolean')` creates a `Boolean numerictype`.

Examples of Constructing numerictype Objects

For example, the following creates a signed `numerictype` object with a 32-bit word length and 30-bit fraction length.

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

T =


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

If you omit the argument `f`, scaling is unspecified.

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

```
T =


            DataTypeMode: Fixed-point: unspecified scaling
                  Signed: true
              WordLength: 32
```

If you omit the arguments w and f, the word length is automatically set to 16 bits and the scaling is unspecified.

```
T = numerictype(1)

T =


            DataTypeMode: Fixed-point: unspecified scaling
                  Signed: true
              WordLength: 16
```

Constructing a numerictype Object with Property Name/Property Value Pairs

You can use property name/property value pairs to set numerictype properties when you create the object.

```
T = numerictype('Signed', true, 'DataTypeMode', ...
'Fixed-point: slope and bias', 'WordLength', 32, 'Slope', ...
2^-2, 'Bias', 4)

T =


            DataTypeMode: Fixed-point: slope and bias scaling
                  Signed: true
              WordLength: 32
                   Slope: 0.25
                    Bias: 4
```

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
```

numerictype Object Properties

In this section...

"Data Type and Scaling Properties" on page 6-6
"Setting numerictype Object Properties" on page 6-7

Data Type and Scaling Properties

All the properties of a `numerictype` object are writable. However, the `numerictype` properties of a `fi` object are not writable once the `fi` object has been created:

- `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
- `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 6-7.

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
                Signed: true
            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 =
```

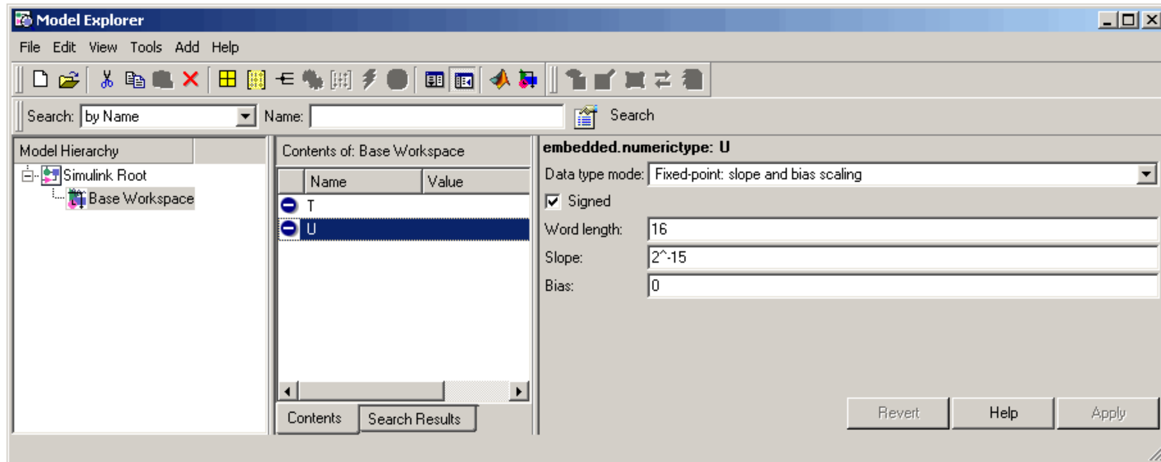
```
              DataTypeMode: Fixed-point: binary point scaling
                    Signed: true
                WordLength: 32
            FractionLength: 31
```

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
                    Signed: true
                WordLength: 16
            FractionLength: 15

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

U =


              DataTypeMode: Fixed-point: slope and bias scaling
                    Signed: true
                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...

"Possible Values of the numerictype Structure Properties" on page 6-10

"Properties That Affect the Slope" on page 6-11

"Stored Integer Value and Real World Value" on page 6-12

Possible Values of the numerictype Structure Properties

The `numerictype` object contains all the data type and scaling attributes of a `fi` object. The object acts the same way as 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 that are valid for `fi` objects.

DataTypeMode	DataType	Scaling	Signed	Word-Length	Fraction-Length	Slope	Bias
Fully specified fixed-point data types							
`Fixed-point: binary point scaling`	`Fixed`	`BinaryPoint`	`1/0` `true/false`	positive integer from 1 to 65,536	positive or negative integer	`1`	`0`
`Fixed-point: slope and bias scaling`	`Fixed`	`SlopeBias`	`1/0` `true/false`	positive integer from 1 to 65,536	`N/A`	any floating-point number	any floating-point number
Partially specified fixed-point data type							
`Fixed-point: unspecified scaling`	`Fixed`	`Unspecified`	`1/0` `true/false`	positive integer from 1 to 65,536	`N/A`	`N/A`	`N/A`

DataTypeMode	DataType	Scaling	Signed	Word-Length	Fraction-Length	Slope	Bias
Fully specified scaled double data types							
`Scaled double: binary point scaling`	`ScaledDouble`	`BinaryPoint`	`1/0` `true/ false`	positive integer from 1 to 65,536	positive or negative integer	1	0
`Scaled double: slope and bias scaling`	`ScaledDouble`	`SlopeBias`	`1/0` `true/ false`	positive integer from 1 to 65,536	N/A	any floating-point number	any floating-point number
Partially specified scaled double data type							
`Scaled double: unspecified scaling`	`ScaledDouble`	`Unspecified`	`1/0` `true/ false`	positive integer from 1 to 65,536	N/A	N/A	N/A
Built-in data types							
`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

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

The FixedExponent and FractionLength properties are related by

$$fixed\ exponent = -fraction\ length$$

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

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

which is equivalent to

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

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

Using numerictype Objects to Share Data Type and Scaling Settings

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. The following example shows the creation of a `numerictype` object, which is then used to create two `fi` objects with the same `numerictype` attributes:

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

T =


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

a = fi(pi,T)

a =

           3.1415926553309


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

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

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

b =

          1.5707963258028


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

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

The following example shows the creation of a `numerictype` object with [Slope Bias] scaling, which is then used to create two `fi` objects with the same `numerictype` attributes:

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

T =


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

```
c = fi(pi, T)

c =

     4

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

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

d = fi(pi/2, T)

d =

     0

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

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

numerictype Object Functions

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

Working with quantizer Objects

Constructing quantizer Objects (p. 7-2)	Explains how to create `quantizer` objects
quantizer Object Properties (p. 7-3)	Outlines the properties of the `quantizer` objects
Quantizing Data with quantizer Objects (p. 7-4)	Discusses using `quantizer` objects to quantize data—how and what quantizing data does
Transformations for Quantized Data (p. 7-6)	Offers a brief explanation of transforming quantized data between representations
quantizer Object Functions (p. 7-7)	Introduces the functions in the toolbox that operate directly on `quantizer` objects

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.

8

Working with the Fixed-Point Embedded MATLAB™ Subset

Supported Functions and Limitations of Fixed-Point Embedded MATLAB™ Subset (p. 8-2)	Lists the Fixed-Point Toolbox™ software features supported by the Embedded MATLAB™ subset
Fixed-Point Embedded MATLAB™ Subset Features (p. 8-9)	Introduces you to Embedded MATLAB MEX and the Embedded MATLAB Function block

Supported Functions and Limitations of Fixed-Point Embedded MATLAB™ Subset

The Embedded MATLAB™ subset is a restricted subset of the MATLAB® language that provides optimizations for:

- Generating efficient, production-quality C code for embedded applications. Embedded MATLAB subset restricts MATLAB semantics to meet the memory and data type requirements of embedded target environments.
- Accelerating fixed-point algorithms.

For more information about the Embedded MATLAB subset, refer to the "Embedded MATLAB" documentation. The Embedded MATLAB subset supports a significant number of Fixed-Point Toolbox™ functions, which are listed in the table below. The following general limitations always apply to the use of Fixed-Point Toolbox software with the Embedded MATLAB subset:

- `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 larger than 32 bits are not supported.
- It is illegal to change the `fimath` or `numerictype` of a given variable once it has been created.
- The `boolean` and `ScaledDouble` values of the `DataTypeMode` and `DataType` properties are not supported.
- `convergent` rounding is not supported.
- The `false` value of the `CastBeforeSum` property is not supported.
- The `numel` function works the same as MATLAB `numel` for `fi` objects in Embedded MATLAB subset, rather than returning `1` as in Fixed-Point Toolbox software.

To learn about the general limitations of the Embedded MATLAB subset that also apply to use with Fixed-Point Toolbox software, refer to "What Is the

Embedded MATLAB Subset?" in the Embedded MATLAB language subset documentation.

Fixed-Point Toolbox™ Functions Supported for Use with the Embedded MATLAB™ Language Subset

Function	Remarks/Limitations
`abs`	—
`all`	—
`any`	—
`bitand`	• Not supported for slope-bias scaled `fi` objects.
`bitandreduce`	—
`bitcmp`	—
`bitconcat`	—
`bitget`	—
`bitor`	• Not supported for slope-bias scaled `fi` objects.
`bitorreduce`	—
`bitreplicate`	—
`bitrol`	—
`bitror`	—
`bitset`	—
`bitshift`	—
`bitsliceget`	—
`bitsll`	—
`bitsra`	—
`bitsrl`	—
`bitxor`	• Not supported for slope-bias scaled `fi` objects.
`bitxorreduce`	—
`ceil`	—
`complex`	—

Fixed-Point Toolbox™ Functions Supported for Use with the Embedded MATLAB™ Language Subset (Continued)

Function	Remarks/Limitations
conj	—
convergent	—
ctranspose	—
diag	—
disp	—
divide	• Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object. • Complex and imaginary divisors are not supported.
double	—
end	—
eps	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, fi single and fi double signals.
eq	• Not supported for fixed-point signals with different biases.

Fixed-Point Toolbox™ Functions Supported for Use with the Embedded MATLAB™ Language Subset (Continued)

Function	Remarks/Limitations
`fi`	• Use to create a fixed-point constant or variable in the Embedded MATLAB language subset. • The default constructor syntax without any input arguments is not supported. • The syntax `fi('PropertyName',PropertyValue...)` is not supported. To use property name/property value pairs, you must first specify the value `v` of the `fi` object as in `fi(v,'PropertyName',PropertyValue...)`. • Works for all input values when complete `numerictype` information of the `fi` object is provided. • Works only for constant input values (value of input must be known at compile time) when complete `numerictype` information of the `fi` object is not specified. • `numerictype` object information must be available for nonfixed-point Simulink® inputs.
`fimath`	• Fixed-point signals coming in to an Embedded MATLAB Function block from Simulink are assigned the `fimath` object defined in the Embedded MATLAB Function block dialog in the Model Explorer. • Use to create `fimath` objects in Embedded MATLAB code.
`fix`	—
`floor`	—
`ge`	• Not supported for fixed-point signals with different biases.
`get`	• The syntax `structure = get(o)` is not supported.
`getlsb`	—
`getmsb`	—
`gt`	• Not supported for fixed-point signals with different biases.
`horzcat`	—
`imag`	—

Fixed-Point Toolbox™ Functions Supported for Use with the Embedded MATLAB™ Language Subset (Continued)

Function	Remarks/Limitations
int8, int16, int32	—
iscolumn	—
isempty	—
isfi	—
isfimath	—
isfinite	—
isinf	—
isnan	—
isnumeric	—
isnumerictype	—
isreal	—
isrow	—
isscalar	—
issigned	—
isvector	—
le	• Not supported for fixed-point signals with different biases.
length	—
logical	—
lowerbound	—
lsb	• Supported for scalar fixed-point signals only. • Supported for scalar, vector, and matrix, fi single and double signals.
lt	• Not supported for fixed-point signals with different biases.
max	—
min	—

Fixed-Point Toolbox™ Functions Supported for Use with the Embedded MATLAB™ Language Subset (Continued)

Function	Remarks/Limitations
minus	• Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object.
mtimes	• Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object.
ndims	—
ne	• Not supported for fixed-point signals with different biases.
nearest	—
numberofelements	• numberofelements and numel both work the same as MATLAB numel for fi objects in the Embedded MATLAB language subset.
numerictype	• Fixed-point signals coming in to an Embedded MATLAB Function block from Simulink are assigned a numerictype 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 numerictype objects in Embedded MATLAB code.
permute	—
plus	• Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object.
pow2	—
range	—
real	—
realmax	—
realmin	—
repmat	—
rescale	—
reshape	—
round	—

Fixed-Point Toolbox™ Functions Supported for Use with the Embedded MATLAB™ Language Subset (Continued)

Function	Remarks/Limitations
`sign`	—
`single`	—
`size`	—
`sqrt`	• Complex and [Slope Bias] inputs error out. • Negative inputs yield a 0 result.
`subsasgn`	—
`subsref`	—
`sum`	—
`times`	• Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object.
`transpose`	—
`tril`	—
`triu`	—
`uint8`, `uint16`, `uint32`	—
`uminus`	—
`uplus`	—
`upperbound`	—
`vertcat`	—

Fixed-Point Embedded MATLAB™ Subset Features

In this section...
"Embedded MATLAB™ MEX" on page 8-9
"Embedded MATLAB™ Function Block" on page 8-12

Embedded MATLAB™ MEX

Embedded MATLAB™ MEX converts M-code to C-MEX functions that contain Embedded MATLAB subset optimizations for automatically accelerating fixed-point algorithms to compiled C code speed in MATLAB®. For more information, refer to "Working with Embedded MATLAB MEX" in the Embedded MATLAB language subset documentation.

Speeding Up Fixed-Point Execution with the emlmex Function

The Embedded MATLAB `emlmex` function can greatly increase the execution speed of your algorithms; however, improper use of the function can also slow execution. In this example, you will use the `emlmex` 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 `emlmex` function.

Algorithm. 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.
%#eml
    [m,n] = size(A);
    w = get(A,'WordLength') + ceil(log2(m));
    f = get(A,'FractionLength');
    B = fi(zeros(1,n),true,w,f,fimath(A));
    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 `emlmex` function. To evaluate the performance of the `emlmex` function when the entire algorithm is compiled, run the following code. The first portion of m-code executes the algorithm using only MATLAB functions. The second portion of the code compiles the entire algorithm using the Embedded MATLAB `emlmex` function. The MATLAB `tic` and `toc` functions keep track of the run times for each method of execution.

```
% MATLAB
fipref('NumericTypeDisplay','short','FimathDisplay','none');
A = fi(randn(1000,10));
tic
B = fi_matrix_column_sum(A)
t_matrix_column_sum_m = toc

% Embedded MATLAB
emlmex fi_matrix_column_sum -o fi_matrix_column_sum_x -eg {A} ...
-I [matlabroot '/toolbox/fixedpoint/fidemos']
tic
B = fi_matrix_column_sum_x(A);
t_matrix_column_sum_eml = toc
```

Trial 2: Worst Performance. Compiling only the smallest unit of computation using the `emlmex` function leads to much slower execution. In some cases, the overhead that results from calling the `emlmex` function inside a nested loop can cause even slower execution than using MATLAB functions alone. To evaluate the performance of the `emlmex` function when only the smallest unit of computation is compiled, run the following code. The first portion of m-code executes the algorithm using only MATLAB functions. The second portion of the code compiles the smallest unit of computation with the `emlmex` 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,fimath(A));
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

    % Embedded MATLAB
    emlmex fi_scalar_sum -o fi_scalar_sum_x -eg {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,fimath(A));
    for j = 1:n
        for i = 1:m
            B(j) = fi_scalar_sum_x(B(j),A(i,j));
            % B(j) = B(j) + A(i,j);
        end
    end
    t_scalar_sum_eml = 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 `emlmex` (`t_matrix_column_sum_eml`) and where only the scalar sum was compiled (`t_scalar_sum_eml`). Even the M-file with no `emlmex` compilation (`t_matrix_column_sum_m`) did better than when only the smallest unit of computation was compiled using `emlmex` (`t_scalar_sum_eml`).

X (Overall Performance Rank)	Time	X/Best	X_m/X_eml
Trial 1: Best Performance			
t_matrix_column_sum_m (2)	1.99759	84.4917	84.4917
t_matrix_column_sum_eml (1)	0.0236424	1	

X (Overall Performance Rank)	Time	X/Best	X_m/X_eml
Trial 2: Worst Performance			
t_scalar_sum_m (4)	10.2067	431.71	2.08017
t_scalar_sum_eml (3)	4.90664	207.536	

Using Data Type Override with Embedded MATLAB™ MEX

Fixed-Point Toolbox™ software ships with a demonstration of how to generate a C-MEX function from M-code. The M-code takes the weighted average of a signal to create a lowpass filter. To run the demo, click the Fixed-Point Lowpass Filtering Using Embedded MATLAB MEX link and follow the instructions in the right pane of the Help browser.

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 M-file using the `emlmex` command in the "Compile the M-File into a MEX File" section of the demo, the resulting MEX-function uses floating-point data.

Embedded MATLAB™ Function Block

The Embedded MATLAB Function block lets you compose a MATLAB language function in a Simulink® model that generates embeddable code using the Embedded MATLAB subset. When you simulate the model or generate code for a target environment, a function in an Embedded MATLAB Function block generates efficient C code. This code meets the strict memory and data type requirements of embedded target environments. In this way, Embedded MATLAB Function blocks bring the power of MATLAB for the

embedded environment into Simulink. For more information, refer to “Using the Simulink® Embedded MATLAB™ Function Block” on page 9-8.

Interoperability with Other Products

Using fi Objects with Simulink® (p. 9-2)	Describes the ways you can use Fixed-Point Toolbox™ `fi` objects with Simulink® models
Using the Simulink® Embedded MATLAB™ Function Block (p. 9-8)	Discusses the use of Fixed-Point Toolbox software with the Simulink Embedded MATLAB™ Function block
Using Embedded MATLAB™ Coder (p. 9-24)	Introduces Embedded MATLAB Coder, which enables you to accelerate fixed-point MATLAB® code
Using fi Objects with Signal Processing Blockset™ Software (p. 9-25)	Describes how to pass fixed-point data back and forth between the MATLAB workspace and Simulink models using Signal Processing Blockset™ blocks
Using fi Objects with Filter Design Toolbox™ Software (p. 9-30)	Provides a brief description of how to use `fi` objects with `dfilt` objects in Filter Design Toolbox™ software

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
                Signed: true
            WordLength: 16
        FractionLength: 15

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

s.signals.values = a

s =

    signals: [1x1 struct]
```

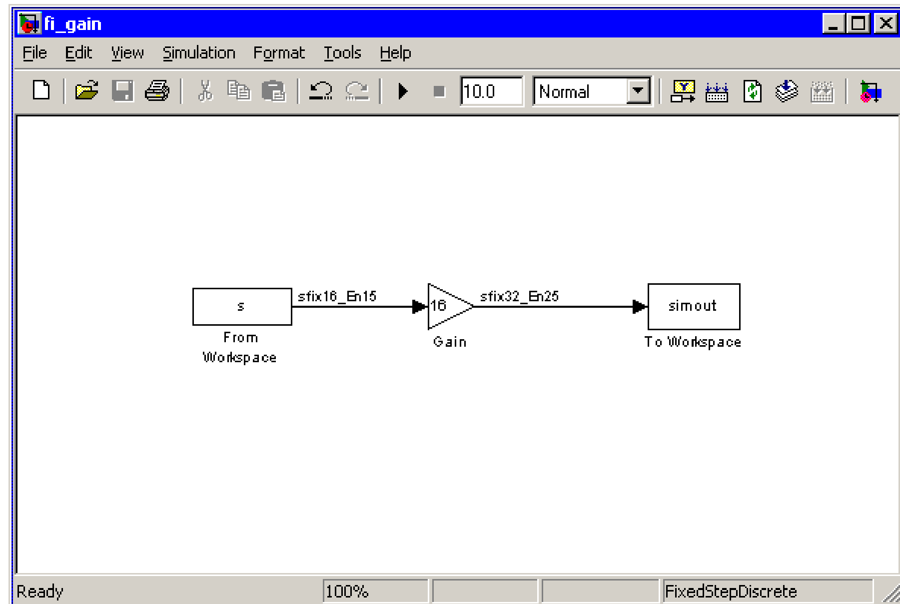
```
s.signals.dimensions = 2

s =

    signals: [1x1 struct]

s.time = [0:10]'

s =

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

The From Workspace block in the following model has the `fi` structure `s` in the **Data** parameter.

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
          Signed: true
      WordLength: 32
  FractionLength: 25

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

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 “Logging Signals” 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 the Simulink® Embedded MATLAB™ Function Block

In this section...

"Using Fixed-Point Data Types with the Embedded MATLAB™ Function Block" on page 9-8

"Using the Embedded MATLAB™ Function Block with Data Type Override" on page 9-9

"Using the Model Explorer with a Fixed-Point Embedded MATLAB™ Function Block" on page 9-10

"Example: Implementing a Fixed-Point Direct Form FIR Using the Embedded MATLAB™ Function Block" on page 9-14

Using Fixed-Point Data Types with the Embedded MATLAB™ Function Block

The Embedded MATLAB Function block lets you compose a MATLAB® language function in a Simulink® model that generates embeddable code using the Embedded MATLAB™ subset. When you simulate the model or generate code for a target environment, a function in an Embedded MATLAB Function block generates efficient C code. This code meets the strict memory and data type requirements of embedded target environments. In this way, Embedded MATLAB Function blocks bring the power of MATLAB for the embedded environment into Simulink.

For more information about the Embedded MATLAB Function block and the Embedded MATLAB subset, refer to the following documentation:

- Embedded MATLAB Function block reference page in the Simulink documentation
- "Using the Embedded MATLAB Function Block" in the Simulink documentation
- "Working with the Embedded MATLAB Subset" in the Embedded MATLAB documentation

A significant number of Fixed-Point Toolbox™ functions are supported by the Embedded MATLAB subset. Refer to "Supported Functions and Limitations

of Fixed-Point Embedded MATLAB™ Subset” on page 8-2 for information about which Fixed-Point Toolbox features are supported by the Embedded MATLAB subset.

Note To simulate models using fixed-point data types in Simulink, you must have a Simulink® Fixed Point™ license.

Using the Embedded MATLAB™ Function Block with Data Type Override

When you use the Embedded 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 and uses these to run the simulation. The following table shows how the Embedded MATLAB Function block determines the data type override equivalent from the data type of the input signal or parameter and the data type override setting in the Simulink model.

Note The Embedded MATLAB Function block does not support the `Scaled doubles` data type override setting.

Input Signal or Parameter Type	Data Type Override Setting	Data Type Override Equivalent
Inherited `single`	`True doubles`	`fi double`
	`True singles`	`fi single`
Specified `single`	`True doubles`	Built-in `double`
	`True singles`	Built-in `single`
Inherited `double`	`True doubles`	`fi double`
	`True singles`	`fi single`
Specified `double`	`True doubles`	Built-in `double`
	`True singles`	Built-in `single`

Input Signal or Parameter Type	Data Type Override Setting	Data Type Override Equivalent
Inherited `Fixed`	`True doubles`	`fi double`
	`True singles`	`fi single`
Specified `Fixed`	`True doubles`	`fi double`
	`True singles`	`fi single`

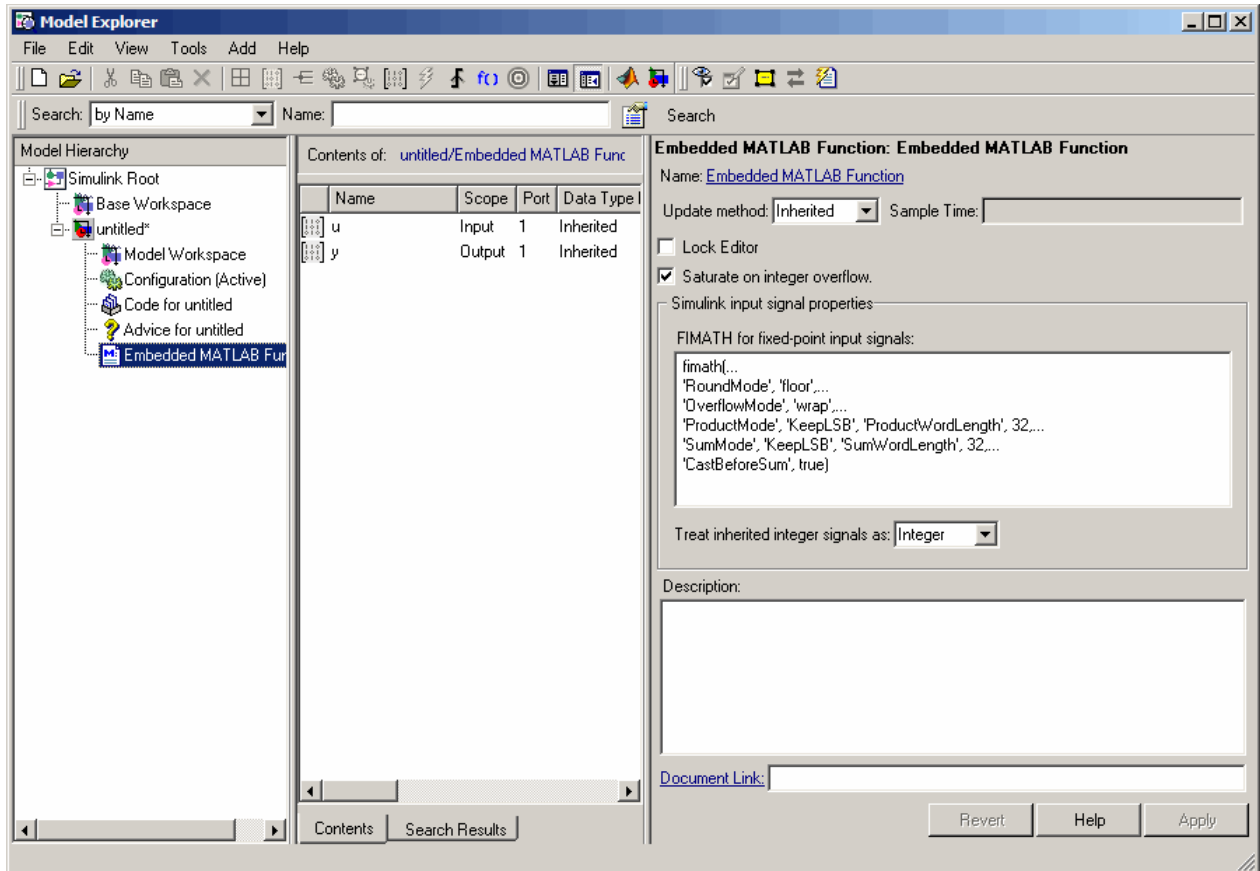
Using the Model Explorer with a Fixed-Point Embedded MATLAB™ Function Block

You can specify parameters for an Embedded MATLAB Function block in a fixed-point model using the Model Explorer. Try the following:

1. Place an Embedded MATLAB Function block in a new model. The block is located 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 and select the **Embedded MATLAB Function** node. The Model Explorer now appears as follows:



The parameters in the **Simulink input signal properties** group box in the **Dialog** pane apply to Embedded MATLAB Function blocks in models that use fixed-point data types.

FIMATH for fixed-point input signals

Define the `fimath` object to be associated with Simulink fixed-point or integer signals entering the Embedded MATLAB Function block as inputs. You can do this in either of two ways:

- Fully define the `fimath` object in the parameter value box using Fixed-Point Toolbox MATLAB code.

- Enter a variable name of a `fimath` object that is defined in the MATLAB or model workspace.

The default `fimath` object entered for this parameter emulates C-style math.

Treat inherited integer signals as
Choose whether to treat inherited integer signals as integers or fixed-point data.

- When you select `Integer`, Simulink integer inputs to the Embedded MATLAB Function block are treated as MATLAB integers.
- When you select `Fixed-point`, Simulink integer inputs to the Embedded MATLAB Function block are treated as Fixed-Point Toolbox `fi` objects.

Sharing Models Containing Fixed-Point Embedded MATLAB™ Function Blocks

Sometimes you might need to share a fixed-point model using the Embedded MATLAB Function block with a coworker. When you do, make sure to move any variables you define in the MATLAB workspace, including `fimath` objects, to the model workspace. For example, try the following:

1. Place an Embedded MATLAB Function block in a new model. The block is located 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 Embedded 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
```

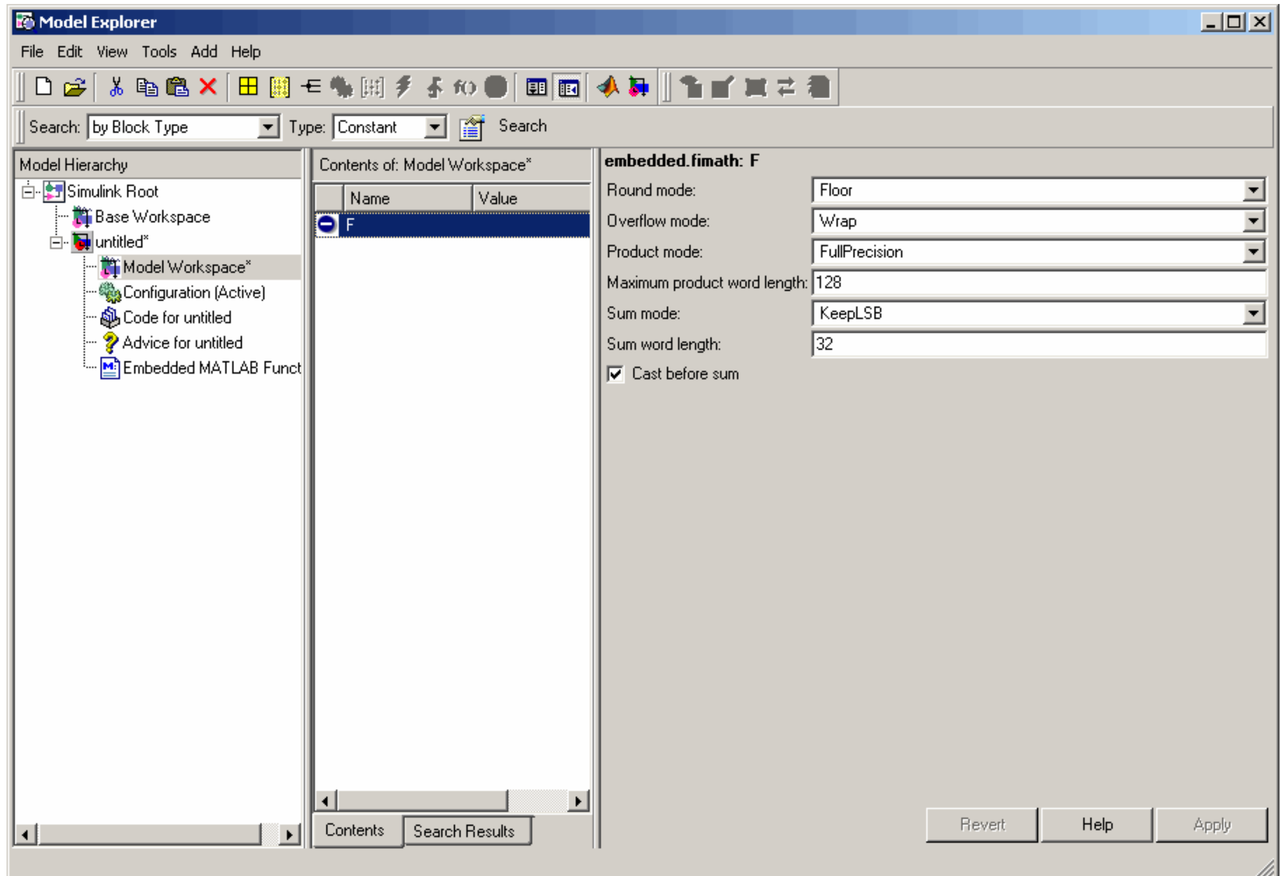
3. Open the Model Explorer by selecting **View** > **Model Explorer** from your model.

4. Expand the **untitled*** node in the **Model Hierarchy** pane of the Model Explorer and select the **Embedded MATLAB Function** node.

5. Enter the variable `F` into the **FIMATH for fixed-point input signals** parameter on the **Dialog** pane and click **Apply**. You have now defined the `fimath` object for any Simulink fixed-point signal entering the Embedded MATLAB Function block as an input.

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 were to send this model to a coworker, that coworker would have to define that same variable in the MATLAB workspace to get the same results as you with this model.

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 looks like this:



You can now e-mail your model to a coworker, and because the variables needed to run the model are included in the workspace of the model itself, your coworker can run the model and get the correct results without performing any extra steps.

Example: Implementing a Fixed-Point Direct Form FIR Using the Embedded MATLAB™ Function Block

The following sections lead you through creating a fixed-point, low-pass, direct form FIR filter in Simulink using Fixed-Point Toolbox software and the Embedded MATLAB Function block. You will perform the following tasks in the sequence shown:

- "Program the Embedded MATLAB™ Function Block" on page 9-15
- "Prepare the Inputs" on page 9-17
- "Create the Model" on page 9-17
- "Define the Input fimath Using the Model Explorer" on page 9-20
- "Run the Simulation" on page 9-22

Program the Embedded MATLAB™ Function Block

1 Place an Embedded MATLAB Function block in a new model. The block is located in the Simulink User-Defined Functions library.

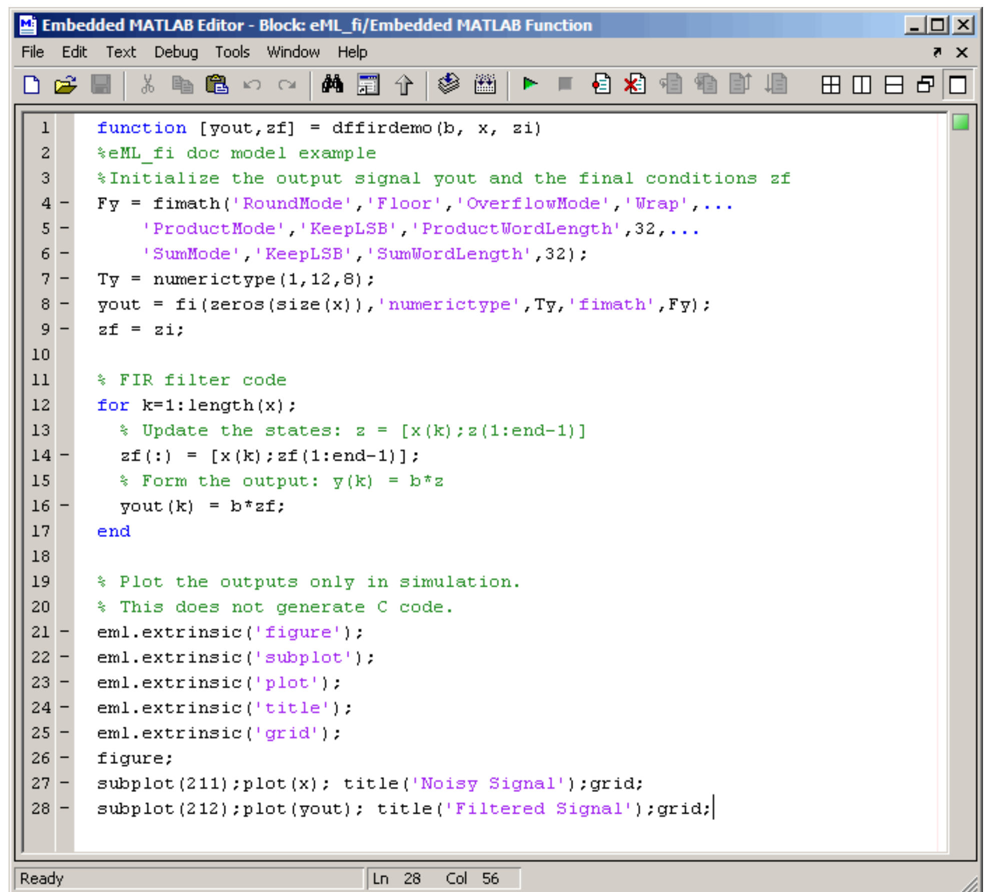
2 Save your model as `eML_fi.mdl`.

3 Double-click the Embedded MATLAB Function block in your model to open the Embedded MATLAB Editor. Type or copy and paste the following MATLAB code, including comments, into the Editor:

```
function [yout,zf] = dffirdemo(b, x, zi)
%eML_fi doc model example
%Initialize the output signal yout and the final conditions zf
Fy = fimath('RoundMode','Floor','OverflowMode','Wrap',...
    'ProductMode','KeepLSB','ProductWordLength',32,...
    'SumMode','KeepLSB','SumWordLength',32);
Ty = numerictype(1,12,8);
yout = fi(zeros(size(x)),'numerictype',Ty,'fimath',Fy);
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.
eml.extrinsic('figure');
```

```
eml.extrinsic('subplot');
eml.extrinsic('plot');
eml.extrinsic('title');
eml.extrinsic('grid');
figure;
subplot(211);plot(x); title('Noisy Signal');grid;
subplot(212);plot(yout); title('Filtered Signal');grid;
```

The Editor should now appear as follows:



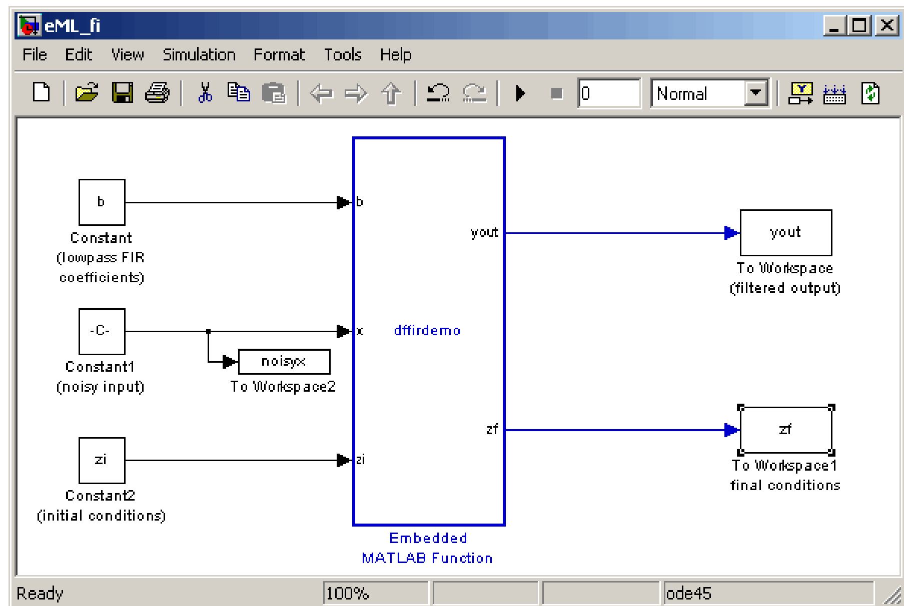
Prepare the Inputs

Define the filter coefficients *b*, noise *x*, and initial conditions *zi* by typing the following 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 system shown below.



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`
	Output data type mode	`Specify via dialog`
	Output data type	`sfix(12)`
	Output scaling mode	`Use specified scaling`
	Output scaling value	`2^-12`
Constant1	**Constant value**	`x+noise`
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	`Sample based`
	Sample time	`1`
	Output data type mode	`Specify via dialog`
	Output data type	`sfix(12)`
	Output scaling mode	`Use specified scaling`
	Output scaling value	`2^-8`

Block	Parameter	Value
Constant2	**Constant value**	`zi`
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	`Sample based`
	Sample time	`inf`
	Output data type mode	`Specify via dialog`
	Output data type	`sfix(12)`
	Output scaling mode	`Use specified scaling`
	Output scaling value	`2^-8`
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

Define the Input fimath Using the Model Explorer

1 Define the `fimath` object to be used for the Embedded MATLAB Function block inputs in the MATLAB workspace. Note that it must have the same properties as the `fimath` object defined in your Embedded MATLAB code in order to perform arithmetic between the quantities:

```
F_in = fimath('RoundMode','Floor','OverflowMode','Wrap',...
    'ProductMode','KeepLSB','ProductWordLength',32,...
    'SumMode','KeepLSB','SumWordLength',32)

F_in =


             RoundMode: floor
          OverflowMode: wrap
```

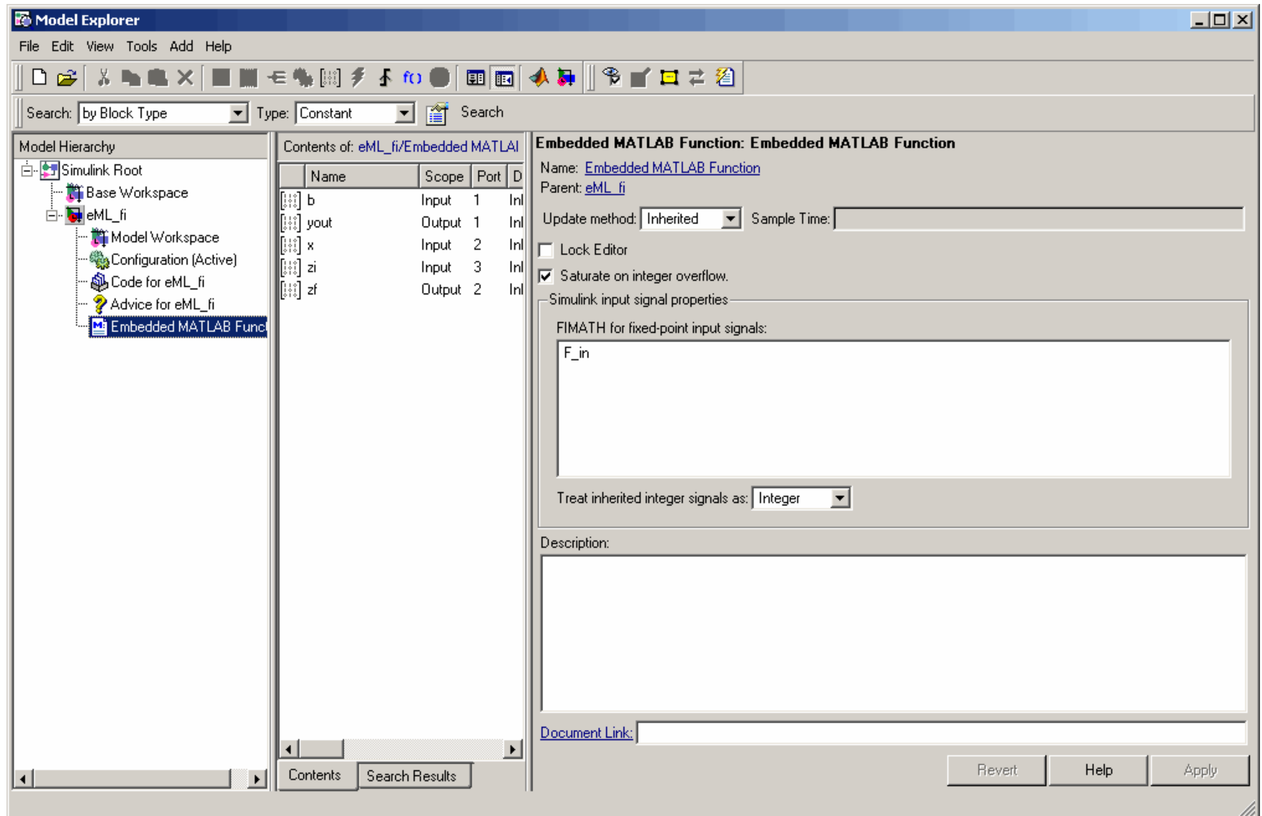
```
            ProductMode: KeepLSB
      ProductWordLength: 32
                SumMode: KeepLSB
          SumWordLength: 32
          CastBeforeSum: true
```

2. Open the Model Explorer for the model by selecting **View** > **Model Explorer**.

3. Click the **Base Workspace** node in the **Model Hierarchy** pane of the Model Explorer. You see the `fimath` `F_in` you just defined listed in the **Contents** pane.

4. Click the **eML_fi** > **Embedded MATLAB Function** node in the **Model Hierarchy** pane. The dialog for the Embedded MATLAB Function block appears in the **Dialog** pane of the Model Explorer.

5. Enter `F_in` in the **FIMATH for fixed-point input signals** parameter on the Embedded MATLAB Function block dialog in the **Dialog** pane of the Model Explorer and click **Apply**. This step sets the `fimath` object for the three inputs entering into the Embedded MATLAB Function block in your model. The Model Explorer now appears as follows:



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. Now 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 directory called `eML_fi_grt_rtw` is created in your current working directory.

3 Navigate to `eML_fi_grt_rtw` > `eML_fi.c`. In this file you can see the code that has been generated from your model. Search on the comment in your code

```
%eML_fi doc model example
```

This brings you to the beginning of the section of the code that is generated from your Embedded MATLAB Function block.

Using Embedded MATLAB™ Coder

Embedded MATLAB™ Coder is a Real-Time Workshop® function (`emlc`) that automatically converts M-code directly to C code. It lets you accelerate MATLAB® code that uses Fixed-Point Toolbox™ software. For more information, refer to "Working with Embedded MATLAB Coder" in the Real-Time Workshop product documentation.

Using fi Objects with Signal Processing Blockset™ Software

In this section...

"Reading Fixed-Point Signals from the Workspace" on page 9-25

"Writing Fixed-Point Signals to the Workspace" on page 9-25

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 Signal Processing Blockset™ 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
                   Signed: true
               WordLength: 16
           FractionLength: 15

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

The Signal From Workspace block in the following model has these settings:

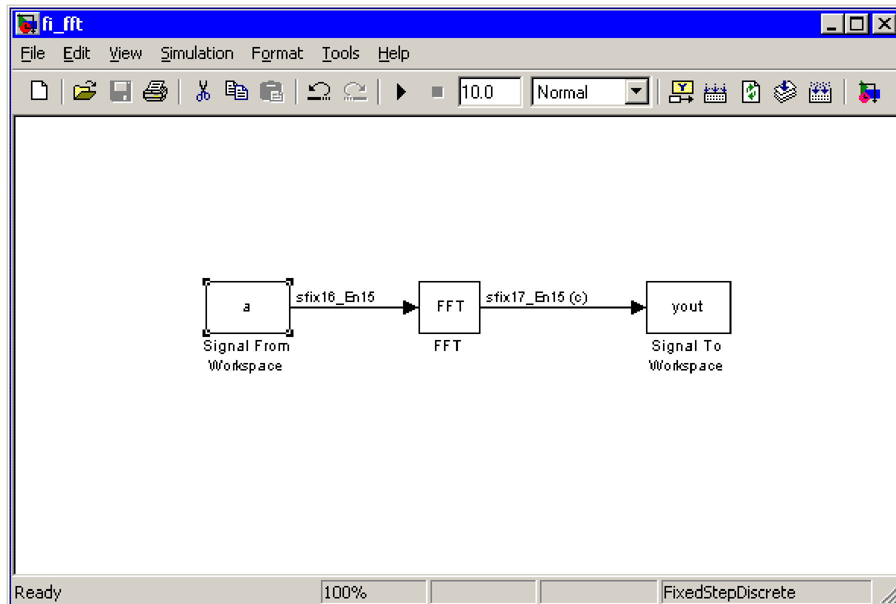
- **Signal** — `a`
- **Sample time** — `1`
- **Samples per frame** — `2`

- **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
                   Signed: true
               WordLength: 17
           FractionLength: 15

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

Using fi Objects with Filter Design Toolbox™ Software

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 Filter Design Toolbox™ software. Also, fixed-point filters in the toolbox return `fi` objects as outputs. Refer to the Filter Design Toolbox software documentation for more information.

Index

A

ANSI C
- compared with `fi` objects 2-20

arithmetic
- fixed-point 4-8
- with [Slope Bias] signals 4-12

arithmetic operations
- fixed-point 2-8

B

binary conversions 2-23

C

casts
- fixed-point 2-16

complex multiplication
- fixed-point 2-11

D

data type override 5-12
demos 1-9
display preferences
- setting 5-5

display settings 1-7

E

Embedded MATLAB
- fixed-point 8-1

Embedded MATLAB Function block
- Fixed-Point Toolbox support 9-8
- using with Model Explorer and fixed-point models 9-10

F

`fi` objects
- constructing 3-2

`fimath` objects 2-13
- constructing 4-2
- properties
  - setting in the Model Explorer 4-6
- setting properties in the Model Explorer 4-6

`fipref` objects
- constructing 5-2

fixed-point arithmetic 4-8
fixed-point data
- reading from workspace 9-2
- writing to workspace 9-2

fixed-point data types
- addition 2-10
- arithmetic operations 2-8
- casts 2-16
- complex multiplication 2-11
- modular arithmetic 2-8
- multiplication 2-11
- overflow handling 2-5
- precision 2-5
- range 2-5
- rounding 2-6
- saturation 2-5
- scaling 2-4
- subtraction 2-10
- two's complement 2-9
- wrapping 2-5

Fixed-Point Embedded MATLAB 8-1
fixed-point math 4-8
fixed-point models
- Embedded MATLAB Function block support 9-8

fixed-point run-time API 9-6
fixed-point signal logging 9-6
Fixed-Point Toolbox
- Embedded MATLAB Function block support 9-8

H

help
    getting 1-5

I

interoperability
    fi objects with Filter Design Toolbox 9-30
    fi objects with Signal Processing Blockset 9-25
    fi objects with Simulink 9-2
    Fixed-Point Toolbox with Embedded MATLAB Function block 9-8

L

licensing 1-4
logging
    overflows and underflows 5-7
logging modes
    setting 5-7

M

math
    with [Slope Bias] signals 4-12
Model Explorer
    setting `embedded.fimath` properties 4-6
    setting `embedded.numerictype` properties 6-8
    using with fixed-point Embedded MATLAB 9-10
modular arithmetic 2-8
multiplication
    fixed-point 2-11

N

`numerictype` objects
    constructing 6-2
    properties
        setting in the Model Explorer 6-8
    setting properties in the Model Explorer 6-8

O

one's complement 2-9
overflow handling 2-5
    compared with ANSI C 2-25
overflows
    logging 5-7

P

padding 2-17
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 4-12

T

two's complement 2-9

U

unary conversions 2-22
underflows
logging 5-7

W

wrapping
fixed-point data types 2-5
writing fixed-point data to workspace 9-2