Fixed-Point Toolbox™ 3
User's Guide

MATLAB®

The MathWorks™
Accelerating the pace of engineering and science
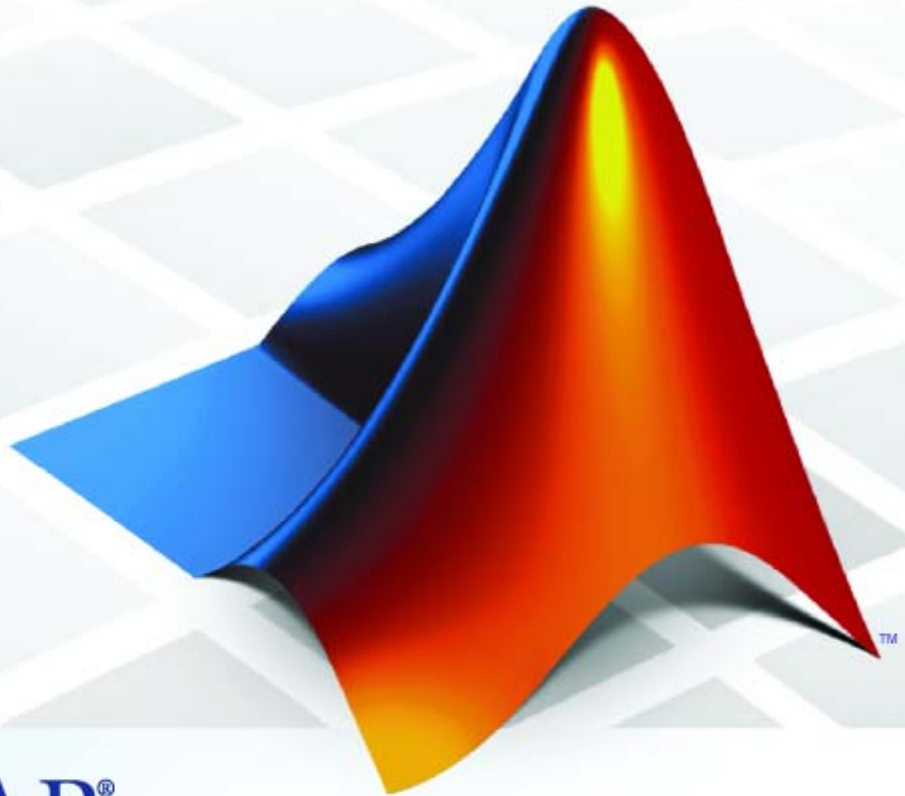
How to Contact The MathWorks

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 2004–2009 by The MathWorks, Inc.

Trademarks

Patents

Revision History

Contents

Working with fi Objects

3

Working with fimath Objects

4

Working with fipref Objects

5

Working with numerictype Objects

6

<div style="text-align:right">Working with quantizer Objects</div>

7

<div style="text-align:right">Working with the Fixed-Point Embedded
MATLAB Subset</div>

8

Interoperability with Other Products

9

Index

1

Getting Started

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 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 for binary point-only signals using the `divide` function or arithmetic operators / and ./

- 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 you set the fipref LoggingMode property to off. A Fixed-Point Toolbox™ license *is* checked out when you

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

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

Getting Help

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

Getting Help in This Document

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

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

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

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

Getting Help at the MATLAB Command Line

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

```
help objectname
```

For example,

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

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

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

For example,

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

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

```
doc fixedpoint/functionname
```

For example,

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

Display Settings

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

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

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

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

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

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

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

Displaying the fimath Properties of fi Objects

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

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

MATLAB returns the following:

```
a =

    3.1415

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

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

b =

    3.1416

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

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

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

Hiding the fimath Properties of fi Objects

If you are working with multiple fi objects that have their own attached fimath object, you may want to turn off the fimath object display:

- NumberDisplay — 'RealWorldValue'

- NumericTypeDisplay — 'full'

- FimathDisplay — 'none'

For example,

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

p =

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

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

a =
    3.1415

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

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

```
isfimathlocal(a)
```

```
ans =
     1
```

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

Shortening the numerictype Display of fi Objects

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

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

a = fi(pi)

a =
    3.1416
      s16,13
```

Demos

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

- Fixed-Point Basics — Demonstrates the basic use of the fixed-point `fi` object

- Number Circle — Illustrates the definitions of unsigned and signed two's complement integer and fixed-point numbers

- Binary Point Scaling — Explains binary point-only scaling

- Fixed-Point Data Type Override, Min/Max Logging, and Scaling — Steps through the workflow of using doubles override and min/max logging in the toolbox to choose appropriate scaling for a fixed-point algorithm

- Fixed-Point C Development — Shows how to use the parameters from a fixed-point MATLAB program in a fixed-point C program

- Fixed-Point Algorithm Development — Presents the development and verification of a simple fixed-point algorithm

- Fixed-Point Fast Fourier Transform (FFT) — Provides an example of converting a textbook Fast Fourier Transform algorithm into fixed-point MATLAB code and then into fixed-point C code

- Analysis of a Fixed-Point State-Space System with Limit Cycles — Demonstrates a limit cycle detection routine applied to a state-space system

- Quantization Error — Demonstrates the statistics of the error when signals are quantized using various rounding methods

- Fixed-Point Lowpass Filtering Using 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 ATAN2 Calculation — Uses the CORDIC algorithm and polynomial approximation to perform a fixed-point calculation of the four quadrant inverse tangent

2

Fixed-Point Concepts

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

Fixed-Point Data Types

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

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

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

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

b_{wl-1}	b_{wl-2}	...	b_5	b_4	b_3	b_2	b_1	b_0

MSB   LSB
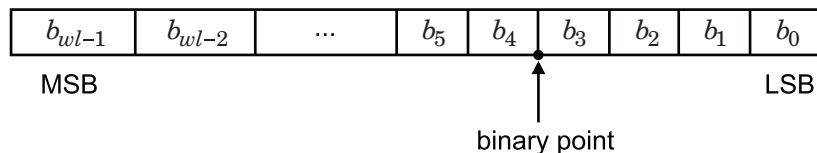
binary point

where

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

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

- Sign/magnitude

- One's complement

- Two's complement

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

Scaling

Fixed-point numbers can be encoded according to the scheme

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

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:

$$S \cdot \left(-2^{wl-1}\right) + B \qquad\qquad B \qquad\qquad S \cdot \left(2^{wl-1} - 1\right) + B$$

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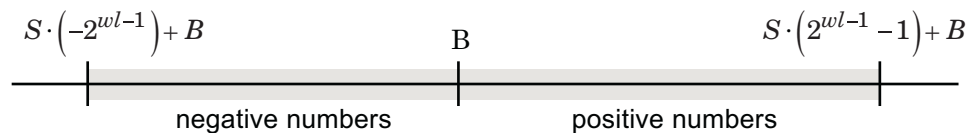
For both signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is 2^{wl}.

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

For slope = 1 and bias = 0:

$$-2^{wl-1} \qquad\qquad\qquad 0 \qquad\qquad\qquad 2^{wl-1} - 1$$

negative numbers   positive numbers
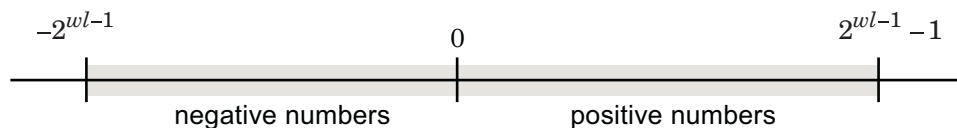
Overflow Handling

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

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

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

Precision

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

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

Rounding Methods

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

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

- `ceil` rounds to the closest representable number in the direction of positive infinity.

- `convergent` rounds to the closest representable number. In the case of a tie, `convergent` rounds to the nearest even number. This is the least biased rounding method provided by the toolbox.

- `fix` rounds to the closest representable number in the direction of zero.

- `floor`, which is equivalent to two's complement truncation, rounds to the closest representable number in the direction of negative infinity.

- `nearest` rounds to the closest representable number. In the case of a tie, `nearest` rounds to the closest representable number in the direction of positive infinity. This rounding method is the default for `fi` object creation and `fi` arithmetic.

- `round` rounds to the closest representable number. In the case of a tie, the `round` method rounds:

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

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

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

- Cost — Independent of the hardware being used, how much processing expense does the rounding method require?

  - Low — The method requires few processing cycles.

  - Moderate — The method requires a moderate number of processing cycles.

  - High — The method requires more processing cycles.

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

- Bias — What is the expected value of the rounded values minus the original values: $E(\hat{\theta} - \theta)$?

  - $E(\hat{\theta} - \theta) < 0$ — The rounding method introduces a negative bias.

  - $E(\hat{\theta} - \theta) = 0$ — The rounding method is unbiased.

  - $E(\hat{\theta} - \theta) > 0$ — The rounding method introduces a positive bias.

- Possibility of Overflow — Does the rounding method introduce the possibility of overflow?

  - Yes — The rounded values may exceed the minimum or maximum representable value.

  - No — The rounded values will never exceed the minimum or maximum representable value.

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

Fixed-Point Toolbox Rounding Method	Simulink Fixed Point Rounding Mode	Cost	Bias	Possibility of Overflow
ceil	Ceiling	Low	Large positive	Yes
convergent	Convergent	High	Unbiased	Yes
fix	Zero	Low	• Large positive for negative samples • Unbiased for samples with evenly distributed positive and negative values • Large negative for positive samples	No
floor	Floor	Low	Large negative	No
nearest	Nearest	Moderate	Small positive	Yes
round	Round	High	• Small negative for negative samples • Unbiased for samples with evenly distributed positive and negative values • Small positive for positive samples	Yes
N/A	Simplest (Simulink Fixed Point only)	Low	Depends on the operation	No

Arithmetic Operations

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

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

Modulo Arithmetic

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

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

9 ...   ... plus 9 more ...

... equals 6.
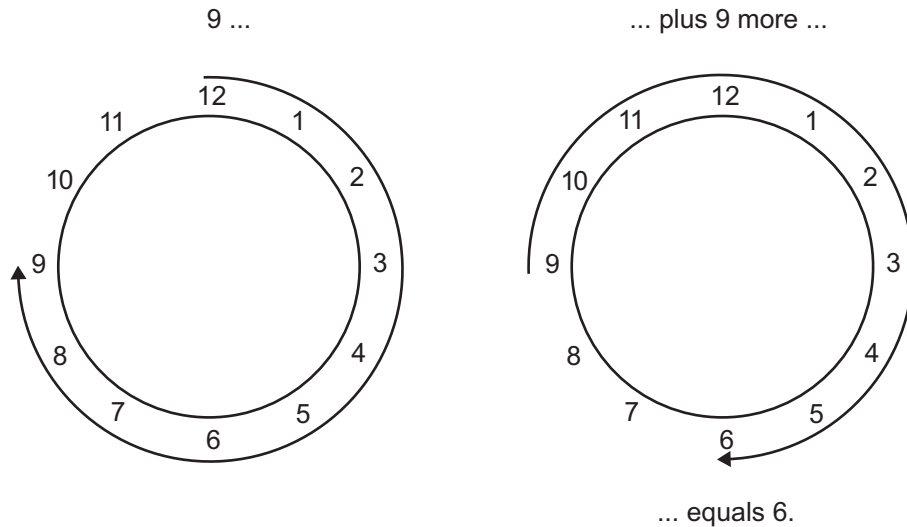
Similarly, binary math can only use the numbers 0 and 1, and any arithmetic results that fall outside this range are wrapped "around the circle" to either 0 or 1.

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:

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

Addition and Subtraction

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

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

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

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

$$
\begin{array}{ll}
010010.100 & (18.5) \\
-0110.110 & (6.75) \\
\hline
\end{array}
$$

two's complement and sign extension $\longrightarrow$

$$
\begin{array}{ll}
010010.100 & (18.5) \\
+111001.010 & (-6.75) \\
\hline
1001011.110 & (11.75)
\end{array}
$$

Carry bit is discarded

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

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

Multiplication

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

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

Multiplication Data Types

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

Real-Real Multiplication. The following diagram shows the data types used by the toolbox in the multiplication of two real numbers. The software returns the output of this operation in the product data type, which is governed by the `fimath` object `ProductMode` property.
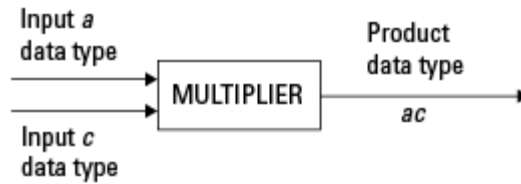
Real-Complex Multiplication. The following diagram shows the data types used by the toolbox in the multiplication of a real and a complex fixed-point number. Real-complex and complex-real multiplication are equivalent. The software returns the output of this operation in the product data type, which is governed by the `fimath` object `ProductMode` property:
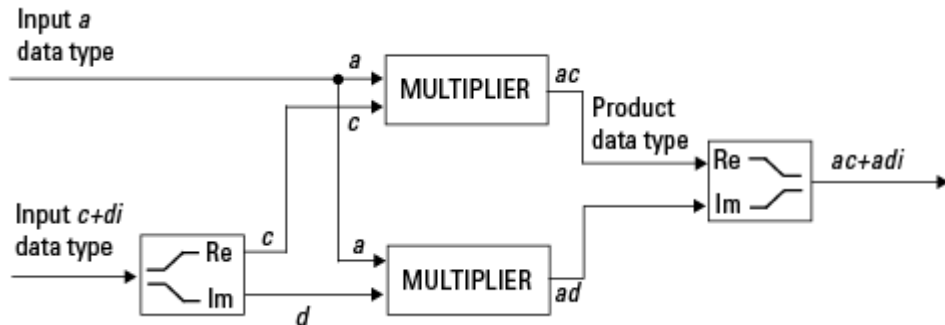
Complex-Complex Multiplication. The following diagram shows the multiplication of two complex fixed-point numbers. Note that the software returns the output of this operation in the sum data type, which is governed by the `fimath` object `SumMode` property. The intermediate product data type is determined by the `fimath` object `ProductMode` property.
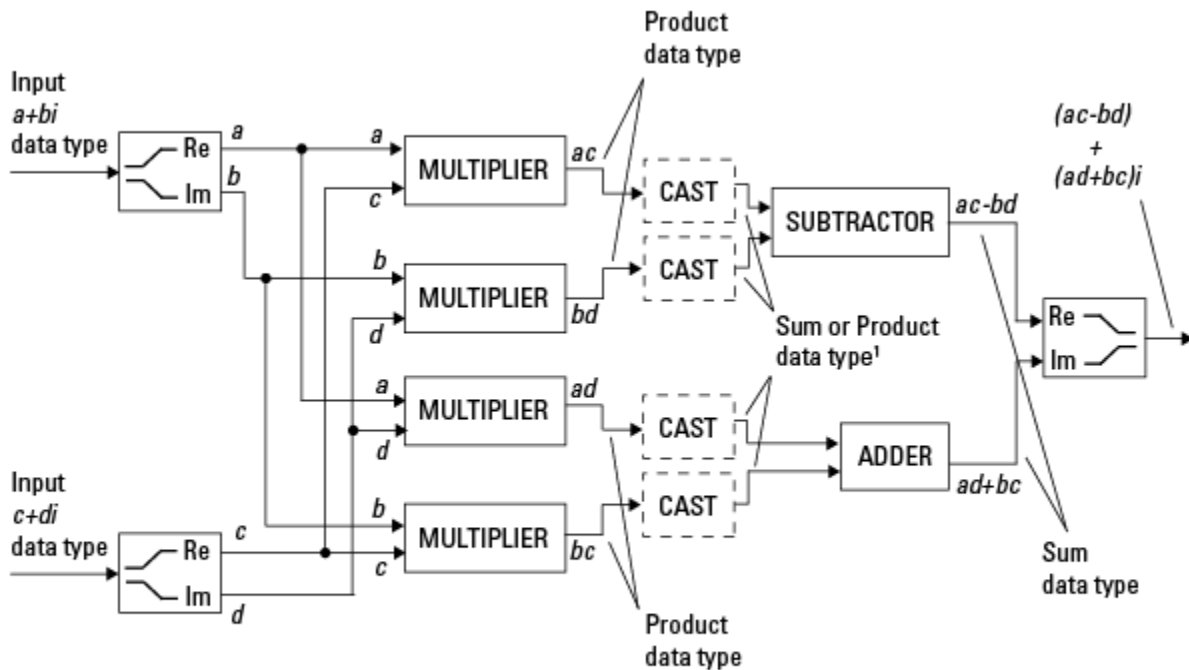
¹ Sum data type if CastBeforeSum is true,
Product data type if CastBeforeSum is false

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

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

for the subtractor, and

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

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

Multiplication with fimath

In the following examples, let

- F = fimath('ProductMode','FullPrecision',...

  'SumMode','FullPrecision')

- T1 = numerictype('WordLength',24,'FractionLength',20)

- T2 = numerictype('WordLength',16,'FractionLength',10)

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

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

x =

     5

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

y = fi(10, T2, F)

y =

    10

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

```
z = x*y

z =

    50


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

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

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

x =

     5


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

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

y =

  10.0000 + 2.0000i


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

```
           FractionLength: 10

z = x*y

z =

  50.0000 +10.0000i


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

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

```
x = fi(5+6i,T1,F)

x =

   5.0000 + 6.0000i


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

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

y =

  10.0000 + 2.0000i


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

```
        FractionLength: 10

z = x*y

z =

  38.0000 +70.0000i


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

Casts

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

Note For more examples of casting, see "Casting fi Objects" on page 3-13.

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:

The source bits must be shifted up to match the
binary point position of the destination data type.

source

destination

This bit from the source data
type "falls off" the high end with
the shift up. Overflow might occur.
The result will saturate or wrap.

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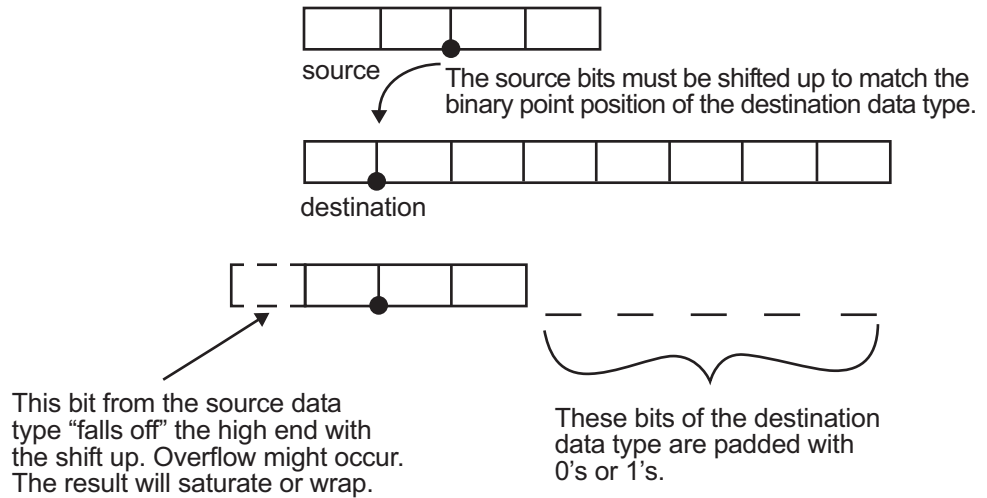
As the diagram shows, the source bits are shifted up so that the binary point
matches the destination binary point position. The highest source bit does
not fit, so overflow might occur and the result can saturate or wrap. The
empty bits at the low end of the destination data type are padded with either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

source

The source bits must be shifted down to match the binary point position of the destination data type.

destination

There is no value for this bit from the source, so the result must be sign-extended to fill the destination data type.

These bits from the source do not fit into the destination data type. The result is rounded.
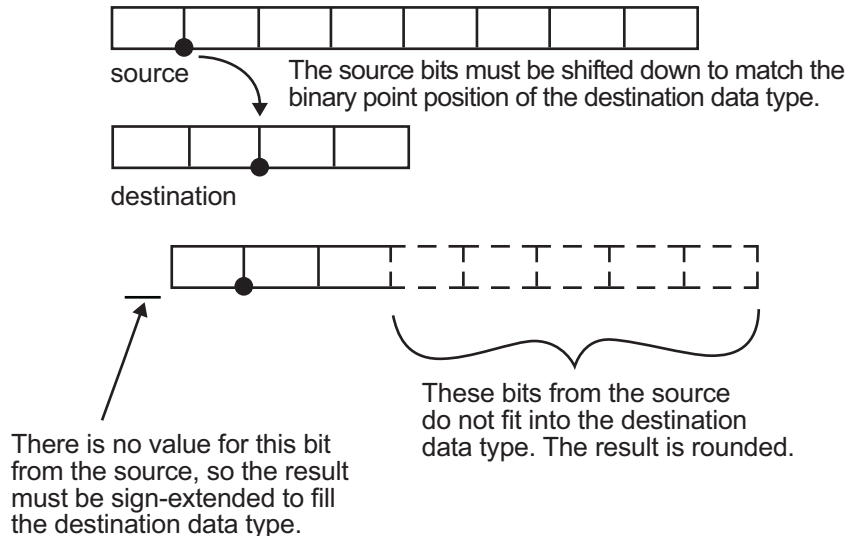
As the diagram shows, the source bits are shifted down so that the binary point matches the destination binary point position. There is no value for the highest bit from the source, so the result is sign-extended to fill the integer portion of the destination data type. The bottom five bits of the source 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

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
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

Constructor	Signed	Word Length	Fraction Length	Minimum	Maximum	Closest ANSI C Equivalent
`fi(x,1,16,0)`	Yes	16	0	−32,768	32,767	`short int`
`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. Round mode is always floor. Overflow mode is obeyed. 0-valued bits are shifted in on the right.
x>>n	bitshift(x,-n)	Result is same numeric type as operand. Round mode is always floor. Overflow mode is obeyed. 0-valued bits are shifted in on the left if the operand is unsigned or signed and positive. 1-valued bits are shifted in on the left if the operand is signed and negative.
+x	+x	Result is same numeric type as operand.
-x	-x	Result is same numeric type as operand. Overflow mode is obeyed. For example, overflow might occur when you negate an unsigned fi or the most negative value of a signed fi.

Binary Conversions

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

ANSI C Usual Binary Conversions

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

Type of One Operand	Type of Other Operand	ANSI C Conversion
long double	Any	long double

Type of One Operand	Type of Other Operand	ANSI C Conversion
`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
fi	int64	• Signed = 1 • WordLength = 64 • FractionLength = 0
fi	uint64	• Signed = 0 • WordLength = 64 • FractionLength = 0

Overflow Handling

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

ANSI C Overflow Handling

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

The results of unsigned integer overflows wrap in ANSI C.

fi Overflow Handling

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

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

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

fimath Object Properties Related to Overflow Handling	Property Value	Description
ProductMode	'FullPrecision'	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.
	'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.

fimath Object Properties Related to Overflow Handling	Property Value	Description
	`'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.
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'`.

fimath Object Properties Related to Overflow Handling	Property Value	Description
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.
	'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.

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

3

Working with fi Objects

Constructing fi Objects

fi Object Syntaxes

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

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

- You can use the `sfi` constructor function to create a new signed `fi` object.

- You can use the `ufi` constructor function to create a new unsigned `fi` object.

- You can use any of the `fi` constructor functions to copy an existing `fi` object.

To get started, type

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

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

```
a =

    0


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

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

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

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

Examples of Constructing fi Objects

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

- `fi`

- `sfi`

- `ufi`

Constructing a fi Object with Property Name/Property Value Pairs

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

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

a =

    3.1415

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

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

```
        MaxSumWordLength: 128
          CastBeforeSum: true
```

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

- If you specify at least one `fimath` object property in the `fi` constructor, the `fi` object gets assigned its own `fimath` object. The `fi` object inherits the remaining unspecified `fimath` object properties from the global fimath.

- If you do not specify any `fimath` object properties in the `fi` object constructor, the `fi` object associates itself with the global fimath. See "Working with the Global fimath" on page 4-6 for more information.

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
          Signedness: Signed
          WordLength: 16
       FractionLength: 15

a = fi(pi, T)

a =

    1.0000


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

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

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

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
              Signedness: Signed
              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. When you do so, the
`fimath` object you specify gets attached to the `fi` object you create. If you do
not specify any `numerictype` object properties, the word length of the `fi` object
defaults to 16 bits. The fraction length is determined by best precision scaling:

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

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
                  Signedness: Signed
                  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
                  Signedness: Unsigned
                  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
                 Signedness: Unsigned
                 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
                 Signedness: Unsigned
                 WordLength: 8
             FractionLength: 6

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

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

Building fi Object Constructors in a GUI

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

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

1 Open the **Insert fi Constructor** dialog box by selecting
  Tools > Fixed-Point Toolbox > Insert fi Constructor from the editor
  menu.

2 Use the edit boxes and drop-down menus to specify the following properties
  of the fi object:

  - **Value** = pi
  - **Data type mode** = Fixed-point: binary point scaling
  - **Signedness** = Signed
  - **Word length** = 16
  - **Fraction length** = 13
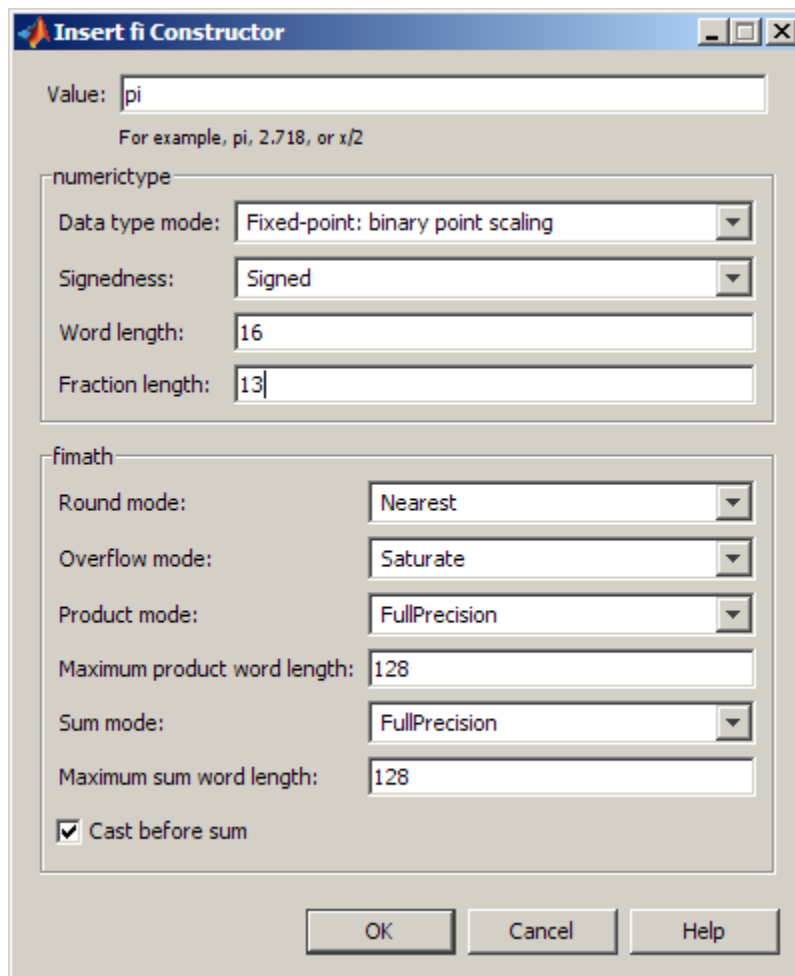
3 To insert the fi object constructor string into your M-file, place your cursor at the desired location in the M-file, and click **OK** on the **Insert fi Constructor** dialog box. Clicking **OK** closes the **Insert fi Constructor** dialog box and automatically populates the fi object constructor string in your M-file:

```
 8        a = fi(pi, 1, 16, 13, ...
 9             'RoundMode', 'Nearest', ...
10             'OverflowMode', 'Saturate', ...
11             'ProductMode', 'FullPrecision', ...
12             'MaxProductWordLength', 128, ...
13             'SumMode', 'FullPrecision', ...
14             'MaxSumWordLength', 128, ...
15             '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
         Signedness: Signed
         WordLength: 16
      FractionLength: 14
```

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

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

a =

    1.9999

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

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

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

b =

    3.1416

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

Copying a fi Object

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

```
a = fi(pi)

a =

    3.1416

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

b =

    3.1416

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

Casting fi Objects

Overwriting by Assignment

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

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

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

A =

  32767

class(A)

ans =

int16
```

Ways to Cast with MATLAB Software

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

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

Casting by Subscripted Assignment

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

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

A =

  127

class(A)

ans =

int8
```

The same is true for `fi` objects:

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

A =

  127
      s8,0
```

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

Casting by Conversion Function

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

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

A =

   127

class(A)

ans =

int8
```

The same is true for fi objects:

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

B =

      32767
     s16,0

A =

    127
     s8,0
```

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

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

T8 =
```

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

T16 = numerictype(1,16,0)

T16 =

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

B = fi(32767,T16)

B =

        32767
      s16,0

A = fi(B, T8)

A =

    127
      s8,0
```

Casting with the reinterpretcast Function

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

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

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

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

B =

    0.7813    1.0000    1.5625    4.0000

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

A =

   12.5000   16.0000   25.0000   -64.0000

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

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

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

binary_A =

00011001   00100000   00110010   10000000

binary_B =

00011001   00100000   00110010   10000000
```

fi Object Properties

Data Properties

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

- `bin` — Stored integer value of a `fi` object in binary

- `data` — Numerical real-world value of a `fi` object

- `dec` — Stored integer value of a `fi` object in decimal

- `double` — Real-world value of a `fi` object, stored as a MATLAB `double` data type

- `hex` — Stored integer value of a `fi` object in hexadecimal

- `int` — Stored integer value of a `fi` object, stored in a built-in MATLAB integer data type. You can also use `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, and `uint64` to get the stored integer value of a `fi` object in these formats

- `oct` — Stored integer value of a `fi` object in octal

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

fimath Properties

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

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

- When you do not specify any `fimath` object properties in the `fi` constructor, the resulting `fi` object associates itself with the global fimath. All `fi` objects created in this way do not have their own attached `fimath` objects.

To determine whether a `fi` object has an attached `fimath` object, use the `isfimathlocal` function.

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

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

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

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

numerictype Properties

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

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

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

- `Bias` — Bias of a `fi` object

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

- `DataTypeMode` — Data type and scaling mode of a `fi` object

- `FixedExponent` — Fixed-point exponent associated with a `fi` object

- FractionLength — Fraction length of the stored integer value of a fi object in bits

- Scaling — Fixed-point scaling mode of a fi object

- Signed — Whether a fi object is signed or unsigned

- Signedness — Whether a fi object is signed or unsigned

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

- Slope — Slope associated with a fi object

- SlopeAdjustmentFactor — Slope adjustment associated with a fi object

- WordLength — Word length of the stored integer value of a fi object in bits

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

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

- "Setting Fixed-Point Properties at Object Creation" on page 3-21

- "Using Direct Property Referencing with fi" on page 3-22

Setting fi Object Properties

You can set fi object properties in two ways:

- Setting the properties when you create the object

- Using direct property referencing

Setting Fixed-Point Properties at Object Creation

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

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

a =

    3.1416


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

              RoundMode: convergent
           OverflowMode: wrap
            ProductMode: FullPrecision
    MaxProductWordLength: 128
                SumMode: FullPrecision
        MaxSumWordLength: 128
          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
             Signedness: Signed
             WordLength: 16
          FractionLength: 13

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

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

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

  b =
      3.1415

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

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

  b.fimath = []
```

```
b =
    3.1415

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

isfimathlocal(b)

ans =
     0
```

fi Object Functions

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

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

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

For example,

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

n =

  25736

a.int

ans =

  25736

h = hex(a)

h =

6488
```

```
a.hex

ans =

6488
```

4

Working with fimath Objects

Constructing fimath Objects

fimath Object Syntaxes

When attached to a fi object, fimath objects define the arithmetic attributes of that fi object. If a fi object does not have an attached fimath object, the global fimath determines the arithmetic attributes of the fi object.

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

- You can use the fimath constructor function to create new fimath objects.

- You can use the fimath constructor function to copy an existing fimath object.

To get started, type

```
F = fimath
```

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

```
F =


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

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

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

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

ans =

     1
```

The syntax

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

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

Building fimath Object Constructors in a GUI

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

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

1 Open the **Insert fimath Constructor** dialog box by selecting **Tools > Fixed-Point Toolbox > Insert fimath Constructor** from the editor menu.

2 Use the edit boxes and drop-down menus to specify the following properties of the `fimath` object:

- **Round mode** = Floor
- **Overflow mode** = Wrap
- **Product mode** = FullPrecision
- **Maximum product word length** = 128
- **Sum mode** = FullPrecision
- **Maximum sum word length** = 128
- **Cast before sum** = Checked
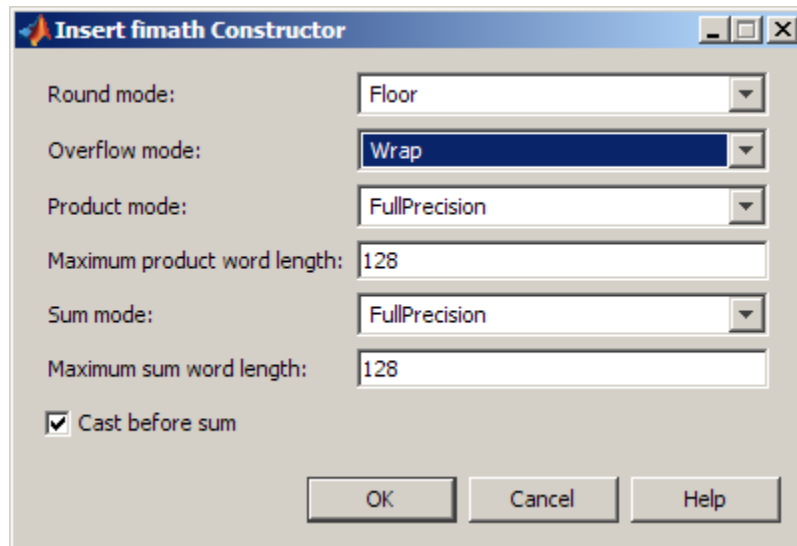
3 To insert the `fimath` object constructor string into your M-file, place your cursor at the desired location in the M-file. Then click **OK** on the **Insert fimath Constructor** dialog box. Clicking **OK** closes the **Insert fimath Constructor** dialog box and automatically populates the `fimath` object constructor string in your M-file:

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

Working with the Global fimath

The *global fimath* is the set of fimath properties that fi constructors associate with, and fimath constructors return, when you do not specify any fimath properties in the constructor. If you do not specify any fimath properties in a fi object constructor, the resulting fi object associates itself with the global fimath. You can partially specify a fimath object in a fi or fimath constructor using property name/property value pairs. Doing so allows the remaining unspecified fimath properties to inherit their values from the global fimath. The factory-default setting of the global fimath has the following properties:

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

You can use dot notation to change any fimath property of a fi object after creating the fi object. You can also use dot notation to associate an existing fi object with the global fimath. For example, the syntax a.fimath = [] removes the attached fimath object from a and associates a with the global fimath.

You can configure the global fimath using the Fixed-Point Toolbox setdefaultfimath, savedefaultfimathpref, removedefaultfimathpref, and resetdefaultfimath functions. The global fimath allows you to define your own set of fimath properties for all fi and fimath objects to use as a default. The following sections show you how to:

- Create a fimath object
- Set your fimath object as the global fimath for your current MATLAB session
- Save your fimath object as the global fimath for all future MATLAB sessions
- Reset the global fimath to the factory default for your current MATLAB session

• Remove a user-configured global fimath from your MATLAB preferences
  and use the factory-default global fimath for all future MATLAB sessions

See the following section for information on how a `fi` object with an attached
`fimath` object interacts with a `fi` object associated with the global fimath:

Configuring the Global fimath

You can configure the global fimath in two ways:

• By using property name/property value pairs inside the `setdefaultfimath`
  function. Any property that you fail to specify a value for defaults to the
  value of the current global fimath.

```
setdefaultfimath('RoundMode', 'Floor', 'OverflowMode', 'Wrap');
```

• By selecting a variable name and using property name/property value pairs
  syntax to create a `fimath` object with the `properties` you want. Then,
  use the `setdefaultfimath` command to set the `fimath` object you created
  as the global fimath. For example:

```
Fdefault = fimath('RoundMode','Floor','OverflowMode','Wrap');
setdefaultfimath(Fdefault);
```

From this point forward (in your current MATLAB session), `fi` and `fimath`
objects use the global fimath you configured. All `fi` objects you create without
specifying any `fimath` properties in the constructor associate themselves with
the global fimath. You can determine if a `fi` object is associated with the
global fimath by using the `isfimathlocal` function:

```
a = fi(pi)

a =
    3.1416

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

```
isfimathlocal(a)

ans =
     0

F1 = fimath

F1 =


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

Saving the Global fimath

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

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

```
savedefaultfimathpref;
```

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

Resetting the Global fimath to the Factory Default

You can reset the global fimath to the factory setting at any time by using the resetdefaultfimath function. To do so, type

```
resetdefaultfimath;
```

at the MATLAB command line.

When you reset the global fimath, all `fi` objects that are associated with it become associated with the factory-default fimath properties. Unless you choose to configure the global fimath again, MATLAB uses the factory-default setting of the global fimath for the remainder of your MATLAB session.

```
b = fi(pi)

b =

    3.1416

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

b.fimath

ans =

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

F1 = fimath

F1 =

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

4-9

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

```
removedefaultfimathpref;
```

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

Using the Global fimath with the Embedded MATLAB Function Block

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

"Using fimath Objects in Embedded MATLAB Function Blocks" on page 8-19

fimath Object Properties

Math, Rounding, and Overflow Properties

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

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

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

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

Setting fimath Object Properties

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

Setting fimath Properties at Object Creation

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

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

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

F =
```

```
          RoundMode: convergent
       OverflowMode: saturate
        ProductMode: FullPrecision
MaxProductWordLength: 128
            SumMode: FullPrecision
    MaxSumWordLength: 128
      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 so by using a period to index into a property by name.

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

```
F.RoundMode

ans =

convergent
```

To set the `OverflowMode` of `F`,

```
F.OverflowMode = 'wrap'

F =


          RoundMode: convergent
       OverflowMode: wrap
        ProductMode: FullPrecision
MaxProductWordLength: 128
            SumMode: FullPrecision
    MaxSumWordLength: 128
      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 following figure shows the Model Explorer when you define the following fimath objects in the MATLAB workspace:

```
F = fimath

F =


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

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

G =


              RoundMode: nearest
           OverflowMode: wrap
            ProductMode: FullPrecision
   MaxProductWordLength: 128
                SumMode: FullPrecision
       MaxSumWordLength: 128
           CastBeforeSum: true
```
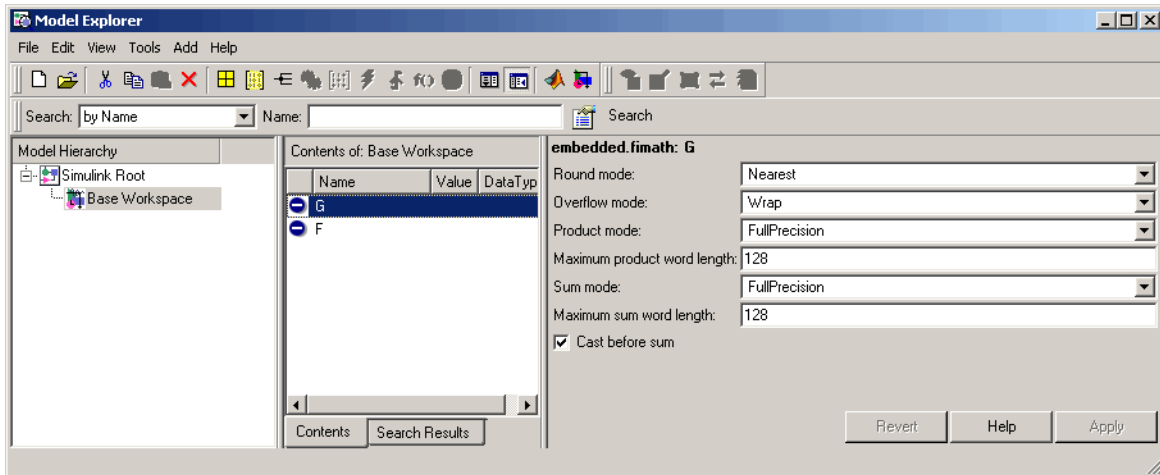
Select the **Base Workspace** node in the **Model Hierarchy** pane to view the current objects in the **Contents** pane. When you select a `fimath` object in the **Contents** pane, you can view and change its properties in the **Dialog** pane.

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

- "Specifying Fixed-Point Parameters in the Model Explorer" on page 8-17
- "Sharing Models with Fixed-Point Embedded MATLAB Function Blocks" on page 8-22

Using fimath Properties to Perform Fixed-Point Arithmetic

In this section...

fimath Rules for Fixed-Point Arithmetic

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

- Individual fi objects can have a fimath object attached to them. fi objects have an attached fimath object when you specify fimath properties in a fi constructor or use dot notation to set a fimath object property of a fi object.

- The global fimath is the set of fimath properties that fi constructors associate with, and fimath constructors return, when you do not specify any fimath properties in the constructor. When you use the sfi or ufi constructor or when you fail to specify any fimath properties in the fi constructor, the resulting fi object associates itself with the global fimath. It does not have its own attached fimath object. See "Working with the Global fimath" on page 4-6 for more information.

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

The following sections discuss the rules for when a fi object with an attached fimath object interacts with a fi object associated with the global fimath.

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

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

- If either a or b has an attached fimath object, the operation uses that attached fimath object to perform the fixed-point arithmetic. The fimath object used for the operation gets attached to the output fi object c.

Unary Operations

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

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

- If a has an attached fimath object, the operation uses that fimath object to perform the fixed-point arithmetic. The fimath object used for the operation attaches to the output fi object b.

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

Concatenation Operations

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

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

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

- If a is a fi object associated with the global fimath, the output fi object d also associates with the global fimath.

- If a has its own fimath object, that fimath object also attaches to the output fi object d.

- If a is not a fi object, the output fi object d inherits the fimath properties of the next left-most fi object. For example, if b is a fi object with an

4-17

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

fimath Object Operations: add, mpy, sub

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

Embedded MATLAB Subset Operations

Fixed-point operations performed with Embedded MATLAB subset functions use the same rules as fixed-point operations performed in MATLAB.

All input signals to the Embedded MATLAB Function block that you treat as `fi` objects associate with whatever you specify for the **Embedded MATLAB Function block fimath** parameter. When you set this parameter to `Same as MATLAB`, your `fi` objects associate with the global fimath. When you set the **Embedded MATLAB Function block fimath** parameter to `Specify other`, you can define your own set of `fimath` properties for all `fi` objects in the Embedded MATLAB Function block to associate with. You can choose to treat only fixed-point input signals as `fi` objects or both fixed-point and integer input signals as `fi` objects. See "Using fimath Objects in Embedded MATLAB Function Blocks" on page 8-19 for more information.

Binary-Point Arithmetic

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

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

```
a = fi(pi)

a =
    3.1416
```

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

a.fimath
isfimathlocal(a)

ans =


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


ans =
     0
```

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

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

ans =

     1

a + b

ans =
```

```
      11.1416

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

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

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

a =

    3

          DataTypeMode: double

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

b =

   27

          DataTypeMode: double

a + b

ans =

   30

          DataTypeMode: double

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

c =
```

```
           12

                DataTypeMode: single

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

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

```
  a = fi(pi)

  a =

      3.1416

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

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

  b =

       1     3
       4     2

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

  a + b
```

```
ans =

    4.1416    6.1416
    7.1416    5.1416


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

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

[Slope Bias] Arithmetic

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

- [Slope Bias] signals must be real.

- You must set the `SumMode` and `ProductMode` properties of the governing `fimath` to `'SpecifyPrecision'` for sum and multiply operations, respectively.

- You must set the `CastBeforeSum` property of the governing `fimath` to `'true'`.

- Fixed-Point Toolbox does not support the `divide` function for [Slope Bias] signals.

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

f =


            RoundMode: nearest
         OverflowMode: saturate
```

```
                                  ProductMode: FullPrecision
                          MaxProductWordLength: 128
                                      SumMode: SpecifyPrecision
                                SumWordLength: 32
                            SumFractionLength: 16
                                 CastBeforeSum: true

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

a =

    3.1416

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

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

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

b =

    22

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

                                RoundMode: nearest
```

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


    a + b

    ans =

       25.1416

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

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

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

Using fimath to Specify Rounding and Overflow Modes

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

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

a =

    -1

b = fi(1,T)

b =

    0.9922
```

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

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

```
b.fimath = F

b =

    0.9922
```

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

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

Using fimath to Share Arithmetic Rules

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

- "Using the Global fimath to Share Arithmetic Rules " on page 4-27
- "Using fimath Objects to Share Arithmetic Rules" on page 4-28

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

Using the Global fimath to Share Arithmetic Rules

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

- Create a `fi` object using the `fi` constructor, and do not specify any `fimath` properties in the constructor call. For example:

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

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

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

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

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

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

Using fimath Objects to Share Arithmetic Rules

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

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

T =

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

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

F =

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

a = fi(pi, T, F)

a =

    -0.8584
```

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

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


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

b =

    1.5708


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

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

Using fimath ProductMode and SumMode

In this section...
"Example Setup" on page 4-30
"FullPrecision" on page 4-31
"KeepLSB" on page 4-32
"KeepMSB" on page 4-33
"SpecifyPrecision" on page 4-34

Example Setup

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

- FullPrecision

- KeepLSB

- KeepMSB

- SpecifyPrecision

To follow along, first set the following preferences:

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

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

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

```
        3.1563
          s8,5


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

FullPrecision

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

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

  b
  b =
      2.7188    %010.10111
        s8,5

  a*b
  ans =
      8.5811    %001000.1001010011
        s16,10

  a+b
  ans =
      5.8750    %0101.11100
        s9,5
```

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

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

KeepLSB

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

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

b
b =
    2.7188    %010.10111
      s8,5

a*b
ans =
    0.5811    %00.1001010011
      s12,10

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

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

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

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

KeepMSB

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

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

b
b =
    2.7188    %010.10111
      s8,5

a*b
ans =
    8.5781    %001000.100101
      s12,6

a+b
ans =
```

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

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

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

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

SpecifyPrecision

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

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

b
b =
    2.7188    %010.10111
```

```
      s8,5

  a*b
  ans =
      0.5781    %0.1001010
        s8,7

  a+b
  ans =
     -0.1250    %1.1110000
        s8,7
```

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

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

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

fimath Object Functions

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

5

Working with 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

Display, Data Type Override, and Logging Properties

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

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

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

Setting fipref Object Properties

Setting fipref Properties at Object Creation

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

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

P =

        NumberDisplay: 'bin'
    NumericTypeDisplay: 'short'
```

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

Using Direct Property Referencing with fipref

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

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

```
P.NumberDisplay

ans =

bin
```

To set the `NumericTypeDisplay` of `P`,

```
P.NumericTypeDisplay = 'full'

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

Using fipref Objects to Set Display Preferences

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

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

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

a =
    3.1415

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

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

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

```
a = fi(pi)

a =

    3.1416

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

```
          FractionLength: 13
```

Next, change the `fipref` display properties:

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

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

a =
0110010010000111
        s16,13
```

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

Using fipref Objects to Set Logging Preferences

In this section...

Logging Overflows and Underflows as Warnings

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 =

      1.0000

minlog(x)

ans =
    -1

noverflows(x)

ans =

         2

nunderflows(x)

ans =

         1
```

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

```
resetlog(x)
maxlog(x)

ans =

     []

minlog(x)

ans =

     []

noverflows(x)

ans =

     []

nunderflows(x)

ans =

     []
```

Using fipref Objects to Set Data Type Override Preferences

In this section...

Overriding the Data Type of fi Objects

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

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

p =

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

a = fi(pi)

a =

    3.1416

          DataTypeMode: double

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

p =
```

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

  a

  a =

      3.1416

              DataTypeMode: double

  b = fi(pi)

  b =

      3.1416

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

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

Using Data Type Override to Help Set Fixed-Point Scaling

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

1 Implement your algorithm using fixed-point `fi` objects, using initial "best guesses" for word lengths and scalings.

2 Set the `fipref DataTypeOverride` property to `ScaledDoubles`, `TrueSingles`, or `TrueDoubles`.

3 Set the `fipref LoggingMode` property to `on`.

4 Use the `maxlog` and `minlog` functions to log the maximum and minimum values achieved by the variables in your algorithm in floating-point mode.

5 Set the `fipref DataTypeOverride` property to `ForceOff`.

6 Use the information obtained in step 4 to set the fixed-point scaling for each variable in your algorithm such that the full numerical range of each variable is representable by its data type and scaling.

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

fipref Object Functions

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

6

Working with numerictype Objects

Constructing numerictype Objects

In this section...

---

numerictype Object Syntaxes

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

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

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

To get started, type

```
T = numerictype
```

to create a default numerictype object.

```
T =


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

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

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

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

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

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

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

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

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

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

T1 =

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

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

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

```
T2 = numerictype(0,32)

T2 =

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

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

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

Example: Copying a numerictype Object

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

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

ans =

     1
```

Example: Building numerictype Object Constructors in a GUI

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

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

1 Open the **Insert numerictype Constructor** dialog box by selecting **Tools > Fixed-Point Toolbox > Insert numerictype Constructor** from the editor menu.

2 Use the edit boxes and drop-down menus to specify the following properties of the numerictype object:

- **Data type mode** = Fixed-point: binary point scaling

- **Signedness** = Signed

- **Word length** = 32

- **Fraction length** = 30
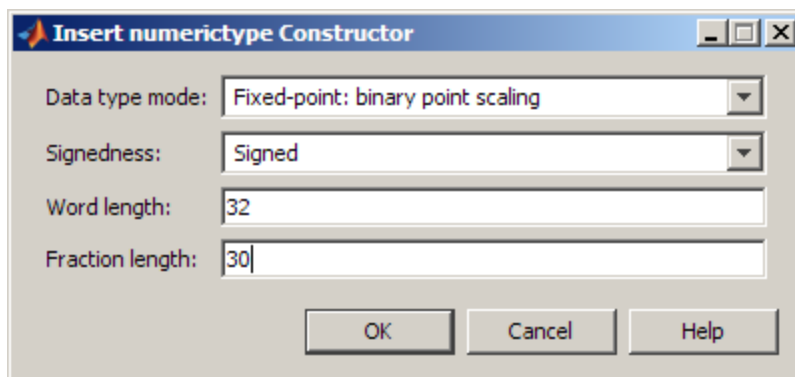
3 To insert the numerictype object constructor string into your M-file, place your cursor at the desired location in the M-file, and click **OK** on the **Insert**

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

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

numerictype Object Properties

Data Type and Scaling Properties

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

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

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

Setting numerictype Object Properties

Setting numerictype Properties at Object Creation

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

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

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

T =


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

Using Direct Property Referencing with numerictype Objects

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

For example, to get the word length of T,

```
T.WordLength

ans =

32
```

To set the fraction length of T,

```
T.FractionLength = 31

T =
```

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

Setting numerictype Properties in the Model Explorer

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

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

```
T = numerictype

T =


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

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

U =


          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 2^-15
                  Bias: 0
```
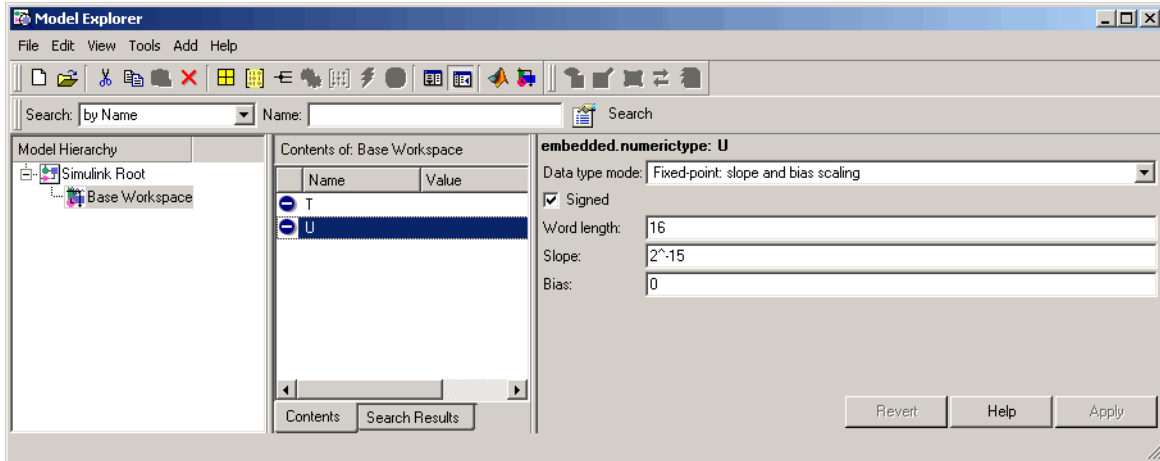
Select the **Base Workspace** node in the **Model Hierarchy** pane to view the current objects in the **Contents** pane. When you select a numerictype object in the **Contents** pane, you can view and change its properties in the **Dialog** pane.

The numerictype Structure

Valid Values for numerictype Structure Properties

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

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

DataTypeMode	DataType	Scaling	Signedness	Word-Length	Fraction-Length	Slope	Bias
Fixed-point data types							
Fixed-point: binary point scaling	Fixed	BinaryPoint	Signed Unsigned Auto	Positive integer from 1 to 65,536	Positive or negative integer	1	0
Fixed-point: slope and bias scaling	Fixed	SlopeBias	Signed Unsigned Auto	Positive integer from 1 to 65,536	N/A	Any floating-point number	Any floating-point number

DataTypeMode	DataType	Scaling	Signedness	Word-Length	Fraction-Length	Slope	Bias
Fixed-point: unspecified scaling	Fixed	Unspecified	Signed Unsigned Auto	Positive integer from 1 to 65,536	N/A	N/A	N/A
Scaled double data types							
Scaled double: binary point scaling	ScaledDouble	BinaryPoint	Signed Unsigned Auto	Positive integer from 1 to 65,536	Positive or negative integer	1	0
Scaled double: slope and bias scaling	ScaledDouble	SlopeBias	Signed Unsigned Auto	Positive integer from 1 to 65,536	N/A	Any floating-point number	Any floating-point number
Scaled double: unspecified scaling	ScaledDouble	Unspecified	Signed Unsigned Auto	Positive integer from 1 to 65,536	N/A	N/A	N/A
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 of fi objects

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

Example 1

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

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

T =


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

a = fi(pi,T)

a =


        3.1415926553309


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

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

```
b =

          1.5707963258028


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

Example 2

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

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

T =

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

c = fi(pi, T)

c =

      4

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

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

```
d =

    0

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

numerictype Object Functions

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

Working with quantizer Objects

Constructing quantizer Objects

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 the 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.

The Embedded MATLAB subset supports variable-size arrays and matrices with known upper bounds. To learn more about using variable-size signals in the Embedded MATLAB subset, see "Generating Code for Variable-Size Data" in 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 greater than 128 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.

- The `false` value of the `CastBeforeSum` property is not supported.

- The `numel` function returns the number of elements of `fi` objects in the Embedded MATLAB subset.

- When you compile code containing `fi` objects with non-trivial slope and bias scaling using the Embedded MATLAB subset, you may see different results than you achieve by running the same code in MATLAB.

To learn about the general limitations of the Embedded MATLAB subset that also apply to use with Fixed-Point Toolbox software, refer to "Overview of the Embedded MATLAB Subset" in the Embedded MATLAB subset documentation.

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

Function	Remarks/Limitations
abs	—
add	—
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	—

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

Function	Remarks/Limitations
bitsrl	—
bitxor	• Not supported for slope-bias scaled fi objects.
bitxorreduce	—
ceil	—
complex	—
conj	—

conv	• Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
	
	• For variable-sized signals, you may see different results between the Embedded MATLAB subset and MATLAB.
	
	  - In the Embedded MATLAB subset, the output for variable-sized signals is always computed using the SumMode property of the governing fimath.
	
	  - In MATLAB, the output for variable-sized signals is computed using the SumMode property of the governing fimath when both inputs are non-scalar. However, if either input is a scalar, MATLAB computes the output using the ProductMode of the governing fimath.

convergent	—
ctranspose	—
diag	• If supplied, the index, k, must be a real and scalar integer value that is not a fi object.
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	—

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

Function	Remarks/Limitations
`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.
`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.

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

Function	Remarks/Limitations
`getlsb`	—
`getmsb`	—
`gt`	• Not supported for fixed-point signals with different biases.
`horzcat`	—
`imag`	—
`int`, `int8`, `int16`, `int32`	—
`iscolumn`	—
`isempty`	—
`isequal`	—
`isfi`	—
`isfimath`	—
`isfimathlocal`	—
`isfinite`	—
`isinf`	—
`isnan`	—
`isnumeric`	—
`isnumerictype`	—
`isreal`	—
`isrow`	—
`isscalar`	—
`issigned`	—
`isvector`	—
`le`	• Not supported for fixed-point signals with different biases.
`length`	—
`logical`	—

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

Function	Remarks/Limitations
`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`	—
`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.
`mpy`	• When you provide complex inputs to the `mpy` function inside of an Embedded MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the **Ports and data manager** and set the **Complexity** parameter for all known complex inputs to `On`.
`mrdivide`	—
`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. • Variable-sized inputs are only supported when the `SumMode` property of the governing `fimath` is set to `Specify precision` or `Keep LSB`. • For variable-sized signals, you may see different results between the Embedded MATLAB subset and MATLAB. ▪ In the Embedded MATLAB subset, the output for variable-sized signals is always computed using the `SumMode` property of the governing `fimath`. ▪ In MATLAB, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath` when both inputs are non-scalar. However, if either input is a scalar, MATLAB computes the output using the `ProductMode` of the governing `fimath`.

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

Function	Remarks/Limitations
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	—
rdivide	—
real	—
realmax	—
realmin	—
reinterpretcast	—
repmat	—
rescale	—
reshape	—
round	—
sfi	—
sign	—

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

Function	Remarks/Limitations
single	—
size	—
sort	—
sqrt	• Complex and [Slope Bias] inputs error out. • Negative inputs yield a 0 result.
sub	—
subsasgn	—
subsref	—
sum	• Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to Specify precision or Keep LSB.
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. • When you provide complex inputs to the times function inside of an Embedded MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the **Ports and data manager** and set the **Complexity** parameter for all known complex inputs to On.
transpose	—
tril	• If supplied, the index, k, must be a real and scalar integer value that is not a fi object.
triu	• If supplied, the index, k, must be a real and scalar integer value that is not a fi object.
ufi	—
uint8, uint16, uint32	—
uminus	—
uplus	—

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

Function	Remarks/Limitations
upperbound	—
vertcat	—

Embedded MATLAB MEX

Using Embedded MATLAB MEX with Fixed-Point Algorithms

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	
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

In this section...
"Composing a MATLAB Language Function in a Simulink Model" on page 8-15
"Using the Embedded MATLAB Function Block with Data Type Override" on page 8-16
"Using Fixed-Point Data Types with the Embedded MATLAB Function Block" on page 8-17
"Example: Implementing a Fixed-Point Direct Form FIR Using the Embedded MATLAB Function Block" on page 8-24

Composing a MATLAB Language Function in a Simulink Model

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

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`
Inherited `Fixed`	True doubles	`fi double`
	True singles	`fi single`
Specified `Fixed`	True doubles	`fi double`
	True singles	`fi single`

For more information about using the Embedded MATLAB Function block with data type override, see the following section of the Simulink documentation:

"Using Data Type Override with the Embedded MATLAB Function Block"

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

The Embedded MATLAB subset supports a significant number of Fixed-Point Toolbox functions. Refer to "Supported Functions and Limitations of the Fixed-Point Embedded MATLAB Subset" on page 8-2 for information about which Fixed-Point Toolbox features the Embedded MATLAB subset supports.

For more information on working with fixed-point Embedded MATLAB Function blocks, see the following sections in the Fixed-Point Toolbox documentation:

- "Specifying Fixed-Point Parameters in the Model Explorer" on page 8-17
- "Using fimath Objects in Embedded MATLAB Function Blocks" on page 8-19
- "Sharing Models with Fixed-Point Embedded MATLAB Function Blocks" on page 8-22

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

Specifying Fixed-Point Parameters in the Model Explorer

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

1 Place an Embedded MATLAB Function block in a new model. You can find the block in the Simulink User-Defined Functions library.

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

3 Expand the **untitled*** node in the **Model Hierarchy** pane of the Model Explorer. Then, select the **Embedded MATLAB Function** node. The Model Explorer now appears as shown in the following figure.
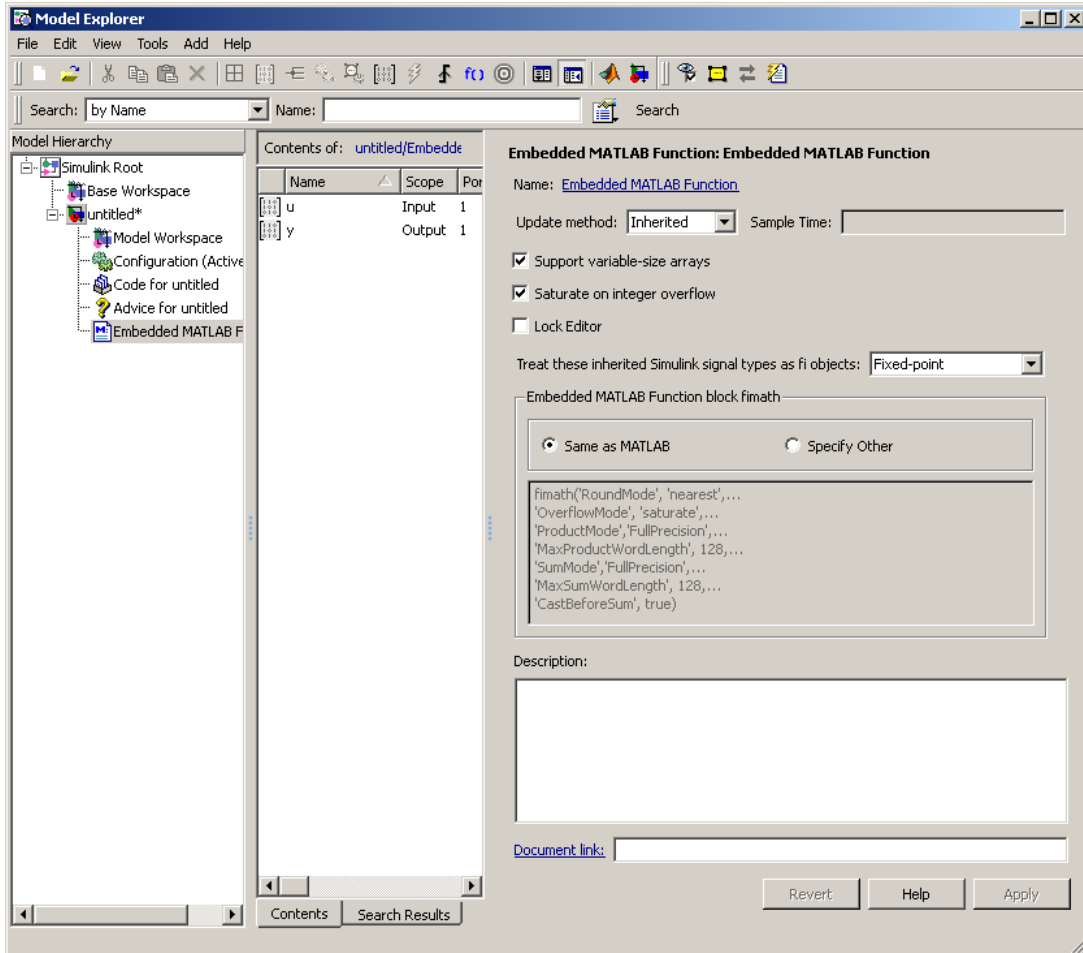
The following parameters in the **Dialog** pane apply to Embedded MATLAB Function blocks in models that use fixed-point and integer data types:

Treat these inherited Simulink signal types as fi objects

Choose whether to treat inherited fixed-point and integer signals as fi objects.

• When you select Fixed-point, the Embedded MATLAB Function block treats all fixed-point inputs as Fixed-Point Toolbox fi objects.

- When you select `Fixed-Point & Integer`, the Embedded MATLAB Function block treats all fixed-point and integer inputs as Fixed-Point Toolbox `fi` objects.

Embedded MATLAB Function block fimath

Specify the `fimath` properties for the block to associate with the following objects:

- All fixed-point and integer input signals to the Embedded MATLAB Function block that you choose to treat as `fi` objects.

- All `fi` and `fimath` objects constructed in the Embedded MATLAB Function block.

You can select one of the following options for the **Embedded MATLAB Function block fimath**:

- **Same as MATLAB** — When you select this option, the block uses the same `fimath` properties as the current global fimath. The edit box appears dimmed and displays the current global fimath in read-only form.

- **Specify other** — When you select this option, you can specify your own `fimath` object in the edit box.

For more information on these parameters, see "Using fimath Objects in Embedded MATLAB Function Blocks" on page 8-19.

Using fimath Objects in Embedded MATLAB Function Blocks

The **Embedded MATLAB Function block fimath** parameter enables you to specify one set of `fimath` object properties for the Embedded MATLAB Function block. The block associates the `fimath` properties you specify with the following objects:

- All fixed-point and integer input signals to the Embedded MATLAB Function block that you choose to treat as `fi` objects.

- All `fi` and `fimath` objects constructed in the Embedded MATLAB Function block.

You can set these parameters on the following dialog box, which you can access through either "The Model Explorer" or the "Ports and Data Manager".

Embedded MATLAB Function: Embedded MATLAB Function

Name: Embedded MATLAB Function

Update method: Inherited ▾   Sample Time: [   ]

☑ Support variable-size arrays

☑ Saturate on integer overflow

☐ Lock Editor

Treat these inherited Simulink signal types as fi objects: Fixed-point ▾

Embedded MATLAB Function block fimath

  ◉ Same as MATLAB   ○ Specify Other

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

Description:

[   ]

Document link: [   ]

[Revert] [Help] [Apply]

- To access this pane through the Model Explorer:

  - Select **View** > **Model Explorer** from your model menu.

  - Then, select the Embedded MATLAB Function block from the Model Hierarchy pane on the left side of the Model Explorer.

- To access this pane through the Ports and Data Manager, select **Tools** > **Edit Data/Ports** from the Embedded MATLAB Editor menu.

When you select **Same as MATLAB** for the **Embedded MATLAB Function block fimath**, the Embedded MATLAB Function block uses the current global fimath. The current global fimath appears dimmed and in read-only form in the edit box.

When you select **Specify other** the block allows you to specify your own `fimath` object in the edit box. You can do so in one of two ways:

- Constructing the `fimath` object inside the edit box.

- Constructing the `fimath` object in the MATLAB or model workspace and then entering its variable name in the edit box.

Note If you use this option and plan to share your model with others, make sure you define the variable in the model workspace. See "Sharing Models with Fixed-Point Embedded MATLAB Function Blocks" on page 8-22 for more information on sharing models.

The Embedded MATLAB subset supports the Fixed-Point Toolbox `isfimathlocal` function. The Embedded MATLAB subset does not support the other Fixed-Point Toolbox functions related to the global fimath because they change the MATLAB preferences. These unsupported functions include:

- `savedefaultfimathpref`

- `setdefaultfimath`

- `removedefaultfimathpref`

- `resetdefaultfimath`

Sharing Models with Fixed-Point Embedded MATLAB Function Blocks

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

1 Place an Embedded MATLAB Function block in a new model. You can find the block in the Simulink User-Defined Functions library.

2 Define a `fimath` object in the MATLAB workspace that you want to use for any Simulink fixed-point signal entering the 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 Select **Specify other** for the **Embedded MATLAB Function block fimath** parameter and enter the variable `F` into the edit box on the **Dialog** pane. Click **Apply** to save your changes.

You have now defined the `fimath` properties to be associated with all Simulink fixed-point input signals and all `fi` and `fimath` objects constructed within the block.

6 Select the **Base Workspace** node in the **Model Hierarchy** pane. You can see the variable F that you have defined in the MATLAB workspace listed in the **Contents** pane. If you send this model to a coworker, that coworker must first define that same variable in the MATLAB workspace to get the same results.

7 Cut the variable F from the base workspace, and paste it into the model workspace listed under the node for your model, in this case **untitled***. The Model Explorer now appears as shown in the following figure.
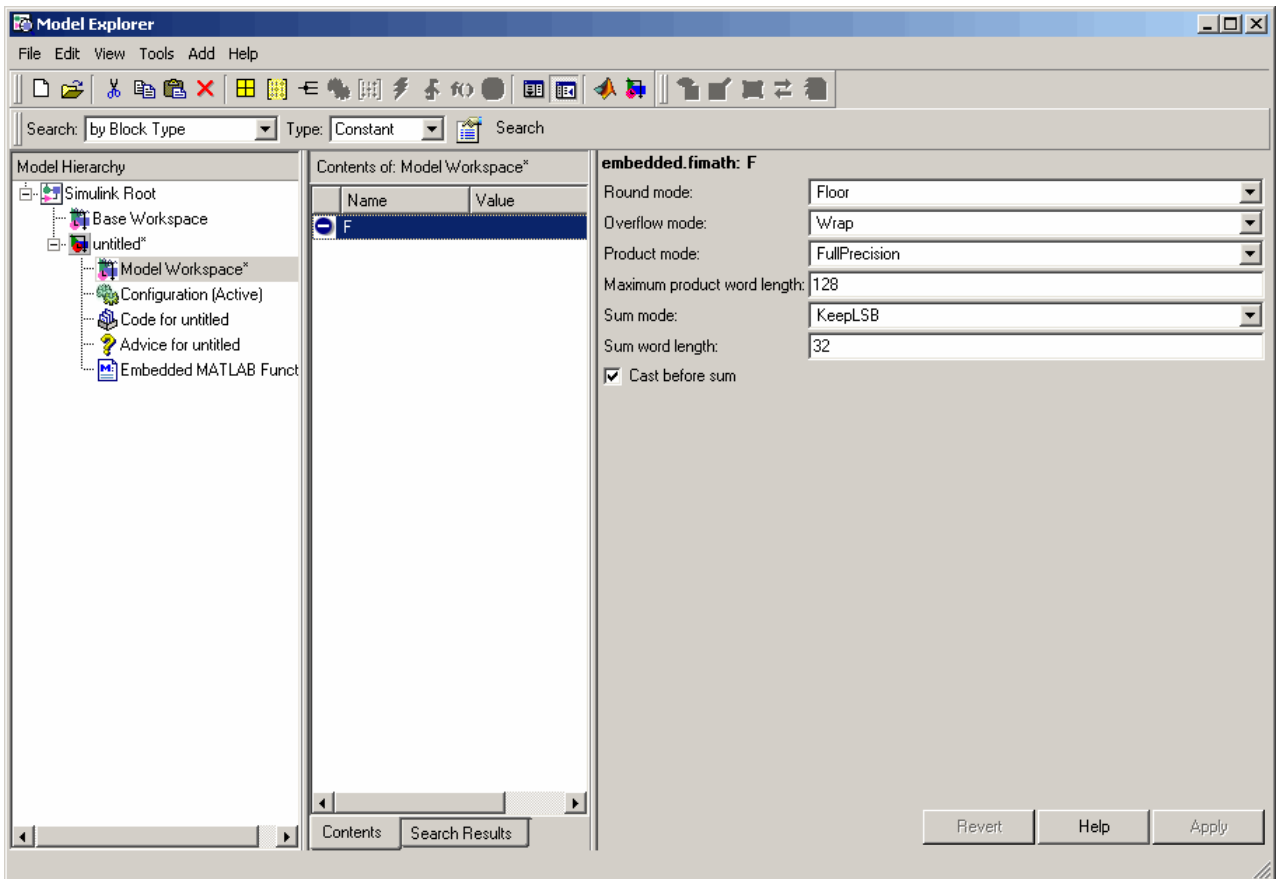
You can now e-mail your model to a coworker. Because you included the required variables in the workspace of the model itself, your coworker can simply run the model and get the correct results. Receiving and running the model does not require any extra steps.

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

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

- "Program the Embedded MATLAB Function Block" on page 8-24
- "Prepare the Inputs" on page 8-26
- "Create the Model" on page 8-27
- "Define the fimath Object Using the Model Explorer" on page 8-31
- "Run the Simulation" on page 8-33

Program the Embedded MATLAB Function Block

1 Place an Embedded MATLAB Function block in a new model. You can find the block 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
Ty = numerictype(1,12,8);
yout = fi(zeros(size(x)),'numerictype',Ty);
zf = zi;

% FIR filter code
```

```
for k=1:length(x);
  % Update the states: z = [x(k);z(1:end-1)]
  zf(:) = [x(k);zf(1:end-1)];
  % Form the output: y(k) = b*z
  yout(k) = b*zf;
end

% Plot the outputs only in simulation.
% This does not generate C code.
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 now appears as shown in the following figure.

Prepare the Inputs

Define the filter coefficients *b*, noise *x*, and initial conditions *zi* by typing the following code at the MATLAB command line:

```
b = fi_fir_coefficients;
load mtlb
x = mtlb;
n = length(x);
noise = sin(2*pi*2140*(0:n-1)'./Fs);
```

```
x = x + noise;
zi = zeros(length(b),1);
```

Create the Model

1 Add blocks to your model to create the following system.
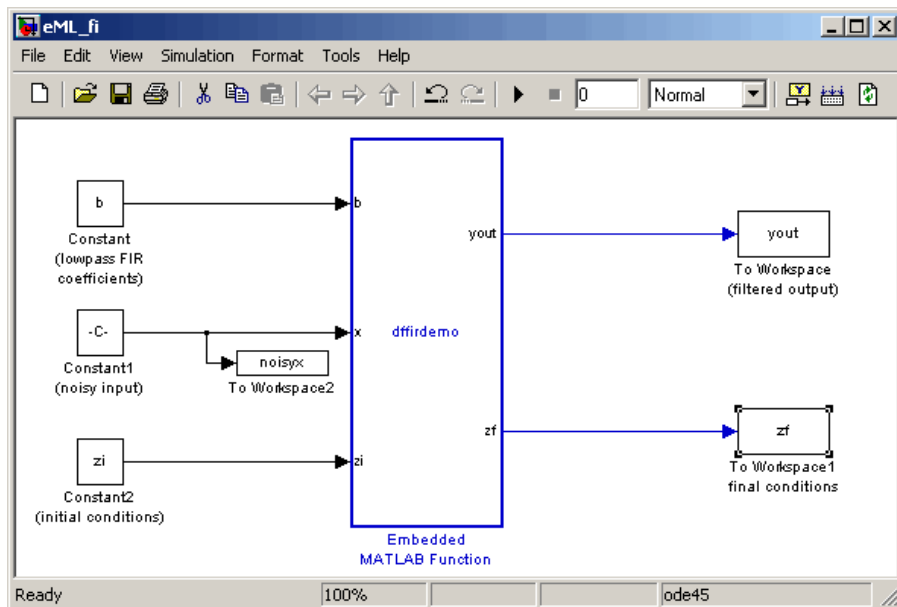
2 Set the block parameters in the model to the following values.

Block	Parameter	Value
Constant	**Constant value**	b
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	Sample based
	Sample time	inf
	Mode	Fixed point
	Signedness	Signed
	Scaling	Slope and bias
	Word length	12
	Slope	2^-12
	Bias	0
Constant1	**Constant value**	x+noise
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	Sample based
	Sample time	1
	Mode	Fixed point
	Signedness	Signed
	Scaling	Slope and bias
	Word length	12
	Slope	2^-8
	Bias	0

Block	Parameter	Value
Constant2	**Constant value**	`zi`
	Interpret vector parameters as 1-D	Unselected
	Sampling mode	`Sample based`
	Sample time	`inf`
	Mode	`Fixed point`
	Signedness	`Signed`
	Scaling	`Slope and bias`
	Word length	`12`
	Slope	`2^-8`
	Bias	`0`
To Workspace	**Variable name**	`yout`
	Limit data points to last	`inf`
	Decimation	`1`
	Sample time	`-1`
	Save format	`Array`
	Log fixed-point data as a fi object	Selected

Block	Parameter	Value
To Workspace1	Variable name	zf
	Limit data points to last	inf
	Decimation	1
	Sample time	-1
	Save format	Array
	Log fixed-point data as a fi object	Selected
To Workspace2	Variable name	noisyx
	Limit data points to last	inf
	Decimation	1
	Sample time	-1
	Save format	Array
	Log fixed-point data as a fi object	Selected

3 From the model menu, select **Simulation > Configuration Parameters** and set the following parameters.

Parameter	Value
Stop time	0
Type	Variable-step
Solver	Discrete (no continuous states)

Click **Apply** to save your changes.

Define the fimath Object Using the Model Explorer

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

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

3 Select **Specify other** for the **Embedded MATLAB Function block fimath** parameter on the Embedded MATLAB Function block dialog box. You can then create the following fimath object in the edit box:

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

The fimath object you define here is associated with fixed-point inputs to the Embedded MATLAB Function block as well as the fi object you construct within the block.

Click **Apply** to save your changes. The model explorer now appears as shown in the following figure.
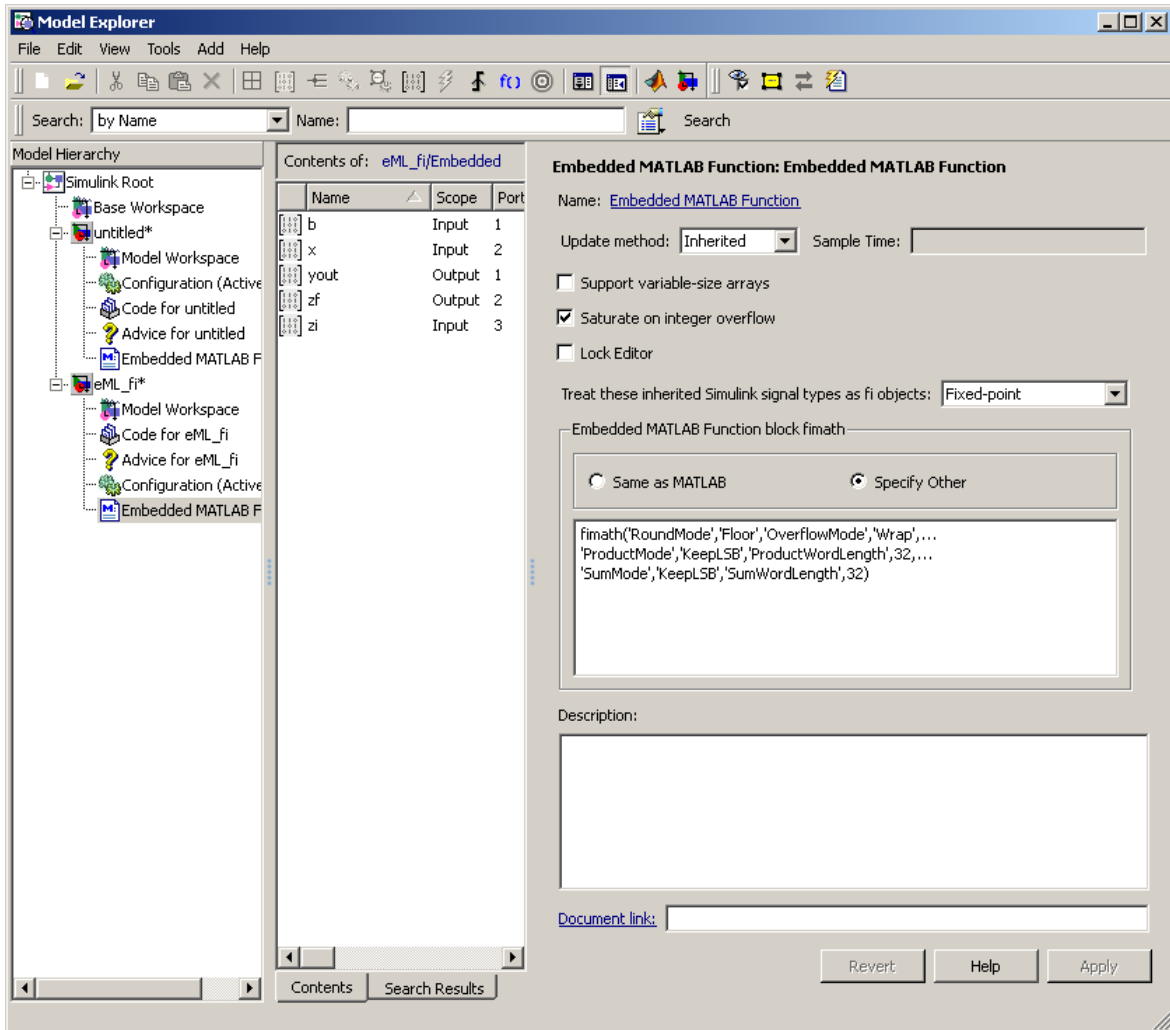
By selecting **Specify other** for the **Embedded MATLAB Function block fimath**, you ensure that your model always uses the `fimath` properties you specified. Thus, the model does not depend on the global fimath and always produces the same results.

Run the Simulation

1 Run the simulation by selecting your model and typing **Ctrl+T**. While the simulation is running, information outputs to the MATLAB command line. You can look at the plots of the noisy signal and the filtered signal.

2 Next, build embeddable C code for your model by selecting the model and typing **Ctrl+B**. While the code is building, information outputs to the MATLAB command line. A folder called `eML_fi_grt_rtw` is created in your current working folder.

3 Navigate to `eML_fi_grt_rtw` > `eML_fi.c`. In this file, you can see the code generated from your model. Search for the following comment in your code:

```
%eML_fi doc model example
```

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

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 "Converting MATLAB Code to C Code" in the Real-Time Workshop product documentation.

Working with Fixed-Point Compilation Reports

In this section...

Generating the Compilation Report

When you compile your M-code with the Embedded MATLAB `emlmex` function or the Real-Time Workshop `emlc` function, you can use the `-report` option to generate a compilation report. This compilation report allows you to examine the data types of the variables and expressions in your code.

To see an example of the compilation report generated by the `emlmex` function, compile `cordic_atan_fixpt.m`. This file ships as a part of the Fixed-Point ATAN2 Calculation demo. You can open the M-file by typing the following at the MATLAB command line:

```
open cordic_atan_fixpt
```

To compile the `cordic_atan_fixpt.m` file, you must provide inputs `x`, `y`, `N`, and `angleLUT`. This example uses the following input values:

```
x = fi(0.23);
y = x;
N = 12;
Tz = numerictype(1,16,13);
angleLUT = fi(atan(2.^-(0:N-1)), 'NumericType', Tz);
```

After you define the input variables in the MATLAB workspace, change your working folder to a local folder and compile the M-file using `emlmex`. Use the `-report` option to generate the compilation report:

```
emlmex cordic_atan_fixpt -eg {x,y,N,angleLUT} -report
```

Opening the Compilation Report

If the compilation is successful, you receive the following message:

`C-MEX generation successful:` Open coder report.

Click the `Open coder report` link to view the report.

If the compilation fails, a link to the error report appears:

`C-MEX generation failed:` Open error report.

Click the `Open error report` link to view the error report and debug your code. For more information on working with error reports, see "Working with Compilation Reports" in the Embedded MATLAB subset documentation.

Viewing Your M-Code

When the compilation report opens, you can hover your cursor over the variables and expressions in your M-code to see their data type information. The compilation report provides color-coded data type information according to the following legend.

Color	Meaning
Green	Data type information is available for the selected variable at this location in the code.
Orange	There is a warning message associated with the selected variable or expression.
Pink	No data type information is available for the selected variable.
Purple	Data type information is available for the selected expression at this location in the code.
Red	There is an error message associated with the selected variable or expression.

Variables in your M-code that have data type information available appear highlighted in green, as shown in the following figure.

Function: **cordic_atan_fixpt** (h:/Documents/My Models/cordic_atan_fixpt.m)

```
 1    function [z,x,y] = cordic_atan_fixpt(y,x,N,angleLUT) %#eml
 2    % Calculate arctangent in range [-pi/2, pi/2] using Vectoring mode CORDIC
 3    % algorithm. Both a and y inputs must be real scalar, x must >= 0.
 4    % Full precision Fimath is used in all fixed-point operations
 5    %
 6    % Inputs:
 7    %  y  : y coordinate or imaginary part of the input vector
 8    %  x  : x coordinate or real part of the input vector
 9    %  N  : total number of iterations, must be a non-negative integer
10    % angleLUT : the angle Llook-up table, has same numerictype as the output
11    %            angle
12    % Output:
13    %  z  : angle that equals atan2(y,x), in radius
14    %       the output angle range is within [-pi/2, +pi/2]
15    %  x  : x coordinate of the last vector at the end of the iterations
16    %  y  : y coordinate of the last vector at the end of the iterations
17    %
18    %    Copyright 1984-2008 The MathWorks, Inc.
19    %    $Revision: 1.1.4.1 $  $Date: 2008/11/19 14:40:34 $
20
21    % initialization
22    tmp = fi(0, numerictype(x)); % Temporary register
23    z = fi(0, numerictype(angleLUT)); % assum z_{0} is 0
24
25    for i = 0:N-1,
26        if y < 0 % negative y leads to counter clock-wise rotation
27            tmp(:) = x - bitsra(y,i); % x_{i+1} = x_{i} - y_{i}>>i
28            y(:) = y + bitsra(x,i);   % y_{i+1} = y_{i} + x_{i}>>i
29            x(:) = tmp;
30            z(:) = z - angleLUT(i
31
32        else  % positive y leads
33            tmp(:) = x + bitsra(y
34            y(:) = y - bitsra(x,i
35            x(:) = tmp;
36            z(:) = z + angleLUT(i
37        end
38    end
```

Information for the selected variable:

Size	1x1
Complex	No
Class	embedded.fi
Signed	Yes
WL	16
FL	17

Expressions in your M-code that have data type information available appear highlighted in purple, as the next figure shows.

```
21    % initialization
22    tmp = fi(0, numerictype(x)); % Temporary register
23    z = fi(0, numerictype(angleLUT)); % assum z_{0} is 0
24
25    for i
26       i    Information for the selected expression:
27              Size   1x1    egative y leads to counter clock-wise rotation
28              Complex No   x - bitsra(y,i); % x_{i+1} = x_{i} - y_{i}>>i
                 Class  embedded.fi  sra(x,i);    % y_{i+1} = y_{i} + x_{i}>>i
29              Signed  Yes   mp;
30                WL  16    - angleLUT(i+1); % z_{i+1} = z_{i} + atan(2^{-i})
31                FL  13
32       else    % positive y leads to clock-wise rotation
33          tmp(:) = x + bitsra(y,i); % x_{i+1} = x_{i} + y_{i}>>i
34          y(:) = y - bitsra(x,i); % y_{i+1} = y_{i} - x_{i}>>i
35          x(:) = tmp;
36          z(:) = z + angleLUT(i+1); % z_{i+1} = z_{i} - atan(2^{-i})
37       end
38    end
```

Viewing Variables in the Variables Tab

To see the data type information for all the variables in your M-file, click on the **Variables** tab of the compilation report. You can expand all `fi` and `fimath` objects listed in the Variables tab to display the properties of the associated `fimath` object, as shown in the following figure.

Variable	Type	Size	Complex	Class	Signed	WL	FL
⊟ z	Output variable	1x1	No	embedded.fi	Yes	16	13

Fimath:

Round mode: **nearest**	Sum mode: **FullPrecision**
Overflow mode: **saturate**	Maximum sum word length: **128**
Product mode: **FullPrecision**	Cast before sum: **Yes**
Maximum product word length: **128**	

Variable	Type	Size	Complex	Class	Signed	WL	FL
⊞ x	Output variable	1x1	No	embedded.fi	Yes	16	17
⊞ y	Output variable	1x1	No	embedded.fi	Yes	16	17
N	Input variable	1x1	No	double	-	-	-
⊞ angleLUT	Input variable	1x12	No	embedded.fi	Yes	16	13
⊞ tmp	Local variable	1x1	No	embedded.fi	Yes	16	17
i	Local variable	1x1	No	double	-	-	-

You can sort the variables by clicking the column headings in the **Variables** tab. To sort the variables by multiple columns, hold down the **Shift** key while clicking on the column headings.

See Also

For more information about using the compilation report with the `emlmex` function, see "Working with Compilation Reports" in the Embedded MATLAB documentation.

For information about using the compilation report with the `emlc` function, see "Working with Compilation Reports" in the Real-Time Workshop documentation.

9

Interoperability with Other Products

- "Using fi Objects with Simulink" on page 9-2
- "Using the Embedded MATLAB Subset" on page 9-7
- "Using fi Objects with Signal Processing Blockset Software" on page 9-8
- "Using fi Objects with Filter Design Toolbox Software" on page 9-13

Using fi Objects with Simulink

Reading Fixed-Point Data from the Workspace

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

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

Writing Fixed-Point Data to the Workspace

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

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

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

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

a =

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


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

s.signals.values = a

s =

    signals: [1x1 struct]

s.signals.dimensions = 2

s =

    signals: [1x1 struct]

s.time = [0:10]'
```

```
s =

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

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

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

In the model, the following parameters in the **Solver** pane of the **Configuration Parameters** dialog have the indicated settings:

- **Start time** — `0.0`
- **Stop time** — `10.0`
- **Type** — `Fixed-step`
- **Solver** — `Discrete (no continuous states)`
- **Fixed step size (fundamental sample time)** — `1.0`
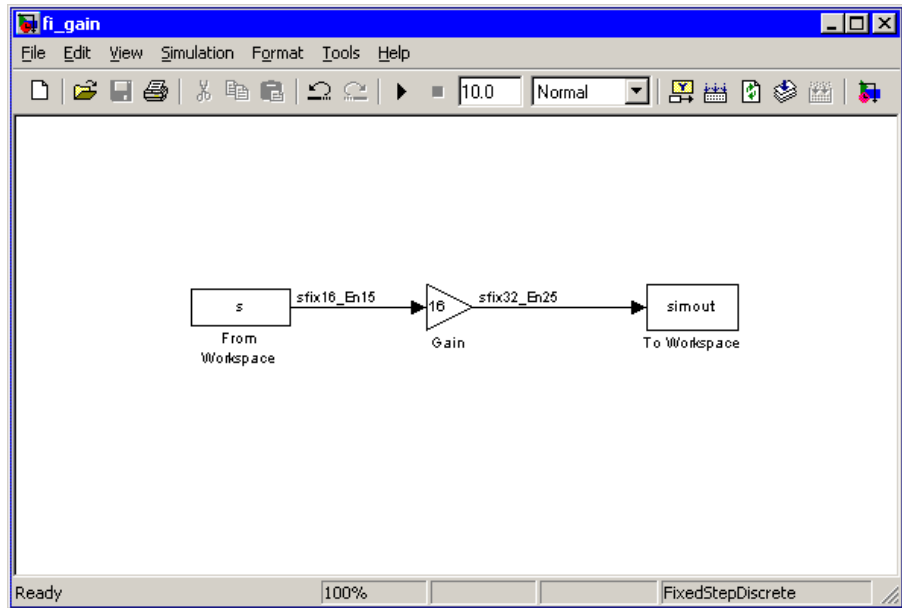
The To Workspace block writes the result of the simulation to the MATLAB workspace as a fi structure.

```
simout.signals.values

ans =

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

```
DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 32
 FractionLength: 25
```

Setting the Value and Data Type of Block Parameters

You can use Fixed-Point Toolbox expressions to specify the value and data type of block parameters in Simulink. Refer to "Block Support for Data and Numeric Signal Types" in the Simulink documentation for more information.

Logging Fixed-Point Signals

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

There are several ways to use Fixed-Point Toolbox software with the Embedded MATLAB subset:

- The Embedded MATLAB `emlmex` function converts your M-code to a C-MEX function and can greatly accelerate the execution speed of your fixed-point algorithms in MATLAB. To use the `emlmex` function, you need a Fixed-Point Toolbox license. For more information, see "Working with Embedded MATLAB MEX" in the Embedded MATLAB documentation.

- The Embedded MATLAB Coder (`emlc`) is a Real-Time Workshop function that automatically converts M-code to C code. Using the Embedded MATLAB Coder allows you to accelerate your MATLAB code that uses Fixed-Point Toolbox software. To use the `emlc` function with Fixed-Point Toolbox software, you also need to have licenses for Simulink and Real-Time Workshop code generation software. For more information, see "Converting MATLAB Code to C Code" in the Real-Time Workshop documentation.

- The Embedded MATLAB Function block allows you to use MATLAB code in your Simulink models that generate embeddable C code. To use the Embedded MATLAB Function block with Fixed-Point Toolbox software, you also need a Simulink license. For more information on the Embedded MATLAB Function block, see "Using the Embedded MATLAB Function Block" in the Simulink documentation.

For more information on using the Embedded MATLAB subset with Fixed-Point Toolbox software, see Chapter 8, "Working with the Fixed-Point Embedded MATLAB Subset".

Using fi Objects with Signal Processing Blockset Software

In this section...
"Reading Fixed-Point Signals from the Workspace" on page 9-8
"Writing Fixed-Point Signals to the Workspace" on page 9-8

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
           Signedness: Signed
           WordLength: 16
        FractionLength: 15
```

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

- **Signal** — a
- **Sample time** — 1
- **Samples per frame** — 2
- **Form output after final data value by** — Setting to zero

The following parameters in the **Solver** pane of the **Configuration Parameters** dialog have these settings:

- **Start time** — 0.0

- **Stop time** — 10.0
- **Type** — Fixed-step
- **Solver** — Discrete (no continuous states)
- **Fixed step size (fundamental sample time)** — 1.0

Remember, to write fixed-point data to the MATLAB workspace as a `fi` object, select the **Log fixed-point data as a fi object** check box on the Signal To Workspace block dialog. Otherwise, fixed-point data is converted to `double` and written to the workspace as `double`.
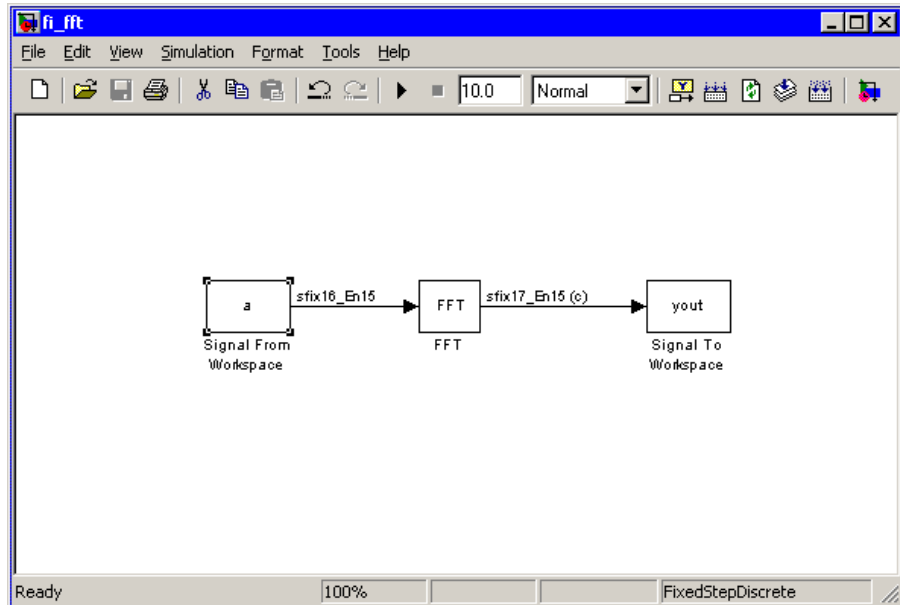
The Signal To Workspace block writes the result of the simulation to the MATLAB workspace as a `fi` object.

```
yout =


(:,:,1) =
```

```
     0.8415   -0.1319
    -0.8415   -0.9561


(:,:,2) =

     1.0504    1.6463
     0.7682    0.3324


(:,:,3) =

    -1.7157   -1.2383
     0.2021    0.6795


(:,:,4) =

     0.3776   -0.6157
    -0.9364   -0.8979


(:,:,5) =

     1.4015    1.7508
     0.5772    0.0678


(:,:,6) =

    -0.5440         0
    -0.5440         0
```

```
     DataTypeMode: Fixed-point: binary point scaling
       Signedness: Signed
       WordLength: 17
   FractionLength: 15
```

Using fi Objects with 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

W

wrapping