

MATLAB® Compiler SDK™

.NET User's Guide



MATLAB®

R2021b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



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

MATLAB® Compiler SDK™ .NET User's Guide

© COPYRIGHT 2002–2021 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

March 2015	Online only	New for Version 6.0 (Release 2015a)
September 2015	Online only	Revised for Version 6.1 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)
September 2016	Online only	Revised for Version 6.3 (Release R2016b)
March 2017	Online only	Revised for Version 6.3.1 (Release R2017a)
September 2017	Online only	Revised for Version 6.4 (Release R2017b)
March 2018	Online only	Revised for Version 6.5 (Release R2018a)
September 2018	Online only	Revised for Version 6.6 (Release R2018b)
March 2019	Online only	Revised for Version 6.6.1 (Release R2019a)
September 2019	Online only	Revised for Version 6.7 (Release R2019b)
March 2020	Online only	Revised for Version 6.8 (Release R2020a)
September 2020	Online only	Revised for Version 6.9 (Release R2020b)
March 2021	Online only	Revised for Version 6.10 (Release R2021a)
September 2021	Online only	Revised for Version 6.11 (Release R2021b)

Getting Started

1

MATLAB Compiler SDK .NET Target Requirements	1-2
System and Product Requirements	1-2
Supported Microsoft .NET Framework Versions	1-2
MATLAB Compiler SDK .NET Limitations	1-3
Path Modifications Required for Accessibility	1-3

Component Integration

2

Common Integration Tasks and Naming Conventions	2-2
Component Access On Another Computer	2-2
Component and Class Naming Conventions	2-2
Integrate a .NET Assembly Into a C# Application	2-4
Data Marshaling with MWArray API	2-6
MWArray Data Conversion Classes	2-6
Pass Data from .NET Code to MATLAB	2-6
Pass Data from MATLAB to .NET Code	2-7
Convert Data Between .NET and MATLAB	2-8
Manually Cast to MATLAB Types	2-8
Pass Input Arguments	2-11
Query Type of Return Value	2-11
Pass Objects by Reference	2-14
Access Real or Imaginary Components Within Complex Arrays	2-16
Jagged Array Processing	2-17
Block Console Display When Creating Figures	2-19
WaitForFiguresToDie Method	2-19
Using WaitForFiguresToDie to Block Execution	2-19
Error Handling and Resource Management	2-21
Error Handling	2-21
Freeing Resources Explicitly	2-22
Limitations on Multiple Assemblies in Single Application	2-23
Work with MATLAB Function Handles	2-23
Work with Objects	2-24

3

Integrate Simple MATLAB Function Into .NET Application	3-2
Prerequisites	3-2
Create Simple Plot	3-2
Create Phone Book	3-7
Integrate Function with Variable Number of Arguments	3-13
Files	3-13
Procedure	3-13
Use Multiple Classes in .NET Assembly	3-19
SpectraComp Application	3-19
Files	3-19
Procedure	3-19
Assign Multiple MATLAB Functions in Component Class	3-27
MatrixMathApp Application	3-27
Files	3-27
Procedure	3-28
Understanding the MatrixMath Program	3-34
Integrate MATLAB Optimization Routines with Objective Functions ...	3-35
Overview	3-35
OptimizeComp Application	3-35
Files	3-35
Procedure	3-36
Build .NET Core Application That Runs on Linux and macOS	3-40
Prerequisites	3-40
Create .NET Assembly	3-40
Create .NET Core Application	3-40
Run .NET Core Application on UNIX	3-43

Microsoft Visual Basic Integration Examples

4

Integrate a .NET Assembly Into a Visual Basic Application	4-2
--	------------

Distribute Integrated .NET Applications

5

Package .NET Applications	5-2
About the MATLAB Runtime	5-3
How is the MATLAB Runtime Different from MATLAB?	5-3
Performance Considerations and the MATLAB Runtime	5-3

Install and Configure MATLAB Runtime	5-4
Download MATLAB Runtime Installer	5-4
Install MATLAB Runtime Interactively	5-4
Install MATLAB Runtime Noninteractively	5-6
Install MATLAB Runtime without Administrator Rights	5-7
Install Multiple MATLAB Runtime Versions on Single Machine	5-7
Install MATLAB and MATLAB Runtime on Same Machine	5-8
Uninstall MATLAB Runtime	5-8

Distribute to End Users

6

Deploy Components to End Users	6-2
Running the Component Installer	6-2
MATLAB Runtime	6-3
MATLAB Runtime Run-Time Options	6-5
What Run-Time Options Can You Specify?	6-5
Getting MATLAB Runtime Option Values Using MWMCR	6-5
MATLAB Runtime User Data Interface	6-7
Supplying Cluster Profiles for Parallel Computing Toolbox Applications ..	6-7
MATLAB Runtime Component Cache and Deployable Archive Embedding	6-11
Impersonation Implementation Using ASP.NET	6-12
Enhanced XML Documentation Files	6-15

Type-Safe Interfaces, WCF, and MEF

7

Type-Safe Interfaces	7-2
Manual Data Marshaling Without a Type-Safe Interface	7-2
Simplified Data Marshaling With a Type-Safe Interface	7-3
How Type-Safe Interfaces Work	7-4
Implement Type-Safe Interface and Integrate into .NET Application	7-7
Write and Test Your MATLAB Code	7-7
Implement Type-Safe Interface	7-7
Create .NET Assembly Using Library Compiler App	7-8
Create .NET Assembly Using compiler.build.dotNETAssembly	7-8
Integrate .NET Assembly Into .NET Application	7-9
Tips	7-11
Create Managed Extensibility Framework Plug-Ins	7-12
Prerequisites	7-12
Addition and Multiplication Applications with MEF	7-12

Create an MEFHost Assembly	7-13
Create a Contract Interface Assembly	7-13
Create a Metadata Attribute Assembly	7-14
Add Contract and Attributes References to MEFHost	7-15
Compile Your Code in Microsoft Visual Studio	7-15
Write MATLAB Functions for MEF Parts	7-15
Create Metadata Files	7-15
Build .NET Components from MATLAB Functions and Metadata	7-16
Install MEF Parts	7-17
Run the MEF Host Program	7-17

Windows Communications Foundation Based Components

8

Create Windows Communications Foundation Component	8-2
Write and Test Your MATLAB Code	8-2
Implement WCF Interface	8-2
Create .NET Assembly Using Library Compiler App	8-3
Create .NET Assembly Using compiler.build.dotNETAssembly	8-4
Develop Server Program Using WCF Interface	8-5
Generate Proxy Code for Clients	8-7
Develop Client Program Using WCF Interface	8-7
Tips	8-9

.NET Remoting

9

.NET Remoting and Windows Communications Foundation	9-2
.NET Remoting	9-2
Windows Communications Foundation	9-2
What's the Difference Between WCF and .NET Remoting?	9-3
Compare MWArray and Native .NET API for Remotable Assemblies	9-4
Using Native .NET Structure and Cell Arrays	9-4
Create Remotable .NET Assembly	9-6
Preparation	9-6
Build Remotable Component Using Library Compiler App	9-6
Build Remotable Component Using compiler.build.dotNETAssembly	9-7
Files Generated by the Compilation Process	9-7
Access Remotable .NET Assembly Using MWArray API	9-9
Coding and Building the Hosting Server Application and Configuration File	9-9
Build Client Application and Configuration File	9-10
Start Server Application	9-12
Start Client Application	9-12

Access Remotable .NET Assembly Using Native .NET API: Magic Square	9-14
Why Use the Native .NET API?	9-14
Coding and Building the Hosting Server Application and Configuration File	9-14
Coding and Building the Client Application and Configuration File	9-15
Starting the Server Application	9-17
Starting the Client Application	9-17
Access Remotable .NET Assembly Using Native .NET API: Cell and Struct	9-19
Why Use the .NET API With Cell Arrays and Structs?	9-19
Building Your Component	9-19
The Native .NET Cell and Struct Example	9-19
Coding and Building the Client Application and Configuration File	9-20
Starting the Server Application	9-22
Starting the Client Application	9-23
Coding and Building the Client Application and Configuration File with the Native MWArray, MWStructArray, and MWCellArray Classes	9-24

Troubleshooting

10

Failure to Find MATLAB Runtime Files	10-2
Failure to Find MATLAB Classes	10-3
Diagnostic Messages	10-4
Enhanced Error Diagnostics Using mstack Trace	10-6

Reference Information

11

Rules for Data Conversion Between .NET and MATLAB	11-2
Managed .NET Types to MATLAB Arrays	11-2
MATLAB Arrays to Managed .NET Types	11-2
.NET Types to MATLAB Types	11-3
Character and String Conversion	11-7
Unsupported MATLAB Array Types	11-7
Interfaces Generated by the MATLAB Compiler SDK	11-9
Single Output API	11-9
Standard API	11-10
feval API	11-11

Deploying .NET Components With the F# Programming Language

A

Integrate .NET Assembly Into F# Application	A-2
Prerequisites	A-2
Step 1: Build the Component	A-2
Step 2: Integrate Component Into F# Application	A-2
Step 3: Deploy the Component	A-4

Getting Started

MATLAB Compiler SDK .NET Target Requirements

In this section...
“System and Product Requirements” on page 1-2
“Supported Microsoft .NET Framework Versions” on page 1-2
“MATLAB Compiler SDK .NET Limitations” on page 1-3
“Path Modifications Required for Accessibility” on page 1-3

System and Product Requirements

You must have the MATLAB and MATLAB Compiler™ products installed to install the MATLAB Compiler SDK product.

The MATLAB Compiler SDK .NET target is available only on Windows®.

For an up-to-date list of all the system and compiler software supported by MATLAB, MATLAB Compiler, and MATLAB Compiler SDK, see https://www.mathworks.com/support/compilers/current_release/.

Supported Microsoft .NET Framework Versions

Install the supported version of the Microsoft® .NET Framework. Your ability to use the latest MATLAB Compiler SDK functionality often depends on having the most current version of the framework installed.

MATLAB Compiler SDK supports version 4.0 of Microsoft .NET Framework.

Building a New Assembly

If you are building a new assembly, you need .NET Framework version 4.0 or above (such as 4.5 or 4.6).

- If you have both 4.x and an older version of .NET Framework (2x-3.x), you should be able to build the assembly.
- If you have only an older version of .NET Framework (2.x-3.x), you need to install version 4.0 or above to build a new assembly.

Loading a Deployed Application

If you are loading a deployed application that references an assembly built with version 4.0 or above, you need .NET Framework version 4.0 or above (such as 4.5 or 4.6).

- As long as you have .NET Framework version 4.0 or above installed, you can load a deployed application built with .NET Framework version 4.0 or above. This is true even if the .NET Framework used for building the assembly has a version higher than the one used for loading the application. The reason is that only features in .NET Framework version 4.0 are used when building the assembly.
- If you have both 4.x and an older version of .NET Framework (2x-3.x), you can load a deployed application.

- If you only have an older version of .NET Framework (2.x-3.x), you need to install version 4.0 or above to load a deployed application.

Building a .NET Application

Building .NET applications is not impacted by the version of .NET Framework 4.x used to build the assembly, provided that the version of Microsoft Visual Studio® supports .NET Framework version 4.0 or above.

MATLAB Compiler SDK .NET Limitations

Using addAssembly (External Interfaces)

.NET assemblies or DLLs built with MATLAB Compiler SDK cannot be loaded back into MATLAB with the .NET External Interface method `addAssembly`.

Serialization of MATLAB Objects Unsupported

There is no support in MATLAB Compiler SDK for serializing MATLAB objects from MATLAB into .NET code.

Path Modifications Required for Accessibility

To use some screen-readers or assistive technologies, such as JAWS®, you must add the following DLLs to your Windows path:

```
matlabroot\sys\java\jre\arch\jre\bin\JavaAccessBridge.dll  
matlabroot\sys\java\jre\arch\jre\bin\WindowsAccessBridge.dll
```


Component Integration

- “Common Integration Tasks and Naming Conventions” on page 2-2
- “Integrate a .NET Assembly Into a C# Application” on page 2-4
- “Data Marshaling with MWArray API” on page 2-6
- “Convert Data Between .NET and MATLAB” on page 2-8
- “Block Console Display When Creating Figures” on page 2-19
- “Error Handling and Resource Management” on page 2-21
- “Limitations on Multiple Assemblies in Single Application” on page 2-23

Common Integration Tasks and Naming Conventions

In this section...

“Component Access On Another Computer” on page 2-2

“Component and Class Naming Conventions” on page 2-2

In “Generate .NET Assembly and Build .NET Application”, steps are illustrated that cover the basics of customizing your code in preparation for integrating your deployed .NET component into a large-scale enterprise application. These steps include:

- Installing MATLAB Runtime on end user computers
- Creating a Microsoft Visual Studio project
- Creating references to the component and the MWArray API
- Specifying component assemblies and namespaces
- Initializing and instantiating your classes
- Invoking the component using some implicit data conversion techniques
- Handling errors using a basic try-catch block

Component Access On Another Computer

To implement your .NET assembly on a computer other than the one on which it was built:

- 1 If the component is not already installed on the machine where you want to develop your application, run the self-extracting executable that you created in “Generate .NET Assembly and Build .NET Application”.
- 2 Reference the .NET assembly in your Microsoft Visual Studio project or from the command line of a CLS-compliant compiler.

You must also add a reference to the MWArray component in `matlabroot\toolbox\dotnetbuilder\bin\architecture\framework_version`. See “Supported Microsoft .NET Framework Versions” on page 1-2 for a list of supported framework versions.

- 3 Instantiate the generated .NET classes and call the class methods as you would with any .NET class. To marshal data between the native .NET types and the MATLAB array type, you need to use either the MWArray data conversion classes or the MWArray native API.

Note For information about these data conversion classes, see the *MWArray Class Library Reference*, which is also available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder.

To avoid using data conversion classes, see “Implement Type-Safe Interface and Integrate into .NET Application” on page 7-7.

Component and Class Naming Conventions

Typically you should specify names for assemblies and classes that will be clear to programmers who use the generated code. For example, if you are encapsulating many MATLAB functions, it helps to determine a scheme of function categories and to create a separate class for each category. Also, the name of each class should be descriptive of what the class does.

The .NET naming guidelines recommends the use of Pascal case for capitalizing the names of identifiers of three or more characters. That is, the first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. For example:

MakeSquare

In contrast, MATLAB programmers typically use all lowercase for names of functions. For example:

makesquare

By convention, the MATLAB Compiler SDK .NET examples use Pascal case.

Valid characters are any alpha or numeric characters, as well as the underscore (`_`) character.

See Also

“Supported Microsoft .NET Framework Versions” on page 1-2

Related Examples

- “Generate .NET Assembly and Build .NET Application”
- “Implement Type-Safe Interface and Integrate into .NET Application” on page 7-7

Integrate a .NET Assembly Into a C# Application

This example shows how to call a .NET assembly from a C# application. To create the .NET assembly from your MATLAB function, see “Generate .NET Assembly and Build .NET Application”.

- 1 Install the .NET assembly from the `for_redistribution` folder.

The generated shared libraries and support files are located in the `for_testing` folder.

- 2 Open Microsoft Visual Studio and create a project. For this example, create a C# Console Application called **MainApp** and create a reference to your assembly file `MagicSquareComp.dll`.
- 3 Add a reference to the `MWArray` API.

If MATLAB is installed on your system	<code>matlabroot\toolbox\dotnetbuilder\bin\win64\<framework_version>\MWArray.dll</code>
If MATLAB Runtime is installed on your system	<code><MATLAB_RUNTIME_INSTALL_DIR>\toolbox\dotnetbuilder\bin\win64\<framework_version>\MWArray.dll</code>

- 4 Go to **Build > Configuration Manager** and change the platform from **Any CPU** to **x64**.
- 5 Copy the following C# code into the project and save it.

C# Code to Implement Application

```
// Make .NET Namespaces available to your generated component.
using System;

using MagicSquareComp;
using MathWorks.MATLAB.NET.Arrays;

// Initialize your classes before you use them.
namespace MainApp
{
    class Program
    {
        {
            static void Main(string[] args)
            {
                Class1 obj = null;
                MWNumericArray input = null;
                MWNumericArray output = null;
                MWArray[] result = null;

                // Because class instantiation and method invocation make their exceptions at run time,
                // you should enclose your code in a try-catch block to handle errors.
                try
                {
                    // Instantiate your component class.
                    obj = new Class1();

                    // Invoke your component.
                    input = 5;
                    result = obj.makesquare(1, input);

                    // Extract the Magic Square you created from the first index of result
                    output = (MWNumericArray)result[0];

                    // print the output.
                    Console.WriteLine(output);
                }
                catch
                {
                    throw;
                }
            }
        }
    }
}
```

- 6 After you finish writing your code, build and run it with Microsoft Visual Studio.

Note When calling your component, you can take advantage of implicit conversion from .NET types to MATLAB types, by passing the native C# value directly to `makeSq`:

```
input = 5;
obj.makesquare(1, input);
```

You can also use explicit conversion:

```
input = new MWNumericArray(5);
obj.makesquare(1, input);
```

Data Marshaling with MWArray API

To support data conversion between managed .NET types and MATLAB types, MATLAB Compiler SDK provides a set of data conversion classes derived from the abstract class `MWArray`. These classes allow you to pass most native .NET value types as parameters directly without using explicit data conversion. You reference the `MWArray` assembly in your managed application to convert native arrays to MATLAB arrays and vice versa. This process is called data marshaling.

When you invoke a method on a component, the input and output parameters are a derived type of `MWArray`. To pass parameters, you can either instantiate one of the `MWArray` subclasses explicitly, or, in many cases, pass the parameters as a managed data type and rely on the implicit data conversion feature of MATLAB Compiler SDK.

For examples that demonstrate guidelines for manual data conversion between various native data types and types compatible with MATLAB, see “Convert Data Between .NET and MATLAB” on page 2-8.

MWArray Data Conversion Classes

The `MWArray` data conversion classes are built as a class hierarchy that represents the major MATLAB array types.

- `MWArray`
- `MWIndexArray`
- `MWCellArray`
- `MWCharacterArray`
- `MWLogicalArray`
- `MWNumericArray`
- `MWStructArray`

The root of the hierarchy is the `MWArray` abstract class. `MWIndexArray` is also an abstract class. The other subclasses represent the major MATLAB array types: `MWNumericArray`, `MWLogicalArray`, `MWCharArray`, `MWCellArray`, and `MWStructArray`.

`MWArray` and its derived classes provide the following functionality:

- Constructors and destructors to instantiate and dispose of MATLAB arrays
- Properties to get and set the underlying array data
- Indexers to support a subset of MATLAB array indexing
- Implicit and explicit data conversion operators
- General methods

For information about these data conversion classes, see the *MWArray Class Library Reference*, which is also available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder.

Pass Data from .NET Code to MATLAB

In most instances, if you use a native .NET primitive or array as an input parameter in a C# program, MATLAB Compiler SDK automatically and transparently converts it to an instance of the appropriate

MWArray class before passing it to the generated method. MATLAB Compiler SDK converts most CLS-compliant strings, numeric types, or multidimensional arrays of these types to an appropriate MWArray type. For a list of the unsupported types, see “Unsupported MATLAB Array Types” on page 11-7. This conversion is transparent in C# applications, but might require an explicit casting operator in other languages, for example, `op_implicit` in Visual Basic®.

For example, consider the .NET statement:

```
result = theFourier.plotfft(3, data, interval);
```

In this statement, the argument `interval` is of the .NET native type `System.Double`. MATLAB Compiler SDK casts this argument to a MATLAB 1-by-1 double `MWNumericArray` type, which is a wrapper class containing a MATLAB double array.

Pass Data from MATLAB to .NET Code

All data returned from a MATLAB function to a .NET method is represented as an instance of the appropriate MWArray subclass. For example, a MATLAB cell array is returned as an `MWCellArray` object.

Returned data is *not* automatically converted to a native array. If you need to get the corresponding native array type, call the `ToArray` method, which converts a MATLAB array to the appropriate native data type, with some exceptions. Cell arrays, structure arrays, and arrays of complex numbers are not available as native .NET types. To represent these data types, you must create an instance of `MWCellArray`, `MWStructArray`, or `MWNumericArray`, respectively.

For a list of the .NET native data types and their equivalents in MATLAB, see “Rules for Data Conversion Between .NET and MATLAB” on page 11-2.

See Also

Related Examples

- “Convert Data Between .NET and MATLAB” on page 2-8
- “Rules for Data Conversion Between .NET and MATLAB” on page 11-2

Convert Data Between .NET and MATLAB

The MATLAB Compiler SDK product provides the `MWArray` assembly to facilitate data conversion between native data and compiled MATLAB functions. For information on the data conversion classes, see “Data Marshaling with `MWArray` API” on page 2-6.

Refer to the following examples for guidelines on how to marshal data between native .NET and MATLAB data types.

Note Data objects that merely pass through either the target or MATLAB environments may not need to be marshaled, particularly if they do not cross a process boundary. Because marshaling is costly, only marshal on demand.

Manually Cast to MATLAB Types

Native Data Conversion

You can explicitly create a numeric constant using the constructor for the `MWNumericArray` class with a `System.Int32` argument. You then pass this variable to one of the generated .NET methods.

```
int data = 24;
MWNumericArray array = new MWNumericArray(data);
Console.WriteLine("Array is of type " + array.NumericType);
```

When you run this example, the results are:

```
Array is of type double
```

The native integer (`int data`) is converted to an `MWNumericArray` containing a 1-by-1 MATLAB double array, which is the default MATLAB type.

To preserve the integer type, use the `MWNumericArray` constructor that provides the ability to control the automatic conversion.

```
MWNumericArray array = new MWNumericArray(data, false);
```

Multidimensional Array Processing

MATLAB and .NET implement different indexing strategies for multidimensional arrays. When you create a variable of type `MWNumericArray`, MATLAB automatically creates an equivalent array, using its own internal indexing. For example, MATLAB indexes using this schema:

```
(row column page1 page2 ...)
```

while .NET indexes as follows:

```
(... page2 page1 row column)
```

Given the multidimensional MATLAB `myarr`:

```
>> myarr(:,:,1) = [1, 2, 3; 4, 5, 6];
>> myarr(:,:,2) = [7, 8, 9; 10, 11, 12];
>> myarr
```

```
myarr(:,:,1) =
    1     2     3
    4     5     6

myarr(:,:,2) =
    7     8     9
   10    11    12
```

You would code this equivalent in .NET:

```
double[, ,] myarr = {{{1.000000, 2.000000, 3.000000},
{4.000000, 5.000000, 6.000000}}, {{7.000000, 8.000000,
9.000000}, {10.000000, 11.000000, 12.000000}}};
```

MWNumericArray Example

Here is a code fragment that shows how to convert a double value (5.0) to a `MWNumericArray` type:

```
MWNumericArray arraySize = 5.0;
magicSquare = magic.MakeSqr(arraySize);
```

After converting and assigning the double value to the variable `arraySize`, you can use the `arraySize` argument with the MATLAB based method without further conversion. In this example, the MATLAB based method is `magic.MakeSqr(arraySize)`.

Specify Array Type

If you want to create a MATLAB numeric array of a specific type, set the optional `makeDouble` argument to `False`. The native type then determines the type of the MATLAB array that is created.

Here, the code specifies that the array should be constructed as a MATLAB 1-by-1 16-bit integer array:

```
short data = 24;
MWNumericArray array = new MWNumericArray(data, false);
Console.WriteLine("Array is of type " + array.NumericType);
```

Running this example produces the following results:

```
Array is of type int16
```

Specify Optional Arguments

In MATLAB, you can use `varargin` and `varargout` to specify arguments that are not required. Consider the following MATLAB function:

```
function y = mysum(varargin)
y = sum([varargin{:}]);
```

This function returns the sum of the inputs. The inputs are provided as a `varargin`, which means that the caller can specify any number of inputs to the function. The result is returned as a scalar double array.

For the `mysum` function, the MATLAB Compiler SDK product generates the following interfaces:

```
// Single output interfaces
public MArray mysum()
public MArray mysum(params MArray[] varargin)
// Standard interface
public MArray[] mysum(int numArgsOut)
public MArray[] mysum(int numArgsOut,
    params MArray[] varargin)
// feval interface
public void mysum(int numArgsOut, ref MArray ArgsOut,
    params MArray[] varargin)
```

You can pass the `varargin` arguments as either an `MArray[]`, or as a list of explicit input arguments. (In C#, the `params` modifier for a method argument specifies that a method accepts any number of parameters of the specific type.) Using `params` allows your code to add any number of optional inputs to the encapsulated MATLAB function.

Here is an example of how you might use the single output interface of the `mysum` method in a .NET application:

```
static void Main(string[] args)
{
    MArray sum= null;
    MySumClass mySumClass = null;
    try
    {
        mySumClass= new MySumClass();
        sum= mySumClass.mysum((double)2, 4);
        Console.WriteLine("Sum= {0}", sum);
        sum= mySumClass.mysum((double)2, 4, 6, 8);
        Console.WriteLine("Sum= {0}", sum);
    }
}
```

The number of input arguments can vary.

Note For this particular signature, you must explicitly cast the first argument to `MArray` or a type other than integer. Doing this distinguishes the signature from the method signature, which takes an integer as the first argument. If the first argument is not explicitly cast to `MArray` or as a type other than integer, the argument can be mistaken as representing the number of output arguments.

Pass a Variable Number of Outputs

When present, `varargout` arguments are handled in the same way that `varargin` arguments are handled. Consider the following MATLAB function:

```
function varargout = randvectors()
for i=1:nargout
    varargout{i} = rand(1, i);
end
```

This function returns a list of random `double` vectors such that the length of the `i`th vector is equal to `i`. The MATLAB Compiler SDK product generates the following .NET interface to this function:

```
public void randvectors()
public MArray[] randvectors(int numArgsOut)
public void randvectors(int numArgsOut, ref MArray[] varargout)
```

In this example, you use the standard interface and request two output arguments.

```
MyVarargOutClass myClass = new MyVarargOutClass();
MArray[] results = myClass.randvectors(2);
Console.WriteLine("First output= {0}", results[0]);
Console.WriteLine("Second output= {0}", results[1]);
```

Pass Input Arguments

The following examples show generated code for the `myprimes` MATLAB function, which has the following definition:

```
function p = myprimes(n)
p = primes(n);
```

Construct a Single Input Argument

The following sample code constructs data as a `MWNumericArray` to be passed as input argument:

```
MWNumericArray data = 5;
MyPrimesClass myClass = new MyPrimesClass();
MArray primes = myClass.myprimes(data);
```

Pass a Native .NET Type

This example passes a native double type to the function.

```
MyPrimesClass myClass = new MyPrimesClass();
MArray primes = myClass.myprimes((double)13);
```

The input argument is converted to a MATLAB 1-by-1 double array, as required by the MATLAB function. This is the default conversion rule for a native double type. For a discussion of the default data conversion for all supported .NET types, see “Rules for Data Conversion Between .NET and MATLAB” on page 11-2.

Use the `feval` Interface

The `feval` interface passes both input and output arguments on the right-hand side of the function call. The output argument `primes` must be preceded by a `ref` attribute.

```
MyPrimesClass myClass = new MyPrimesClass();
MArray[] maxPrimes = new MArray[1];
maxPrimes[0] = new MWNumericArray(13);
MArray[] primes = new MArray[1];
myClass.myprimes(1, ref primes, maxPrimes);
```

Query Type of Return Value

The previous examples show guidelines to use if you know the type and dimensionality of the output argument. Sometimes in MATLAB programming this information is unknown, or can vary. In this case, the code that calls the method might need to query the type and dimensionality of the output arguments.

There are two ways to make the query:

- Use .NET reflection to query any object for its type.
- Use any of several methods provided by the `MWArray` class to query information about the underlying MATLAB array.

.NET Reflection

You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object. You can then invoke the type's methods or access its fields and properties. For more information on reflection, see the MSDN Library.

The following code sample calls the `myprimes` method and then determines the type using reflection. The example assumes that the output is returned as a numeric vector array, but the exact numeric type is unknown.

GetPrimes Using .NET Reflection

```
public void GetPrimes(int n)
{
    MWArray primes= null;
    MyPrimesClass myPrimesClass= null;
    try
    {
        myPrimesClass= new MyPrimesClass();
        primes= myPrimesClass.myprimes((double)n);
        Array primesArray= ((MWNumericArray)primes).
            ToVector(MWArrayComponent.Real);
        if (primesArray is double[])
        {
            double[] doubleArray= (double[])primesArray;
            /* Do something with doubleArray . . . */
        }
        else if (primesArray is float[])
        {
            float[] floatArray= (float[])primesArray;
            /* Do something with floatArray . . . */
        }
        else if (primesArray is int[])
        {
            int[] intArray= (int[])primesArray;
            /*Do something with intArray . . . */
        }
        else if (primesArray is long[])
        {
            long[] longArray= (long[])primesArray;
            /*Do something with longArray . . . */
        }
        else if (primesArray is short[])
        {
            short[] shortArray= (short[])primesArray;
            /*Do something with shortArray . . . */
        }
        else if (primesArray is byte[])
        {
            byte[] byteArray= (byte[])primesArray;
            /*Do something with byteArray . . . */
        }
    }
}
```



```

    }
    else
    {
        throw new ApplicationException("
            Bad type returned from myprimes");
    }
}
}

```

The example uses the `toVector` method to return a .NET primitive array (`primesArray`), which represents the underlying MATLAB array. See the following code fragment from the example:

```

primes= myPrimesClass.myprimes((double)n);
    Array primesArray= ((MWNumericArray)primes).
        ToVector(MWArrayComponent.Real);

```

The `toVector` is a method of the `MWNumericArray` class. It returns a copy of the array component in column major order. The type of the array elements is determined by the data type of the numeric array.

MWArray Query

This example uses the `MWNumericArray.NumericType` method, along with `MWNumericType` enumeration to determine the type of the underlying MATLAB array. See the `switch (numericType)` statement.

GetPrimes Using MWArray Query

```

public void GetPrimes(int n)
{
    MWArray primes= null;
    MyPrimesClass myPrimesClass= null;
    try
    {
        myPrimesClass= new MyPrimesClass();
        primes= myPrimesClass.myprimes((double)n);
        if ((!primes.IsNumericArray) || (2 !=
            primes.NumberofDimensions))
        {
            throw new ApplicationException("Bad type returned
                by mwprimes");
        }
        MWNumericArray _primes= (MWNumericArray)primes;
        MWNumericType numericType= _primes.NumericType;
        Array primesArray= _primes.ToVector(
            MWArrayComponent.Real);
        switch (numericType)
        {
            case MWNumericType.Double:
            {
                double[] doubleArray= (double[])primesArray;
                /* (Do something with doubleArray . . .) */
                break;
            }
            case MWNumericType.Single:
            {
                float[] floatArray= (float[])primesArray;
                /* (Do something with floatArray . . .) */
            }
        }
    }
}

```

```
        break;
    }
    case MWNumericType.Int32:
    {
        int[] intArray= (int[])primesArray;
        /* (Do something with intArray . . .) */
        break;
    }
    case MWNumericType.Int64:
    {
        long[] longArray= (long[])primesArray;
        /* (Do something with longArray . . .) */
        break;
    }
    case MWNumericType.Int16:
    {
        short[] shortArray= (short[])primesArray;
        /* (Do something with shortArray . . .) */
        break;
    }
    case MWNumericType.UInt8:
    {
        byte[] byteArray= (byte[])primesArray;
        /* (Do something with byteArray . . .) */
        break;
    }
    default:
    {
        throw new ApplicationException("Bad type returned
            by myprimes");
    }
}
}
```

The code in the example also checks the dimensionality by calling `NumberOfDimensions`; see the following code fragment:

```
if ((!primes.IsNumericArray) || (2 !=
    primes.NumberofDimensions))
    {
        throw new ApplicationException("Bad type returned
            by mwprimes");
    }
```

This call throws an exception if the array is not numeric and of the proper dimension.

Pass Objects by Reference

`MWObjectArray`, a special subclass of `MWArray`, lets you create a MATLAB array that references .NET objects.

You can create a MATLAB code wrapper around .NET objects using `MWObjectArray`. Use this technique to pass objects by reference to MATLAB functions and return .NET objects. The examples in this section present some common use cases.

Pass .NET Object into .NET Assembly

To pass an object into a MATLAB Compiler SDK assembly:

- 1 Write the MATLAB function that references a .NET type.

```
function addItem(hDictionary, key, value)

    if ~isa(hDictionary, 'System.Collections.Generic.IDictionary')
        error('foo:IncorrectType',
            ... 'expecting a System.Collections.Generic.Dictionary');
    end

    hDictionary.Add(key, value);

end
```

- 2 Create a .NET object to pass to the MATLAB function.

```
Dictionary char2Ascii= new Dictionary();
char2Ascii.Add("A", 65);
char2Ascii.Add("B", 66);
```

- 3 Create an instance of MWObjectArray to wrap the .NET object.

```
MWObjectArray MWchar2Ascii=
    new MWObjectArray(char2Ascii);
```

- 4 Pass the wrapped object to the MATLAB function.

```
myComp.addItem(MWchar2Ascii, 'C', 67);
```

Return Custom .NET Object in MATLAB Function Using Deployed .NET Assembly

You can use MWObjectArray to clone an object inside a MATLAB Compiler SDK .NET assembly. Continuing with the previous example, perform the following steps:

- 1 Write the MATLAB function that references a .NET type.

```
function result= add(hMyDouble, value)

    if ~isa(hMyDouble, 'MyDoubleComp.MyDouble')
        error('foo:IncorrectType', 'expecting a MyDoubleComp.MyDouble');
    end
    hMyDoubleClone= hMyDouble.Clone();
    result= hMyDoubleClone.Add(value);

end
```

- 2 Create the object.

```
MyDouble myDouble= new MyDouble(75);
```

- 3 Create an instance of MWObjectArray to wrap the .NET object.

```
MWObjectArray MWdouble= new MWObjectArray(myDouble);
origRef = new MWObjectArray(hash);
```

- 4 Pass the wrapped object to the MATLAB function and retrieve the returned cloned object.

```
MWObjectArray result=
    (MWObjectArray)myComp.add(MWdouble, 25);
```

- 5 Unwrap the .NET object and print the result.

```
MyDouble doubleClone= (MyDouble)result.Object;

Console.WriteLine(myDouble.ToDouble());
Console.WriteLine(doubleClone.ToDouble());
```

Clone MWObjectArray

When calling the `Clone` method on `MWObjectArray`, the following rules apply for the wrapped object:

- If the wrapped object is a `ValueType`, it is deep-copied.
- If an object is not a `ValueType` and implements `ICloneable`, the `Clone` method for the object is called.
- The `MemberwiseClone` method is called on the wrapped object.

```
MWObjectArray aDate = new MWObjectArray(new  
    DateTime(1, 1, 2010));  
MWObjectArray clonedDate = aDate.Clone();
```

Optimization Using MWObjectArray

For a full example of how to use `MWObjectArray` to create a reference to a .NET object and pass it to a component, see “Integrate MATLAB Optimization Routines with Objective Functions” on page 3-35.

MWObjectArray and Application Domains

Every ASP .NET web application deployed to IIS is started in a separate `AppDomain`.

The MATLAB .NET interface must support the .NET type wrapped by `MWObjectArray`. If the `MWObjectArray` is created in the default `AppDomain`, the wrapped type has no other restrictions.

If the `MWObjectArray` is not created in the default `AppDomain`, the wrapped .NET type must be serializable. This limitation is imposed by the fact that the object needs to be marshaled from the non-default `AppDomain` to the default `AppDomain` in order for MATLAB to access it.

MWObjectArray Limitation

If you have any global objects in your C# code, then you will get a Windows exception on exiting the application. To overcome this limitation, use one of these solutions:

- Explicitly clear global objects before exiting the application.
- Call `TerminateApplicationEx` method before exiting the application.

```
MWMCR.TerminateApplicationEx();
```

For more information on `TerminateApplicationEx`, see the `MWArray` Class Library Reference.

Access Real or Imaginary Components Within Complex Arrays

Component Extraction

When you access a complex array (an array made up of both real and imaginary data), you extract both real and imaginary parts (called components) by default. This method call extracts both real and imaginary components:

```
MWNumericArray complexResult= complexDouble[1, 2];
```

It is also possible to extract only the real or imaginary component of a complex matrix by calling the appropriate component indexing method.

Component Indexing on Complex Numeric Arrays

To return the real or imaginary component from a full complex numeric array, call the `.real` or `.imaginary` method on `MWArrayComponent`.

```
complexResult= complexDouble[MWArrayComponent.Real, 1, 2];
complexResult= complexDouble[MWArrayComponent.Imaginary, 1, 2];
```

To assign the real or imaginary component to a full complex numeric array, call the `.real` or `.imaginary` method on `MWArrayComponent`.

```
matrix[MWArrayComponent.Real, 2, 2]= 5;
matrix[MWArrayComponent.Imaginary, 2, 2]= 7;
```

You can return the real or imaginary component from a sparse complex numeric array in Microsoft Visual Studio 8 and later.

```
complexResult= sparseComplexDouble[MWArrayComponent.Real, 4, 3];
complexResult = sparseComplexDouble[MWArrayComponent.Imaginary, 4, 3];
```

Convert MATLAB Arrays to .NET Arrays

To convert MATLAB arrays to .NET arrays call the `toArray` method with either the `.real` or `.imaginary` method on `MWArrayComponent`.

```
Array nativeArray_real= matrix.ToArray(MWArrayComponent.Real);
Array nativeArray_imag= matrix.ToArray(MWArrayComponent.Imaginary);
```

Convert MATLAB Arrays to .NET Vectors

To convert MATLAB vectors to .NET vectors (single dimension arrays) call the `.real` or `.imaginary` method on `MWArrayComponent`.

```
Array nativeArray= sparseMatrix.ToVector(MWArrayComponent.Real);
Array nativeArray= sparseMatrix.ToVector(MWArrayComponent.Imaginary);
```

Jagged Array Processing

A jagged array is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes, as opposed to the elements of a non-jagged array whose elements are of the same dimensions and size.

Web services, in particular, process data almost exclusively in jagged arrays.

`MWNumericArrays` can only process jagged arrays with a rectangular shape.

In the following code snippet, a rectangular jagged array of type `int` is initialized and populated.

```
int[][] jagged = new int[5][];
for (int i = 0; i < 5; i++)
    jagged[i] = new int[10];
MWNumericArray jaggedMWArray = new MWNumericArray(jagged);
Console.WriteLine(jaggedMWArray);
```

See Also

Related Examples

- “Data Marshaling with MWArray API” on page 2-6
- “Rules for Data Conversion Between .NET and MATLAB” on page 11-2

Block Console Display When Creating Figures

In this section...

“WaitForFiguresToDie Method” on page 2-19

“Using WaitForFiguresToDie to Block Execution” on page 2-19

WaitForFiguresToDie Method

The MATLAB Compiler SDK product adds a `WaitForFiguresToDie` method to each .NET class that it creates.

The purpose of `WaitForFiguresToDie` is to block execution of a calling program as long as figures created in encapsulated MATLAB code are displayed. Typically you use `WaitForFiguresToDie` when:

- There are one or more figures open that were created by a .NET assembly created by the MATLAB Compiler SDK product.
- The method that displays the graphics requires user input before continuing.
- The method that calls the figures was called from `main()` in a console program.

When `WaitForFiguresToDie` is called, execution of the calling program is blocked if any figures created by the calling object remain open.

`WaitForFiguresToDie` takes no arguments. Your application can call `WaitForFiguresToDie` any time during execution.

Tip Consider using the `console.readline` method when possible, as it accomplishes much of this functionality in a standardized manner.

Caution Calling `WaitForFiguresToDie` from an interactive program can make the application stop responding. This method should be called *only* from console-based programs.

Using WaitForFiguresToDie to Block Execution

The following example illustrates using `WaitForFiguresToDie` from a .NET application. The example uses a .NET assembly created by the MATLAB Compiler SDK product. The component encapsulates MATLAB code that draws a simple plot.

- 1 Create a work folder for your source code. In this example, the folder is `D:\work\plotdemo`.
- 2 In this folder, create the following MATLAB file named `drawplot.m`:

```
function drawplot()
plot(1:10);
```

- 3 Build the .NET component with the **Library Compiler** app or `compiler.build.dotNETAssembly` using the following information:

Field	Value
Library Name	Figure

Field	Value
Class Name	Plotter
File to Compile	drawplot.m

For example, if you are using `compiler.build.dotNETAssembly`, type:

```
buildResults = compiler.build.dotNETAssembly('drawplot.m', ...  
'AssemblyName','Figure', ...  
'ClassName','Plotter');
```

For more details, see the instructions in “Generate .NET Assembly and Build .NET Application”.

- 4 In Visual Studio, create a C# **Console App (.NET Framework)**. Replace the generated source code with the following code:

```
using Figure.Plotter;  
  
public class Main  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            plotter p = new Plotter();  
            try  
            {  
                p.drawplot();  
                p.WaitForFiguresToDie();  
            }  
            catch (Exception e)  
            {  
                console.WriteLine(e);  
            }  
        }  
    }  
}
```

- 5 Add a reference to your generated assembly file `Figure.dll`.
- 6 Compile and run the application.

The program displays a plot from 1 to 10 in a MATLAB figure window. The application ends when you close the figure.

Note To see what happens without the call to `WaitForFiguresToDie`, comment out the call, rebuild the application, and run it. In this case, the figure is drawn and is immediately destroyed as the application exits.

See Also

`libraryCompiler` | `compiler.build.dotNETAssembly` | `deploytool`

Related Examples

- “Generate .NET Assembly and Build .NET Application”
- “Integrate Simple MATLAB Function Into .NET Application” on page 3-2

Error Handling and Resource Management

When creating the .NET application, it is a good practice to properly handle run-time errors and manage resources.

Error Handling

As with managed code, any errors that occur during execution of a MATLAB function or during data conversion are signaled by a standard .NET exception.

Like any other .NET application, an application that calls a method generated by MATLAB Compiler SDK can handle errors by either catching and handling the exception locally or allowing the calling method to catch it.

Here are examples for each way of error handling.

In the `GetPrimes` example, the method itself handles the exception.

```
public double[] GetPrimes(int n)
{
    MWArray primes= null;
    MyPrimesClass myPrimesClass= null;
    try
    {
        myPrimesClass= new MyPrimesClass();
        primes= myPrimesClass.myprimes((double)n);
        return ((double[])(MWNumericArray)primes).
            ToVector(MWArrayComponent.Real);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Exception: {0}", ex);
        return new double[0];
    }
}
```

In the next example, the method that calls `myprimes` does not catch the exception. Instead, its calling method (that is, the method that calls the method that calls `myprimes`) handles the exception.

```
public double[] GetPrimes(int n)
{
    MWArray primes= null;
    MyPrimesClass myPrimesClass= null;
    try
    {
        myPrimesClass= new MyPrimesClass();
        primes= myPrimesClass.myprimes((double)n);
        return ((double[])(MWNumericArray)primes).
            ToVector(MWArrayComponent.Real);
    }
    catch (Exception e)
    {
        throw;
    }
}
```

```
    }  
}
```

Freeing Resources Explicitly

Typically, the `Dispose` method is called from a `finally` section in a `try-finally` block.

```
try  
{  
    /* Allocate a huge array */  
    MWNumericArray array = new MWNumericArray(1000,1000);  
    .  
    . (use the array)  
    .  
}  
finally  
{  
    /* Explicitly dispose of the managed array and its */  
    /* native resources */  
    if (null != array)  
    {  
        array.Dispose();  
    }  
}
```

The statement `array.Dispose()` frees the memory allocated by both the managed wrapper and the native MATLAB array.

The `MWArray` class provides two disposal methods: `Dispose` and the static method `DisposeArray`. The `DisposeArray` method is more general in that it disposes of either a single `MWArray` or an array of arrays of type `MWArray`.

Limitations on Multiple Assemblies in Single Application

In this section...

“Work with MATLAB Function Handles” on page 2-23

“Work with Objects” on page 2-24

When developing applications that use multiple MATLAB .NET assemblies, consider that the following cannot be shared between assemblies:

- MATLAB function handles
- MATLAB figure handles
- MATLAB objects
- C, Java®, and .NET objects
- Executable data stored in cell arrays and structures

Work with MATLAB Function Handles

MATLAB function handles can be passed between an application and the MATLAB Runtime instance from which it originated. However, a MATLAB function handle cannot be passed into a MATLAB Runtime instance other than the one in which it originated. For example, suppose you had two MATLAB functions, `get_plot_handle` and `plot_xy`, and `plot_xy` used the function handle created by `get_plot_handle`.

`get_plot_handle.m`

```
function h = get_plot_handle(lnSpec, lnWidth, mkEdge, mkFace, mkSize)
h = @draw_plot;
    function draw_plot(x, y)
        plot(x, y, lnSpec, ...
            'LineWidth', lnWidth, ...
            'MarkerEdgeColor', mkEdge, ...
            'MarkerFaceColor', mkFace, ...
            'MarkerSize', mkSize)
    end
end
```

`plot_xy.m`

```
function plot_xy(x, y, h)
h(x, y);
end
```

If you compiled them into two shared libraries, the call to `plot_xy` would throw an exception.

```
using System;

using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;

using get_plot_handle;
using plot_xy;

namespace MathWorks.Examples.PlotApp
{
    class PlotCSApp
    {
        static void Main(string[] args)
```

```
{
  try
  {
    // Create objects for the generated functions
    get_plot_handle.Class1 plotter=
      new get_plot_handle.Class1();
    plot_xy.Class1 plot = new plot_xy.Class1();

    MWArray h = plotter.get_plot_handle('--rs', (double)2,
      'k','g', (double)10);

    double[] x_data = {1,2,3,4,5,6,7,8,9};
    double[] y_data = {2,6,12,20,30,42,56,72,90};
    MWArray x = new MWArray(x_data);
    MWArray y = new MWArray(y_data);
    plot.plot_xy(x, y, h);
  }

  catch(Exception exception)
  {
    Console.WriteLine("Error: {0}", exception);
  }
}
}
```

The correct way to handle the situation is to compile both functions into a single assembly.

```
using System;

using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;

using plot_functions;

namespace MathWorks.Examples.PlotApp
{
  class PlotCSApp
  {
    static void Main(string[] args)
    {
      try
      {
        // Create object for the generated functions
        Class1 plot= new Class1();

        MWArray h = plot.get_plot_handle('--rs', (double)2,
          'k','g', (double)10);

        double[] x_data = {1,2,3,4,5,6,7,8,9};
        double[] y_data = {2,6,12,20,30,42,56,72,90};
        MWArray x = new MWArray(x_data);
        MWArray y = new MWArray(y_data);
        plot.plot_xy(x, y, h);
      }

      catch(Exception exception)
      {
        Console.WriteLine("Error: {0}", exception);
      }
    }
  }
}
```

Work with Objects

MATLAB Compiler SDK enables you to return the following types of objects from the MATLAB Runtime to your application code:

- MATLAB
- C++
- .NET

- Java

However, you cannot pass an object created in one MATLAB Runtime instance into a different MATLAB Runtime instance. This conflict can happen when a function that returns an object and a function that manipulates that object are compiled into different assemblies.

For example, you develop a bank account class and two functions. The first creates a bank account for a customer based on some set of conditions. The second transfers funds between two accounts.

account.m

```
% Saved as account.m
classdef account < handle

    properties
        name
    end

    properties (SetAccess = protected)
        balance = 0
        number
    end

    methods
        function obj = account(name)
            obj.name = name;
            obj.number = round(rand * 1000);
        end

        function deposit(obj, deposit)
            new_bal = obj.balance + deposit;
            obj.balance = new_bal;
        end

        function withdraw(obj, withdrawl)
            new_bal = obj.balance - withdrawl;
            obj.balance = new_bal;
        end
    end
end
```

open_acct .m

```
% Saved as open_acct .m
function acct = open_acct(name, open_bal )

    acct = account(name);

    if open_bal > 0
        acct.deposit(open_bal);
    end

end
```

transfer.m

```
% Saved as transfer.m
function transfer(source, dest, amount)
```

```
if (source.balance > amount)
    dest.deposit(amount);
    source.withdraw(amount);
end
```

```
end
```

If you compiled `open_acct.m` and `transfer.m` into separate assemblies, you could not transfer funds using accounts created with `open_acct`. The call to `transfer` throws an exception.

One way of resolving this is to compile both functions into a single assembly. You could also refactor the application such that you are not passing MATLAB objects to the functions.

C# Integration Examples

- “Integrate Simple MATLAB Function Into .NET Application” on page 3-2
- “Integrate Function with Variable Number of Arguments” on page 3-13
- “Use Multiple Classes in .NET Assembly” on page 3-19
- “Assign Multiple MATLAB Functions in Component Class” on page 3-27
- “Integrate MATLAB Optimization Routines with Objective Functions” on page 3-35
- “Build .NET Core Application That Runs on Linux and macOS” on page 3-40

Integrate Simple MATLAB Function Into .NET Application

In this section...

“Prerequisites” on page 3-2

“Create Simple Plot” on page 3-2

“Create Phone Book” on page 3-7

Note The examples for the MATLAB Compiler SDK product are in `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion`, where *VSVersion* specifies the version of Microsoft Visual Studio .NET you are using. You can load projects for all the examples by opening the following solution in Visual Studio:

```
matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\DotNetExamples.sln
```

The Simple Plot on page 3-2 example shows you how to create a .NET assembly that calls a MATLAB function to display a plot. For an example that uses a MATLAB function to modify a structure array, see Phone Book on page 3-7.

In the following examples, you perform these steps to integrate a MATLAB function into a .NET application:

- Use the MATLAB Compiler SDK product to convert a MATLAB function to a method of a .NET class and wrap the class in a .NET assembly.
- Access the component in either a C# application or a Visual Basic application by instantiating your .NET class and using the `MWArray` class library to handle data conversion.
- Build and run the generated application using the Visual Studio .NET development environment.

Prerequisites

- Verify that you have met all of the MATLAB Compiler SDK .NET target requirements. For details, see “MATLAB Compiler SDK .NET Target Requirements” on page 1-2.
- Verify that you have Microsoft Visual Studio installed.
- End users must have an installation of MATLAB Runtime to run the application. For details, see “Install and Configure MATLAB Runtime”.

For testing purposes, you can use an installation of MATLAB instead of MATLAB Runtime.

Create Simple Plot

Files

MATLAB Function Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\PlotExample\PlotComp\drawgraph.m</code>
C# Code Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\PlotExample\PlotCSApp\PlotApp.cs</code>

Visual Basic Code Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\PlotExample\PlotVBAApp\PlotApp.vb</code>
----------------------------	--

Procedure

- 1 Copy the following folder that ships with the MATLAB product to your work folder:

```
matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\PlotExample
```

At the MATLAB command prompt, navigate to the `PlotExample\PlotComp` subfolder in your work folder.

- 2 Examine the `drawgraph` function located in `PlotExample\PlotComp`.

```
function drawgraph(coords)
plot(coords(1,:), coords(2,:));
pause(5)
```

Test the function at the MATLAB command prompt.

```
x = 0:0.01:10;
y = sin(x);
z = [x;y];
drawgraph(z)
```

The function outputs a figure that displays a sine wave.

- 3 Build the .NET component with the **Library Compiler** app or compiler.`build.dotNETAssembly` using the following information:

Field	Value
Library Name	PlotComp
Class Name	Plotter
File to Compile	drawgraph.m

For example, if you are using `compiler.build.dotNETAssembly`, type:

```
buildResults = compiler.build.dotNETAssembly('drawgraph.m', ...
'AssemblyName','PlotComp', ...
'ClassName','Plotter');
```

For more details, see the instructions in “Generate .NET Assembly and Build .NET Application”.

- 4 Decide whether you are using C# or Visual Basic to access the component.

- **C#**

If you are using C#, write source code for a C# application that accesses the component.

The sample application for this example is in `PlotExample\PlotCSApp\PlotApp.cs`.

PlotApp.cs

```
using System;

using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;

using PlotComp;

namespace MathWorks.Examples.PlotApp
{
    /// <summary>
    /// This application demonstrates plotting x-y data by graphing a simple
    /// parabola into a MATLAB figure window.

```

```
/// </summary>
class PlotCSApp
{
    #region MAIN

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
        try
        {
            const int numPoints= 10; // Number of points to plot

            // Allocate native array for plot values
            double [,] plotValues= new double[2, numPoints];

            // Plot 5x vs x^2
            for (int x= 1; x <= numPoints; x++)
            {
                plotValues[0, x-1]= x*5;
                plotValues[1, x-1]= x*x;
            }

            // Create a new plotter object
            Plotter plotter= new Plotter();

            // Plot the two sets of values - Note the ability to cast
            // the native array to a MATLAB numeric array
            plotter.drawgraph((MWNumericArray)plotValues);

            Console.ReadLine(); // Wait for user to exit application
        }
        catch(Exception exception)
        {
            Console.WriteLine("Error: {0}", exception);
        }
    }
}
#endregion
}
```

This statement creates an instance of the `Plotter` class:

```
Plotter plotter= new Plotter();
```

This statement explicitly casts the native `plotValues` to `MWNumericArray` and then calls the method `drawgraph`:

```
plotter.drawgraph((MWNumericArray)plotValues);
```

- **Visual Basic**

If you are using Visual Basic, write source code for a Visual Basic application that accesses the component.

The sample application for this example is in `PlotExample\PlotVBApp\PlotApp.vb`.

PlotApp.vb

```
Imports System
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays
Imports PlotComp
```

```

Namespace MathWorks.Examples.PlotApp
    ' <summary>
    ' This application demonstrates plotting x-y data by graphing a simple
    ' parabola into a MATLAB figure window.
    ' </summary>
    Class PlotDemoApp
#Region " MAIN "
        ' <summary>
        ' The main entry point for the application.
        ' </summary>
        Shared Sub Main(ByVal args() As String)
            Try
                Const numPoints As Integer = 10 ' Number of points to plot
                Dim idx As Integer
                Dim plotValues(,) As Double = New Double(1, numPoints - 1) {}
                Dim coords As MWNumericArray

                'Plot 5x vs x^2
                For idx = 0 To numPoints - 1
                    Dim x As Double = idx + 1
                    plotValues(0, idx) = x * 5
                    plotValues(1, idx) = x * x
                Next idx

                coords = New MWNumericArray(plotValues)

                ' Create a new plotter object
                Dim plotter As Plotter = New Plotter

                ' Plot the values
                plotter.drawgraph(coords)

                Console.ReadLine() ' Wait for user to exit application

            Catch exception As Exception
                Console.WriteLine("Error: {0}", exception)
            End Try
        End Sub
#End Region
    End Class
End Namespace

```

This statement creates an instance of the `Plotter` class:

```
Dim plotter As Plotter = New Plotter
```

This statement calls the method `drawgraph`:

```
plotter.drawgraph(coords)
```

In either case, the `PlotApp` program does the following:

- Creates two arrays of double values.
- Creates a `Plotter` object.
- Calls the `drawgraph` method to plot the equation using the MATLAB `plot` function.
- Uses `MWNumericArray` to represent the data needed by the `drawgraph` method to plot the equation.
- Uses a `try-catch` block to catch and handle any exceptions.

5 Open the .NET project file that corresponds to your application language using Visual Studio.

- **C#**

If you are using C#, the `PlotCSApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `PlotCSApp.csproj` in Windows Explorer. You can also open it from the desktop by right-clicking **PlotCSApp.csproj** and selecting **Open Outside MATLAB**.

- **Visual Basic**

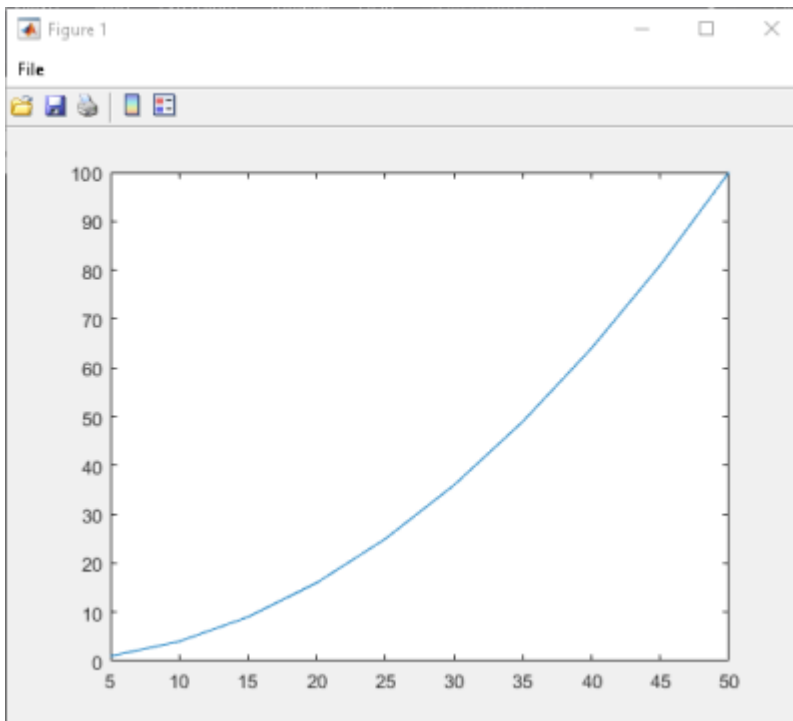
If you are using Visual Basic, the `PlotVBApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `PlotVBApp.vbproj` in Windows Explorer. You can also open it from the desktop by right-clicking **PlotVBApp.vbproj** and selecting **Open Outside MATLAB**.

- 6 Add a reference to your assembly file `PlotComp.dll` located in the folder where you generated or installed the assembly.
- 7 Add a reference to the `MWArray` API.

If MATLAB is installed on your system	<code>matlabroot\toolbox\dotnetbuilder\bin\win64\<version>\MWArray.dll</code>
If MATLAB Runtime is installed on your system	<code><MATLAB_RUNTIME_INSTALL_DIR>\toolbox\dotnetbuilder\bin\win64\<version>\MWArray.dll</code>

- 8 Build and run the `PlotApp` application in Visual Studio .NET.

The application displays the following plot:



- 9 To follow up on this example:
 - Try running the generated application on a different computer.
 - Try building an installer for the package using `compiler.package.installer`.
 - Try integrating an assembly that consists of multiple functions.

Create Phone Book

In this example, you create a .NET assembly that calls a MATLAB function to modify a structure array that contains phone numbers.

Files

MATLAB Function Location	<i>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\PhoneBookExample\PhoneBookComp\makephone.m</i>
C# Code Location	<i>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\PhoneBookExample\PhoneBookCSApp\PhoneBookApp.cs</i>
Visual Basic Code Location	<i>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\PhoneBookExample\PhoneBookVBApp\PhoneBookApp.vb</i>

Procedure

- 1 Copy the following folder that ships with MATLAB to your work folder:

matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\PhoneBookExample

At the MATLAB command prompt, navigate to the PhoneBookExample\PhoneBookComp subfolder in your work folder.

- 2 Examine the makephone function located in PhoneBookExample\PhoneBookComp.

```
function book = makephone(friends)
book = friends;
for i = 1:numel(friends)
    numberStr = num2str(book(i).phone);
    book(i).external = ['(508) 555-' numberStr];
end
```

Test the function at the MATLAB command prompt.

```
friends(1).name = "Jordan Robert";
friends(1).phone = 3386;
friends(2).name = "Mary Smith";
friends(2).phone = 3912;
struct2table(makephone(friends))
```

ans =

2×3 table

name	phone	external
"Jordan Robert"	3386	{ '(508) 555-3386' }
"Mary Smith"	3912	{ '(508) 555-3912' }

- 3 Build the .NET component with the **Library Compiler** app or `compiler.build.dotNETAssembly` using the following information:

Field	Value
Library Name	PhoneBookComp

Field	Value
Class Name	PhoneBook
File to Compile	makephone

For example, if you are using `compiler.build.dotNETAssembly`, type:

```
buildResults = compiler.build.dotNETAssembly('makephone.m', ...
'AssemblyName', 'PhoneBookComp', ...
'ClassName', 'PhoneBook');
```

For more details, see the instructions in “Generate .NET Assembly and Build .NET Application”.

4 Decide whether you are using C# or Visual Basic to access the component.

- **C#**

If you are using C#, write source code for a C# application that accesses the component.

The sample application for this example is in
PhoneBookExample\PhoneBookCSApp\PhoneBookApp.cs.

PhoneBookApp.cs

```
/* Necessary package imports */
using System;
using System.Collections.Generic;
using System.Text;
using MathWorks.MATLAB.NET.Arrays;
using PhoneBookComp;

namespace MathWorks.Examples.PhoneBookApp
{
    //
    // This class demonstrates the use of the MWStructArray class
    //
    class PhoneBookApp
    {
        static void Main(string[] args)
        {
            PhoneBook thePhonebook = null; /* Stores deployment class instance */
            MWStructArray friends= null; /* Sample input data */
            MWArray[] result= null; /* Stores the result */
            MWStructArray book= null; /* Output data extracted from result */

            /* Create the new deployment object */
            thePhonebook= new PhoneBook();

            /* Create an MWStructArray with two fields */
            String[] myFieldNames= { "name", "phone" };
            friends= new MWStructArray(2, 2, myFieldNames);

            /* Populate struct with some sample data --- friends and phone */
            /* number extensions */
            friends["name", 1]= new MWCharArray("Jordan Robert");
            friends["phone", 1]= 3386;
            friends["name", 2]= new MWCharArray("Mary Smith");
            friends["phone", 2]= 3912;
            friends["name", 3]= new MWCharArray("Stacy Flora");
            friends["phone", 3]= 3238;
            friends["name", 4]= new MWCharArray("Harry Alpert");
            friends["phone", 4]= 3077;

            /* Show some of the sample data */
            Console.WriteLine("Friends: ");
            Console.WriteLine(friends.ToString());

            /* Pass it to a MATLAB function that determines external phone number */
            result= thePhonebook.makephone(1, friends);
            book= (MWStructArray)result[0];

            Console.WriteLine("Result: ");
            Console.WriteLine(book.ToString());
        }
    }
}
```

```

/* Extract some data from the returned structure */
Console.WriteLine("Result record 2:");

Console.WriteLine(book["name", 2]);
Console.WriteLine(book["phone", 2]);
Console.WriteLine(book["external", 2]);

/* Print the entire result structure using the helper function below */
Console.WriteLine("");
Console.WriteLine("Entire structure:");

DispStruct(book);

Console.ReadLine();
}

public static void DispStruct(MWStructArray arr)
{
    Console.WriteLine("Number of Elements: " + arr.NumberOfElements);

    int[] dims= arr.Dimensions;

    Console.WriteLine("Dimensions: " + dims[0]);

    for (int idx= 1; idx < dims.Length; idx++)
    {
        Console.WriteLine("-by-" + dims[idx]);
    }

    Console.WriteLine("\nNumber of Fields: " + arr.NumberOfFields);
    Console.WriteLine("Standard MATLAB view:");
    Console.WriteLine(arr.ToString());

    Console.WriteLine("Walking structure:");

    string[] fieldNames= arr.FieldNames;

    for (int element= 1; element <= arr.NumberOfElements; element++)
    {
        Console.WriteLine("Element " + element);

        for (int field= 0; field < arr.NumberOfFields; field++)
        {
            MWArray fieldVal= arr[arr.FieldNames[field], element];

            /* Recursively print substructures, */
            /* give string display of other classes */
            if (fieldVal.GetType() == typeof(MWStructArray))
            {
                Console.WriteLine("    " + fieldNames[field] + ":
                    nested structure:");
                Console.WriteLine("+++ Begin of \" + fieldNames[field] + \"\
                    nested structure");
                DispStruct((MWStructArray)fieldVal);
                Console.WriteLine("+++ End of \" + fieldNames[field] +
                    \"\ nested structure");
            }
            else
            {
                Console.WriteLine("    " + fieldNames[field] + ": ");
                Console.WriteLine(fieldVal.ToString());
            }
        }
    }
}
}
}
}

```

- **Visual Basic**

If you are using Visual Basic, write source code for a Visual Basic application that accesses the component.

The sample application for this example is in
 \PhoneBookExample\PhoneBookVBApp\PhoneBookApp.vb.

PhoneBookApp.vb

```

' Necessary package imports

Imports MathWorks.MATLAB.NET.Arrays
Imports PhoneBookComp

'
' getphone class demonstrates the use of the MWStructArray class

Public Module PhoneBookVbApp
    Public Sub Main()
        Dim thePhonebook As phonebook 'Stores deployment class instance
        Dim friends As MWStructArray 'Sample input data
        Dim result As Object() 'Stores the result
        Dim book As MWStructArray 'Output data extracted from result

        ' Create the new deployment object
        thePhonebook = New phonebook()

        ' Create an MWStructArray with two fields
        Dim myFieldNames As String() = {"name", "phone"}
        friends = New MWStructArray(2, 2, myFieldNames)

        ' Populate struct with some sample data --- friends and phone numbers
        friends("name", 1) = New MWCharArray("Jordan Robert")
        friends("phone", 1) = 3386
        friends("name", 2) = New MWCharArray("Mary Smith")
        friends("phone", 2) = 3912
        friends("name", 3) = New MWCharArray("Stacy Flora")
        friends("phone", 3) = 3238
        friends("name", 4) = New MWCharArray("Harry Alpert")
        friends("phone", 4) = 3077

        ' Show some of the sample data
        Console.WriteLine("Friends: ")
        Console.WriteLine(friends.ToString())

        ' Pass it to a MATLAB function that determines external phone number
        result = thePhonebook.makephone(1, friends)
        book = CType(result(0), MWStructArray)
        Console.WriteLine("Result: ")
        Console.WriteLine(book.ToString())

        ' Extract some data from the returned structure '
        Console.WriteLine("Result record 2:")

        Console.WriteLine(book("name", 2))
        Console.WriteLine(book("phone", 2))
        Console.WriteLine(book("external", 2))

        ' Print the entire result structure using the helper function below
        Console.WriteLine("")
        Console.WriteLine("Entire structure:")
        dispStruct(book)
    End Sub

    Sub dispStruct(ByVal arr As MWStructArray)
        Console.WriteLine("Number of Elements: " + arr.NumberOfElements.ToString())
        'int numDims = arr.NumberofDimensions
        Dim dims As Integer() = arr.Dimensions
        Console.WriteLine("Dimensions: " + dims(0).ToString())

        Dim i As Integer
        For i = 1 To dims.Length
            Console.WriteLine("-by-" + dims(i - 1).ToString())
        Next i
        Console.WriteLine("")
        Console.WriteLine("Number of Fields: " + arr.NumberOfFields.ToString())
        Console.WriteLine("Standard MATLAB view:")
        Console.WriteLine(arr.ToString())
        Console.WriteLine("Walking structure:")

        Dim fieldNames As String() = arr.FieldNames

        Dim element As Integer
        For element = 1 To arr.NumberOfElements
            Console.WriteLine("Element " + element.ToString())
            Dim field As Integer
            For field = 0 To arr.NumberOfFields - 1

```



```

Dim fieldVal As MArray = arr(arr.FieldName(field), element)
' Recursively print substructures, give string display of other classes
If (TypeOf fieldVal Is MStructArray) Then
    Console.WriteLine(" " + fieldNames(field) + ": nested structure:")
    Console.WriteLine("+++ Begin of \" + fieldNames[field] +
        " \" nested structure")

    dispStruct(CType(fieldVal, MStructArray))
    Console.WriteLine("+++ End of \" + fieldNames[field] +
        " \" nested structure")
Else
    Console.WriteLine(" " + fieldNames(field) + ": ")
    Console.WriteLine(fieldVal.ToString())
End If
Next field
Next element
End Sub
End Module

```

In either case, the PhoneBookApp program does the following:

- Creates a structure array using MStructArray to represent the example phonebook data containing names and phone numbers.
 - Instantiates the Phonebook class as thePhonebook object, as shown:
`thePhonebook = new phonebook();`
 - Calls the MATLAB function makephone to create a modified copy of the structure by adding an additional field, as shown:
`result = thePhonebook.makephone(1, friends);`
 - Displays the resulting structure array.
- 5 Open the .NET project file that corresponds to your application language using Visual Studio.

- **C#**

If you are using C#, the PhoneBookCSApp folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking PhoneBookCSApp.csproj in Windows Explorer. You can also open it from the desktop by right-clicking **PhoneBookCSApp.csproj** and selecting **Open Outside MATLAB**.

- **Visual Basic**

If you are using Visual Basic, the PhoneBookVBApp folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking PhoneBookVBApp.vbproj in Windows Explorer. You can also open it from the desktop by right-clicking **PhoneBookVBApp.vbproj** and selecting **Open Outside MATLAB**.

- 6 Create a reference to your assembly file PhoneBookComp.dll located in the folder where you generated the assembly.
- 7 Create a reference to the MArray API, which is located in:

MATLAB	<i>matlabroot</i> \toolbox\dotnetbuilder\bin\win64\ <i><version></i> \MArray.dll
MATLAB Runtime	<i><MATLAB_RUNTIME_INSTALL_DIR></i> \toolbox\dotnetbuilder\bin\win64\ <i><version></i> \MArray.dll

- 8 Build and run the PhoneBookComp application in Visual Studio .NET.

The application displays the following output:

```

Friends:
2x2 struct array with fields:
    name
    phone

```

```
Result:
2x2 struct array with fields:
    name
    phone
    external
Result record 2:
Mary Smith
3912
(508) 555-3912
```

```
Entire structure:
Number of Elements: 4
Dimensions: 2-by-2
Number of Fields: 3
Standard MATLAB view:
2x2 struct array with fields:
    name
    phone
    external
```

```
Walking structure:
Element 1
    name: Jordan Robert
    phone: 3386
    external: (508) 555-3386
Element 2
    name: Mary Smith
    phone: 3912
    external: (508) 555-3912
Element 3
    name: Stacy Flora
    phone: 3238
    external: (508) 555-3238
Element 4
    name: Harry Alpert
    phone: 3077
    external: (508) 555-3077
```

See Also

[libraryCompiler](#) | [compiler.build.dotNETAssembly](#) | [deploytool](#)

Related Examples

- “Generate .NET Assembly and Build .NET Application”
- “Integrate Function with Variable Number of Arguments” on page 3-13

Integrate Function with Variable Number of Arguments

This example shows you how to create a .NET application using a MATLAB function that takes a variable number of arguments instead of just one.

In this example, you perform the following steps:

- Use the MATLAB Compiler SDK product to convert the MATLAB function `drawgraph` to a method of a .NET class (`Plotter`) and wrap the class in a .NET assembly (`VarArgComp`). The `drawgraph` function displays a plot of the input parameters and is called as a method of the `Plotter` class.
- Access the component in a C# application (`VarArgApp.cs`) or a Visual Basic application (`VarArgApp.vb`) by instantiating the `Plotter` class and using `MWArray` to represent data.
- Build and run the `VarArgDemoApp` application using the Visual Studio .NET development environment.

Files

MATLAB Functions	<code>drawgraph.m</code> <code>extractcoords.m</code>
MATLAB Function Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\VarArgExample\VarArgComp\</code>
C# Code Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\VarArgExample\VarArgCSApp\VarArgApp.cs</code>
Visual Basic Code Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\VarArgExample\VarArgVBApp\VarArgApp.vb</code>

Procedure

- 1 Copy the following folder that ships with the MATLAB product to your work folder:

```
matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\VarArgExample
```

At the MATLAB command prompt, navigate to the new `VarArgExample\VarArgComp` subfolder in your work folder.

- 2 Examine the `drawgraph` and `extractcoords` functions.

```
function [xyCoords] = DrawGraph(colorSpec, varargin)

    numVarArgIn= length(varargin);
    xyCoords= zeros(numVarArgIn, 2);

    for idx = 1:numVarArgIn
        xCoord = varargin{idx}(1);
        yCoord = varargin{idx}(2);

        x(idx) = xCoord;
        y(idx) = yCoord;

        xyCoords(idx,1) = xCoord;
        xyCoords(idx,2) = yCoord;
    end

    xmin = min(0, min(x));
    ymin = min(0, min(y));

    axis([xmin fix(max(x))+3 ymin fix(max(y))+3])

    plot(x, y, 'color', colorSpec);
```

```
function [varargout] = ExtractCoords(coords)
    for idx = 1:nargout
        varargout{idx}= coords(idx,:);
    end
end
```

- 3 Build the .NET component with the **Library Compiler** app or `compiler.build.dotNETAssembly` using the following information:

Field	Value
Library Name	VarArgComp
Class Name	Plotter
Files to Compile	extractcoords.m drawgraph.m

For example, if you are using `compiler.build.dotNETAssembly`, type:

```
buildResults = compiler.build.dotNETAssembly(["extractcoords.m","drawgraph.m"], ...
    'AssemblyName','VarArgComp', ...
    'ClassName','Plotter');
```

For more details, see the instructions in “Generate .NET Assembly and Build .NET Application”.

- 4 Decide whether you are using C# or Visual Basic to access the component.

- **C#**

If you are using C#, write source code for a C# application that accesses the component.

The sample application for this example is in `VarArgExample\VarArgCSApp\VarArgApp.cs`.

VarArgApp.cs

```
using System;

using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;

using VarArgComp;

namespace MathWorks.Examples.VarArgApp
{
    /// <summary>
    /// This application demonstrates how to call components
    /// having methods with varargin/varargout arguments.
    /// </summary>
    class VarArgApp
    {
        #region MAIN

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            // Initialize the input data
            MWNumericArray colorSpec= new double[]
                {0.9, 0.0, 0.0};

            MWNumericArray data=
                new MWNumericArray(new int[]{{1,2},{2,4},
                    {3,6},{4,8},{5,10}});

            MWArray[] coords= null;

            try
            {
                // Create a new plotter object
                Plotter plotter= new Plotter();
```

```

//Extract a variable number of two element x and y coordinate
// vectors from the data array
coords= plotter.extractcoords(5, data);

// Draw a graph using the specified color to connect the
// variable number of input coordinates.
// Return a two column data array containing the input coordinates.
data= (MWNumericArray)plotter.drawgraph(colorSpec,
    coords[0], coords[1], coords[2],coords[3], coords[4]);

Console.WriteLine("result=\n{0}", data);

Console.ReadLine(); // Wait for user to exit application

// Note: You can also pass in the coordinate array directly.
data= (MWNumericArray)plotter.drawgraph(colorSpec, coords);

Console.WriteLine("result=\n{0}", data);

Console.ReadLine(); // Wait for user to exit application
}

catch(Exception exception)
{
    Console.WriteLine("Error: {0}", exception);
}
}

#endregion
}
}

```

The following statements are alternative ways to call the `drawgraph` method:

```

data= (MWNumericArray)plotter.drawgraph(colorSpec,
    coords[0], coords[1], coords[2],coords[3], coords[4]);
...
data= (MWNumericArray)plotter.drawgraph((MWArray)colorSpec, coords);

```

- **Visual Basic**

If you are using Visual Basic, write source code for a Visual Basic application that accesses the component.

The sample application for this example is in `VarArgExample\VarArgVbApp\VarArgApp.vb`.

VarArgApp.vb

```

Imports System

Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays

Imports VarArgComp

Namespace MathWorks.Demo.VarArgDemoApp

    ' <summary>
    ' This application demonstrates how to call components having methods with
    ' varargin/varargout arguments.
    ' </summary>
    Class VarArgDemoApp

        #Region " MAIN "

            ' <summary>
            ' The main entry point for the application.
            ' </summary>
            Shared Sub Main(ByVal args() As String)

                ' Initialize the input data
                Dim colorSpec As MWNumericArray =
                    New MWNumericArray(New Double() {0.9, 0.0, 0.0})
            
```

```

Dim data As MWNumericArray =
    New MWNumericArray(New Integer(.) {{1, 2}, {2, 4}, {3, 6}, {4, 8}, {5, 10}})
Dim coords() As MArray = Nothing

Try
    ' Create a new plotter object
    Dim plotter As Plotter = New Plotter

    'Extract a variable number of two element x and y coordinate
    ' vectors from the data array
    coords = plotter.extractcoords(5, data)

    ' Draw a graph using the specified color to connect the variable number of
    ' input coordinates.
    ' Return a two column data array containing the input coordinates.
    data = CType(plotter.drawgraph(colorSpec, coords(0), coords(1), coords(2),
        coords(3), coords(4)), _
        MWNumericArray)

    Console.WriteLine("result={0}{1}", Chr(10), data)

    Console.ReadLine() ' Wait for user to exit application

    ' Note: You can also pass in the coordinate array directly.
    data = CType(plotter.drawgraph(colorSpec, coords), MWNumericArray)

    Console.WriteLine("result=\{0}{1}", Chr(10), data)

    Console.ReadLine() ' Wait for user to exit application

Catch exception As Exception
    Console.WriteLine("Error: {0}", exception)
End Try
End Sub
#End Region

End Class
End Namespace

```

The following statements are alternative ways to call the `drawgraph` method:

```

data = CType(plotter.drawgraph(colorSpec, coords(0), coords(1), coords(2),
    coords(3), coords(4)), MWNumericArray)
...
data = CType(plotter.drawgraph(colorSpec, coords), MWNumericArray)

```

In either case, the `VarArgApp` program does the following:

- Initializes three arrays (`colorSpec`, `data`, and `coords`) using the `MArray` class library
 - Creates a `Plotter` object
 - Calls the `extracoords` and `drawgraph` methods
 - Uses `MWNumericArray` to represent the data needed by the methods
 - Uses a `try-catch` block to catch and handle any exceptions
- 5 Open the `.NET` project file that corresponds to your application language using Visual Studio.
- **C#**

If you are using `C#`, the `VarArgCSApp` folder contains a Visual Studio `.NET` project file for this example. Open the project in Visual Studio `.NET` by double-clicking `VarArgCSApp.csproj` in Windows Explorer. You can also open it from the desktop by right-clicking `VarArgCSApp.csproj` and selecting **Open Outside MATLAB**.

- **Visual Basic**

If you are using Visual Basic, the `VarArgVBApp` folder contains a Visual Studio `.NET` project file for this example. Open the project in Visual Studio `.NET` by double-clicking `VarArgVBApp.vbproj` in Windows Explorer. You can also open it from the desktop by right-clicking `VarArgVBApp.vbproj` and selecting **Open Outside MATLAB**.

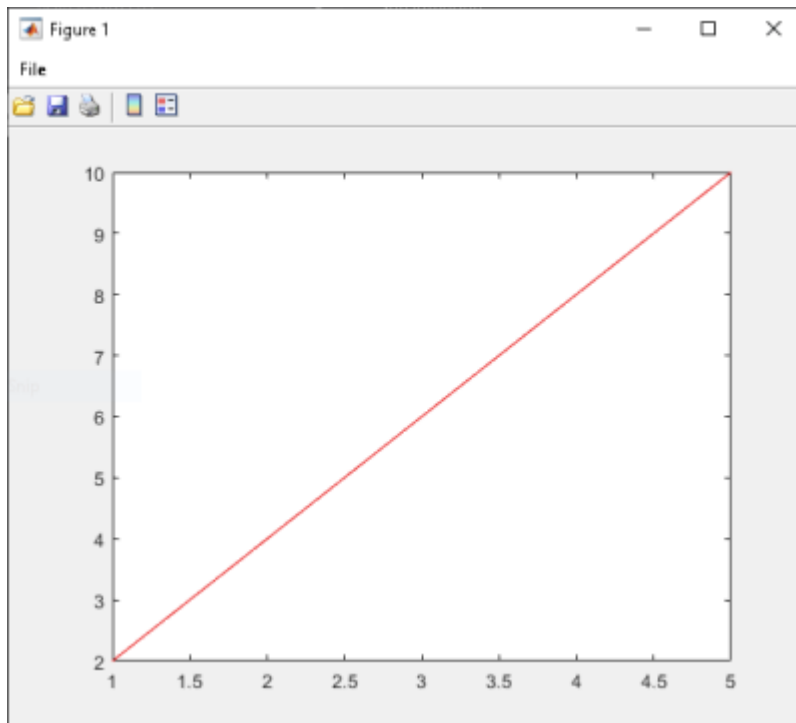
- 6 Create a reference to your assembly file `VarArgComp.dll` located in the folder where you generated or installed the assembly.
- 7 Create a reference to the `MWArray` API.

If MATLAB is installed on your system	<code>matlabroot\toolbox\dotnetbuilder\bin\win64\<framework_version>\MWArray.dll</code>
If MATLAB Runtime is installed on your system	<code><MATLAB_RUNTIME_INSTALL_DIR>\toolbox\dotnetbuilder\bin\win64\<framework_version>\MWArray.dll</code>

- 8 Build and run the `VarArgApp` application in Visual Studio .NET.

The program displays the following output:

```
result=
     1     2
     2     4
     3     6
     4     8
     5    10
```



See Also

`libraryCompiler` | `compiler.build.dotNETAssembly` | `deploytool`

Related Examples

- “Generate .NET Assembly and Build .NET Application”

- “Assign Multiple MATLAB Functions in Component Class” on page 3-27

Use Multiple Classes in .NET Assembly

In this section...

“SpectraComp Application” on page 3-19

“Files” on page 3-19

“Procedure” on page 3-19

This example shows you how to create a .NET assembly that uses multiple classes to analyze a signal and graph the result.

In this example, you perform the following steps:

- Use the MATLAB Compiler SDK product to create an assembly (`SpectraComp`) containing more than one class.
- Access the component in a C# application (`SpectraApp.cs`) or a Microsoft Visual Basic application (`SpectraApp.vb`), including use of the `MWArray` class hierarchy to represent data.
- Build and run the application using the Visual Studio .NET development environment.

SpectraComp Application

The class `SignalAnalyzer` performs a fast Fourier transform (FFT) on an input data array. A method of this class, `computefft`, returns the results of that FFT as two output arrays—an array of frequency points and the power spectral density.

The second class, `Plotter`, graphs the returned data using the `plotfft` method. The two methods `computefft` and `plotfft` encapsulate MATLAB functions. `computefft` computes the FFT and power spectral density of the input data and computes a vector of frequency points based on the length of the data entered and the sampling interval. `plotfft` plots the FFT data and the power spectral density in a MATLAB figure window.

Files

MATLAB Functions	<code>computefft.m</code> <code>plotfft.m</code>
MATLAB Function Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\SpectraExample\SpectraComp\</code>
C# Code Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\SpectraExample\SpectraCSApp\SpectraApp.cs</code>
Visual Basic Code Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\SpectraExample\SpectraVBApp\SpectraApp.vb</code>

Procedure

- 1 Copy the following folder that ships with the MATLAB product to your work folder:

```
matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\SpectraExample
```

At the MATLAB command prompt, navigate to the new `SpectraExample\SpectraComp` subfolder in your work folder.

- Examine the MATLAB functions `computefft.m` and `plotfft.m`.

computefft.m

```
function [fftData, freq, powerSpect] =
    ComputeFFT(data, interval)
if (isempty(data))
    fftdata = [];
    freq = [];
    powerspect = [];
    return;
end
if (interval <= 0)
    error('Sampling interval must be greater than zero');
    return;
end
fftData = fft(data);
freq = (0:length(fftData)-1)/(length(fftData)*interval);
powerSpect = abs(fftData)/(sqrt(length(fftData)));
```

plotfft.m

```
function PlotFFT(fftData, freq, powerSpect)

len = length(fftData);
if (len <= 0)
    return;
end
plot(freq(1:floor(len/2)), powerSpect(1:floor(len/2)))
xlabel('Frequency (Hz)'), grid on
title('Power spectral density')
```

- Build the .NET component with the **Library Compiler** app or `compiler.build.dotNETAssembly`.

Name the library `SpectraComp`. Map a class named `SignalAnalyzer` to the function `computefft.m`, and map a second class named `Plotter` to the function `plotfft.m`.

For example, if you are using `compiler.build.dotNETAssembly`, type:

```
cmap = containers.Map;
cmap('SignalAnalyzer') = 'computefft.m';
cmap('Plotter') = 'plotfft.m';
buildResults = compiler.build.dotNETAssembly(cmap, ...
    'AssemblyName', 'SpectraComp');
```

For more details, see the instructions in “Generate .NET Assembly and Build .NET Application”.

- Decide whether you are using C# or Visual Basic to access the component.

- **C#**

If you are using C#, write source code for a C# application that accesses the component.

The sample application for this example is in `SpectraExample\SpectraCSApp\SpectraApp.cs`.

SpectraApp.cs

```
using System;

using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;
```

```

using SpectraComp;

namespace MathWorks.Examples.SpectraApp
{
    /// <summary>
    /// This application computes and plots the power spectral density of an input signal.
    /// </summary>
    class SpectraCSApp
    {
        #region MAIN

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            try
            {
                const double interval= 0.01; // The sampling interval
                const int numSamples= 1001; // The number of samples

                // Construct input data as sin(2*PI*15*t) + (sin(2*PI*40*t)) plus a
                // random signal. Duration= 10; Sampling interval= 0.01
                MWNumericArray data= new MWNumericArray(MWArrayComplexity.Real,
                    MWNumericType.Double, numSamples);

                Random random= new Random();

                // Initialize data
                for (int idx= 1; idx <= numSamples; idx++)
                {
                    double t= (idx-1)* interval;

                    data[idx]= Math.Sin(2.0*Math.PI*15.0*t) + Math.Sin(2.0*Math.PI*40.0*t) +
                        random.NextDouble();
                }

                // Create a new signal analyzer object
                SignalAnalyzer signalAnalyzer= new SignalAnalyzer();

                // Compute the fft and power spectral density for the data array
                MWArray[] argsOut= signalAnalyzer.computefft(3, data, interval);

                // Print the first twenty elements of each result array
                int numElements= 20;

                MWNumericArray resultArray= new MWNumericArray(MWArrayComplexity.Complex,
                    MWNumericType.Double, numElements);

                for (int idx= 1; idx <= numElements; idx++)
                {
                    resultArray[idx]= ((MWNumericArray)argsOut[0])[idx];
                }

                Console.WriteLine("FFT:\n{0}\n", resultArray);

                for (int idx= 1; idx <= numElements; idx++)
                {
                    resultArray[idx]= ((MWNumericArray)argsOut[1])[idx];
                }

                Console.WriteLine("Frequency:\n{0}\n", resultArray);

                for (int idx= 1; idx <= numElements; idx++)
                {
                    resultArray[idx]= ((MWNumericArray)argsOut[2])[idx];
                }

                Console.WriteLine("Power Spectral Density:\n{0}", resultArray);

                // Create a new plotter object
                Plotter plotter= new Plotter();

                // Plot the fft and power spectral density for the data array
                plotter.plotfft(argsOut[0], argsOut[1], argsOut[2]);

                Console.ReadLine(); // Wait for user to exit application
            }
        }
    }
}

```

```

        catch(Exception exception)
        {
            Console.WriteLine("Error: {0}", exception);
        }
    }
}
#endregion
}
}

```

The following statement shows how to use the `MWArray` class library to construct a `MWNumericArray` that is used as method input to the `computefft` function.

```
MWNumericArray data= new MWNumericArray(MWArrayComplexity.Real,
MWNumericType.Double, numSamples);
```

The following statements create an instance of the class `SignalAnalyzer` and call the method `computefft`, requesting 3 outputs.

```
SignalAnalyzer signalAnalyzer = new SignalAnalyzer();
...
MWArray[] argsOut= signalAnalyzer.computefft(3, data, interval);
```

- **Visual Basic**

If you are using Visual Basic, write source code for a Visual Basic application that accesses the component.

The sample application for this example is in `SpectraExample\SpectraVBApp\SpectraApp.vb`.

SpectraApp.vb

```
Imports System

Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays

Imports SpectraComp

Namespace MathWorks.Examples.SpectraApp
    ' <summary>
    ' This application computes and plots the power spectral density of an input signal.
    ' </summary>
    Class SpectraDemoApp
#Region " MAIN "
        ' <summary>
        ' The main entry point for the application.
        ' </summary>
        Shared Sub Main(ByVal args() As String)
            Try
                Const interval As Double = 0.01 ' The sampling interval
                Const numSamples As Integer = 1001 ' The number of samples

                ' Construct input data as sin(2*PI*15*t) + (sin(2*PI*40*t) plus a
                ' random signal. Duration= 10; Sampling interval= 0.01
                Dim data As MWNumericArray = New MWNumericArray(MWArrayComplexity.Real,
                    MWNumericType.Double, numSamples)

                Dim random As Random = New Random

                ' Initialize data
                Dim t As Double
                Dim idx As Integer
                For idx = 1 To numSamples
                    t = (idx - 1) * interval
                    data(idx) = New MWNumericArray(Math.Sin(2.0 * Math.PI * 15.0 * t) +
                        Math.Sin(2.0 * Math.PI * 40.0 * t) +

```

```

        random.NextDouble())
    Next idx

    ' Create a new signal analyzer object
    Dim signalAnalyzer As SignalAnalyzer = New SignalAnalyzer

    ' Compute the fft and power spectral density for the data array
    Dim argsOut() As MWArray = signalAnalyzer.computefft(3, data,
        MWArray.op_implicit(interval))

    ' Print the first twenty elements of each result array
    Dim numElements As Integer = 20
    Dim resultArray As MWNumericArray =
        New MWNumericArray(MWArrayComplexity.Complex,
            MWNumericType.Double, numElements)

    For idx = 1 To numElements
        resultArray(idx) = (CType(argsOut(0), MWNumericArray))(idx)
    Next idx

    Console.WriteLine("FFT:{0}{1}{2}", Chr(10), resultArray, Chr(10))

    For idx = 1 To numElements
        resultArray(idx) = (CType(argsOut(1), MWNumericArray))(idx)
    Next idx

    Console.WriteLine("Frequency:{0}{1}{2}", Chr(10), resultArray, Chr(10))

    For idx = 1 To numElements
        resultArray(idx) = (CType(argsOut(2), MWNumericArray))(idx)
    Next idx

    Console.WriteLine("Power Spectral Density:{0}{1}{2}",
        Chr(10), resultArray, Chr(10))

    ' Create a new plotter object
    Dim plotter As Plotter = New Plotter

    ' Plot the fft and power spectral density for the data array
    plotter.plotfft(argsOut(0), argsOut(1), argsOut(2))

    Console.ReadLine() ' Wait for user to exit application

Catch exception As Exception
    Console.WriteLine("Error: {0}", exception)

End Try
End Sub
#End Region

End Class
End Namespace

```

The following statements show how to use the `MWArray` class library to construct the necessary data types:

```

Dim data As MWNumericArray = New MWNumericArray_
    (MWArrayComplexity.Real, MWNumericType.Double, numSamples)
...
Dim resultArray As MWNumericArray = New MWNumericArray_
    (MWArrayComplexity.Complex,
        MWNumericType.Double, numElements)

```

The following statements create an instance of the class `SignalAnalyzer` and call the method `computefft`, requesting three outputs:

```

Dim signalAnalyzer As SignalAnalyzer = New SignalAnalyzer
...
Dim argsOut() As MWArray =
    signalAnalyzer.computefft(3, data,
        MWArray.op_implicit(interval))

```

In either case, the `SpectraApp` program does the following:

- Constructs an input array with values representing a random signal with two sinusoids at 15 and 40 Hz embedded inside of it
 - Creates an `MWNumericArray` array that contains the data
 - Instantiates a `SignalAnalyzer` object
 - Calls the `computefft` method, which computes the FFT, frequency, and the spectral density
 - Instantiates a `Plotter` object
 - Calls the `plotfft` method, which plots the data
 - Uses a `try/catch` block to handle exceptions
- 5 Open the .NET project file that corresponds to your application language using Visual Studio.

- **C#**

If you are using C#, the `SpectraCSApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `SpectraCSApp.csproj` in Windows Explorer. You can also open it from the desktop by right-clicking **SpectraCSApp.csproj** and selecting **Open Outside MATLAB**.

- **Visual Basic**

If you are using Visual Basic, the `SpectraVBApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `SpectraVBApp.vbproj` in Windows Explorer. You can also open it from the desktop by right-clicking **SpectraVBApp.vbproj** and selecting **Open Outside MATLAB**.

- 6 Add a reference to your assembly file `SpectraComp.dll`.

- 7 Add a reference to the `MWArray` API.

If MATLAB is installed on your system	<code>matlabroot\toolbox\dotnetbuilder\bin\win64\<framework_version>\MWArray.dll</code>
If MATLAB Runtime is installed on your system	<code><MATLAB_RUNTIME_INSTALL_DIR>\toolbox\dotnetbuilder\bin\win64\<framework_version>\MWArray.dll</code>

- 8 Build and run the `SpectraApp` application in Visual Studio .NET.

The program displays the following output:

```
FFT:
  1.0e+02 *

  4.8646 + 0.0000i
 -0.0289 + 0.1080i
 -0.0326 + 0.0237i
 -0.0141 - 0.0148i
  0.0674 - 0.0487i
  0.0753 + 0.0669i
  0.0275 - 0.0101i
 -0.0429 + 0.0472i
  0.0803 - 0.1163i
 -0.0619 - 0.1072i
  0.0565 - 0.0502i
 -0.0223 + 0.0587i
 -0.0853 - 0.0812i
```

```
-0.0662 - 0.0143i
0.0543 - 0.0972i
0.0814 - 0.0463i
-0.0981 - 0.0190i
0.0042 + 0.0083i
-0.0339 + 0.0290i
0.0291 + 0.0036i
```

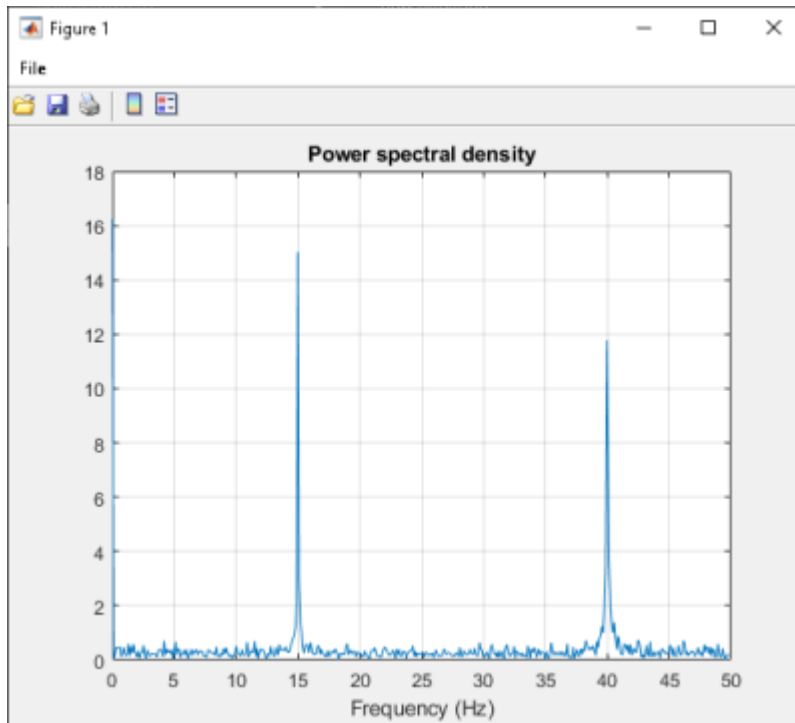
Frequency:

```
0.0000 + 0.0000i
0.0999 + 0.0000i
0.1998 + 0.0000i
0.2997 + 0.0000i
0.3996 + 0.0000i
0.4995 + 0.0000i
0.5994 + 0.0000i
0.6993 + 0.0000i
0.7992 + 0.0000i
0.8991 + 0.0000i
0.9990 + 0.0000i
1.0989 + 0.0000i
1.1988 + 0.0000i
1.2987 + 0.0000i
1.3986 + 0.0000i
1.4985 + 0.0000i
1.5984 + 0.0000i
1.6983 + 0.0000i
1.7982 + 0.0000i
1.8981 + 0.0000i
```

Power Spectral Density:

```
15.3755 + 0.0000i
0.3534 + 0.0000i
0.1274 + 0.0000i
0.0646 + 0.0000i
0.2628 + 0.0000i
0.3183 + 0.0000i
0.0925 + 0.0000i
0.2016 + 0.0000i
0.4465 + 0.0000i
0.3912 + 0.0000i
0.2387 + 0.0000i
0.1985 + 0.0000i
0.3723 + 0.0000i
0.2140 + 0.0000i
0.3520 + 0.0000i
0.2960 + 0.0000i
0.3158 + 0.0000i
0.0294 + 0.0000i
0.1411 + 0.0000i
0.0927 + 0.0000i
```

Plotting Power spectral density, please wait...



See Also

`libraryCompiler` | `compiler.build.dotNETAssembly` | `deploytool`

Related Examples

- “Generate .NET Assembly and Build .NET Application”

More About

- “Convert Data Between .NET and MATLAB” on page 2-8

Assign Multiple MATLAB Functions in Component Class

In this section...

"MatrixMathApp Application" on page 3-27

"Files" on page 3-27

"Procedure" on page 3-28

"Understanding the MatrixMath Program" on page 3-34

This example shows you how to create a .NET matrix math program using multiple MATLAB functions.

In this example, you perform the following steps:

- Assign more than one MATLAB function to a component class.
- Access the component in a C# application (`MatrixMathApp.cs`) or a Visual Basic application (`MatrixMathApp.vb`) by instantiating `Factor` and using the `MWArray` class library to handle data conversion.
- Build and run the `MatrixMathApp` application using the Visual Studio .NET development environment.

MatrixMathApp Application

The `MatrixMathApp` application performs Cholesky, LU, and QR factorizations on a simple tridiagonal matrix (finite difference matrix) with the following form:

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

You supply the size of the matrix on the command line, and the program constructs the matrix and performs the three factorizations. The original matrix and the results are printed to standard output. You may optionally perform the calculations using a sparse matrix by specifying the string "sparse" as the second parameter on the command line.

Files

MATLAB Functions	<code>cholesky.m</code> <code>ludecomp.m</code> <code>qrdecomp.m</code>
MATLAB Function Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\MatrixMathExample\MatrixMathComp\</code>
C# Code Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\MatrixMathExample\MatrixMathCSApp\MatrixMathApp.cs</code>
Visual Basic Code Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\MatrixMathExample\MatrixMathVBApp\MatrixMathApp.vb</code>

Procedure

- 1 Copy the following folder that ships with the MATLAB product to your work folder:

```
matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\MatrixMathExample
```

At the MATLAB command prompt, navigate to the new `MatrixMathExample\MatrixMathComp` subfolder in your work folder.

- 2 Examine the MATLAB functions `cholesky.m`, `ludecomp.m`, and `qrdecomp.m`.

```
function [L] = Cholesky(A)
    L = chol(A);

function [L,U] = LUDecomp(A)
    [L,U] = lu(A);

function [Q,R] = QRDecomp(A)

    [Q,R] = qr(A);
```

- 3 Build the .NET component with the **Library Compiler** app or `compiler.build.dotNETAssembly` using the following information:

Field	Value
Library Name	MatrixMathComp
Class Name	Factor
Files to compile	cholesky ludecomp qrdecomp

For example, if you are using `compiler.build.dotNETAssembly`, type:

```
buildResults = compiler.build.dotNETAssembly(["cholesky.m", "ludecomp.m", "qrdecomp.m"], ...
    'AssemblyName', 'MatrixMathComp', ...
    'ClassName', 'Factor');
```

For more details, see the instructions in “Generate .NET Assembly and Build .NET Application”.

- 4 Decide whether you are using C# or Visual Basic to access the component.

- **C#**

If you are using C#, write source code for a C# application that accesses the component.

The sample application for this example is in `MatrixMathExample\MatrixMathCSApp\MatrixMathApp.cs`.

MatrixMathApp.cs

```
using System;

using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;

using MatrixMathComp;

namespace MathWorks.Examples.MatrixMath
{
    /// <summary>
    /// This application computes cholesky, LU, and QR factorizations of a finite
    /// difference matrix of order N.
    /// The order is passed into the application on the command line.
    /// </summary>
    /// <remarks>
    /// Command Line Arguments:
    /// <newpara></newpara>
    /// args[0] - Matrix order(N)
    /// <newpara></newpara>
    /// args[1] - (optional) sparse; Use a sparse matrix
    /// </remarks>
```

```

class MatrixMathApp
{
    #region MAIN

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
        bool makeSparse= true;
        int matrixOrder= 4;

        MWNumericArray matrix= null; // The matrix to factor

        MWArray argOut= null; // Stores single factorization result
        MWArray[] argsOut= null; // Stores multiple factorization results

        try
        {
            // If no argument specified, use defaults
            if (0 != args.Length)
            {
                // Convert matrix order
                matrixOrder= Int32.Parse(args[0]);

                if (0 >= matrixOrder)
                {
                    throw new ArgumentOutOfRangeException("matrixOrder", matrixOrder,
                        "Must enter a positive integer for the matrix order(N)");
                }

                makeSparse= ((1 < args.Length) && (args[1].Equals("sparse")));
            }

            // Create the test matrix. If the second argument is "sparse",
            // create a sparse matrix.
            matrix= (makeSparse)
                ? MWNumericArray.MakeSparse(matrixOrder, matrixOrder,
                    MWArrayComplexity.Real, (matrixOrder+(2*(matrixOrder-1))))
                : new MWNumericArray(MWArrayComplexity.Real,
                    MWNumericType.Double, matrixOrder, matrixOrder);

            // Initialize the test matrix
            for (int rowIdx= 1; rowIdx <= matrixOrder; rowIdx++)
                for (int colIdx= 1; colIdx <= matrixOrder; colIdx++)
                    if (rowIdx == colIdx)
                        matrix[rowIdx, colIdx]= 2.0;
                    else if ((colIdx == rowIdx+1) || (colIdx == rowIdx-1))
                        matrix[rowIdx, colIdx]= -1.0;

            // Create a new factor object
            Factor factor= new Factor();

            // Print the test matrix
            Console.WriteLine("Test Matrix:\n{0}\n", matrix);

            // Compute and print the cholesky factorization using the
            // single output syntax
            argOut= factor.cholesky((MWArray)matrix);

            Console.WriteLine("Cholesky
                Factorization:\n{0}\n", argOut);

            // Compute and print the LU factorization using the multiple output syntax
            argsOut= factor.ludecomp(2, matrix);

            Console.WriteLine("LU Factorization:\nL
                Matrix:\n{0}\nU Matrix:\n{1}\n", argsOut[0],
                argsOut[1]);

            MWNumericArray.DisposeArray(argsOut);

            // Compute and print the QR factorization
            argsOut= factor.qrdecomp(2, matrix);

            Console.WriteLine("QR Factorization:\nQ Matrix:\n{0}\nR Matrix:\n{1}\n",
                argsOut[0], argsOut[1]);

            Console.ReadLine();
        }
    }
}

```

```

        catch(Exception exception)
        {
            Console.WriteLine("Error: {0}", exception);
        }

        finally
        {
            // Free native resources
            if (null != (object)matrix) matrix.Dispose();
            if (null != (object)argOut) argOut.Dispose();

            MWNumericArray.DisposeArray(argsOut);
        }
    }
}
#endregion
}
}

```

This statement creates an instance of the class `Factor`:

```
Factor factor= new Factor();
```

The following statements call the methods that encapsulate the MATLAB functions:

```

argOut= factor.cholesky((MArray)matrix);
...
argsOut= factor.ludecomp(2, matrix);
...
argsOut= factor.qrdecomp(2, matrix);
...

```

- **Visual Basic**

If you are using Visual Basic, write source code for a Visual Basic application that accesses the component.

The sample application for this example is in `MatrixMathExample\MatrixMathVBApp\MatrixMathApp.vb`.

MatrixMathApp.vb

```

Imports System

Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays

Imports MatrixMathComp

Namespace MathWorks.Demo.MatrixMathApp

    ' <summary>
    ' This application computes cholesky, LU, and QR factorizations of a
    ' finite difference matrix of order N.
    ' The order is passed into the application on the command line.
    ' </summary>
    ' <remarks>
    ' Command Line Arguments:
    ' <newpara></newpara>
    ' args[0] - Matrix order(N)
    ' <newpara></newpara>
    ' args[1] - (optional) sparse; Use a sparse matrix
    ' </remarks>
    Class MatrixMathDemoApp

#Region " MAIN "

        ' <summary>
        ' The main entry point for the application.
        ' </summary>
        Shared Sub Main(ByVal args() As String)

```

```

Dim makeSparse As Boolean = True
Dim matrixOrder As Integer = 4

Dim matrix As MWNumericArray = Nothing ' The matrix to factor

Dim argOut As MWArray = Nothing ' Stores single factorization result
Dim argsOut() As MWArray = Nothing ' Stores multiple factorization results

Try
    ' If no argument specified, use defaults
    If (0 <> args.Length) Then
        'Convert matrix order
        matrixOrder = Int32.Parse(args(0))

        If (0 > matrixOrder) Then
            Throw New ArgumentOutOfRangeException("matrixOrder", matrixOrder, _
                "Must enter a positive integer for the matrix order(N)")
        End If

        makeSparse = ((1 < args.Length) AndAlso (args(1).Equals("sparse")))
    End If

    ' Create the test matrix. If the second argument
    ' is "sparse", create a sparse matrix.
    matrix = IIf(makeSparse, _
        MWNumericArray.MakeSparse(matrixOrder, matrixOrder,
            MWArrayComplexity.Real,
            (matrixOrder + (2 * (matrixOrder - 1)))), _
        New MWNumericArray(MWArrayComplexity.Real, MWNumericType.Double,
            matrixOrder, matrixOrder))

    ' Initialize the test matrix
    For rowIdx As Integer = 1 To matrixOrder
        For colIdx As Integer = 1 To matrixOrder
            If rowIdx = colIdx Then
                matrix(rowIdx, colIdx) = New MWNumericArray(2.0)
            ElseIf colIdx = rowIdx + 1 Or colIdx = rowIdx - 1 Then
                matrix(rowIdx, colIdx) = New MWNumericArray(-1.0)
            End If
        Next colIdx
    Next rowIdx

    ' Create a new factor object
    Dim factor As Factor = New Factor

    ' Print the test matrix
    Console.WriteLine("Test Matrix:{0}{1}{2}", Chr(10), matrix, Chr(10))

    ' Compute and print the cholesky factorization using
    ' the single output syntax
    argOut = factor.cholesky(matrix)

    Console.WriteLine("Cholesky Factorization:{0}{1}{2}",
        Chr(10), argOut, Chr(10))

    ' Compute and print the LU factorization using the multiple output syntax
    argsOut = factor.ludecomp(2, matrix)

    Console.WriteLine("LU Factorization:
        {0}L Matrix:{1}{2}{3}U Matrix:{4}{5}{6}", Chr(10), Chr(10),
        argsOut(0), Chr(10), Chr(10), argsOut(1), Chr(10))

    MWNumericArray.DisposeArray(argsOut)

    ' Compute and print the QR factorization
    argsOut = factor.qrdecomp(2, matrix)

    Console.WriteLine("QR Factorization:
        {0}Q Matrix:{1}{2}{3}R Matrix:{4}{5}{6}", Chr(10), Chr(10),
        argsOut(0), Chr(10), Chr(10), argsOut(1), Chr(10))

    Console.ReadLine()

Catch exception As Exception
    Console.WriteLine("Error: {0}", exception)

Finally
    ' Free native resources

```

```

        If Not (matrix Is Nothing) Then
            matrix.Dispose()
        End If
        If Not (argOut Is Nothing) Then
            argOut.Dispose()
        End If

        MWNumericArray.DisposeArray(argsOut)
    End Try
End Sub
#End Region
End Class
End Namespace

```

This statement creates an instance of the class `Factor`:

```
Dim factor As Factor = New Factor
```

The following statements call the methods that encapsulate the MATLAB functions:

```
argOut = factor.cholesky(matrix)
```

```
argsOut = factor.ludecomp(2, matrix)
```

```
argsOut = factor.qrdecomp(2, matrix)
```

- Open the .NET project file that corresponds to your application language using Visual Studio.

- **C#**

If you are using C#, the `MatrixMathCSApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `MatrixMathCSApp.csproj` in Windows Explorer. You can also open it from the desktop by right-clicking **MatrixMathCSApp.csproj** and selecting **Open Outside MATLAB**.

- **Visual Basic**

If you are using Visual Basic, the `MatrixMathVBApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `MatrixMathVBApp.vbproj` in Windows Explorer. You can also open it from the desktop by right-clicking **MatrixMathVBApp.vbproj** and selecting **Open Outside MATLAB**.

- Add a reference to your assembly file `MatrixMathComp.dll` located in the folder where you generated the assembly.
- Add a reference to the `MWArray` API.

If MATLAB is installed on your system	<code>matlabroot\toolbox\dotnetbuilder\bin\win64\<framework_version>\MWArray.dll</code>
If MATLAB Runtime is installed on your system	<code><MATLAB_RUNTIME_INSTALL_DIR>\toolbox\dotnetbuilder\bin\win64\<framework_version>\MWArray.dll</code>

- Build and run the `MatrixMathApp` application in Visual Studio.

The program displays the following output:

Test Matrix:

```

(1,1)      2
(2,1)     -1
(1,2)     -1
(2,2)      2

```

```
(3,2)    -1
(2,3)    -1
(3,3)     2
(4,3)    -1
(3,4)    -1
(4,4)     2
```

Cholesky Factorization:

```
(1,1)    1.4142
(1,2)   -0.7071
(2,2)    1.2247
(2,3)   -0.8165
(3,3)    1.1547
(3,4)   -0.8660
(4,4)    1.1180
```

LU Factorization:

L Matrix:

```
(1,1)    1.0000
(2,1)   -0.5000
(2,2)    1.0000
(3,2)   -0.6667
(3,3)    1.0000
(4,3)   -0.7500
(4,4)    1.0000
```

U Matrix:

```
(1,1)    2.0000
(1,2)   -1.0000
(2,2)    1.5000
(2,3)   -1.0000
(3,3)    1.3333
(3,4)   -1.0000
(4,4)    1.2500
```

QR Factorization:

Q Matrix:

```
(1,1)   -0.8944
(2,1)    0.4472
(1,2)   -0.3586
(2,2)   -0.7171
(3,2)    0.5976
(1,3)   -0.1952
(2,3)   -0.3904
(3,3)   -0.5855
(4,3)    0.6831
(1,4)    0.1826
(2,4)    0.3651
(3,4)    0.5477
(4,4)    0.7303
```

R Matrix:

```
(1,1)   -2.2361
(1,2)    1.7889
(2,2)   -1.6733
(1,3)   -0.4472
(2,3)    1.9124
(3,3)   -1.4639
(2,4)   -0.5976
```

```
(3,4)      1.9518  
(4,4)      0.9129
```

Understanding the MatrixMath Program

The `MatrixMath` program takes one or two arguments from the command line. The first argument is converted to the integer order of the test matrix. If the string `sparse` is passed as the second argument, a sparse matrix is created to contain the test array. The Cholesky, LU, and QR factorizations are then computed and the results are displayed.

The main method has three parts:

- The first part sets up the input matrix, creates a new factor object, and calls the `cholesky`, `ludcomp`, and `qrdecomp` methods. This part is executed inside of a `try` block. This is done so that if an exception occurs during execution, the corresponding `catch` block will be executed.
- The second part is the `catch` block. The code prints a message to standard output to let the user know about the error that has occurred.
- The third part is a `finally` block to manually clean up native resources before exiting. This is optional, as the garbage collector will automatically clean up resources for you.

See Also

`libraryCompiler` | `compiler.build.dotNETAssembly` | `deploytool`

Related Examples

- “Generate .NET Assembly and Build .NET Application”
- “Use Multiple Classes in .NET Assembly” on page 3-19

More About

- “Convert Data Between .NET and MATLAB” on page 2-8

Integrate MATLAB Optimization Routines with Objective Functions

In this section...

“Overview” on page 3-35

“OptimizeComp Application” on page 3-35

“Files” on page 3-35

“Procedure” on page 3-36

Overview

This example shows you how to create a .NET application that finds a local minimum of an objective function using the MATLAB optimization function `fminsearch` and the `MWObjectArray` class.

In this example, you perform the following steps:

- Use the MATLAB Compiler SDK product to create an assembly (`OptimizeComp`). This assembly applies MATLAB optimization routines to objective functions implemented as .NET objects.
- Access the component in either a C# application (`OptimizeApp.cs`) or a Visual Basic application (`OptimizeApp.vb`).
- Use the `MWObjectArray` class to create a reference to a .NET object using C# (`BananaFunction.cs`) or Visual Basic (`BananaFunction.vb`), and pass that object to the component.
- Build and run the application using the Visual Studio .NET development environment.

OptimizeComp Application

The `OptimizeComp` application finds a local minimum of an objective function and returns the minimal location and value. The component uses the MATLAB optimization function `fminsearch`. This example optimizes the Rosenbrock banana function used in the `fminsearch` documentation.

The class `OptimizeComp.OptimizeClass` performs an unconstrained nonlinear optimization on an objective function implemented as a .NET object. A method of this class, `doOptim`, accepts an initial value (NET object) that implements the objective function, and returns the location and value of a local minimum.

The second method, `displayObj`, is a debugging tool that lists the characteristics of a .NET object. The two methods `doOptim` and `displayObj` encapsulate MATLAB functions.

Files

MATLAB Functions	<code>doOptim.m</code> <code>displayObj.m</code>
MATLAB Function Location	<code>matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\OptimizeExample\OptimizeComp</code>

C# Code Location	<i>matlabroot</i> \toolbox\dotnetbuilder\Examples\VSVersion\NET\OptimizeExample\OptimizeCSApp\BananaFunction.cs
Visual Basic Code Location	<i>matlabroot</i> \toolbox\dotnetbuilder\Examples\VSVersion\NET\OptimizeExample\OptimizeVBAApp\BananaFunction.vb
MWArray API Reference Location	<i>matlabroot</i> \help\dotnetbuilder\MWArrayAPI

Procedure

- 1 Copy the following folder that ships with MATLAB to your work folder:
matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\OptimizeExample

At the MATLAB command prompt, navigate to the new `OptimizeExample\OptimizeComp` subfolder in your work folder.

- 2 Examine the MATLAB code that you want to access. This example uses `doOptim.m` and `displayObj.m`.

```
function [x,fval] = doOptim(h, x0)
mWrapper = @(x) h.evaluateFunction(x);
```

```
directEval = h.evaluateFunction(x0)
wrapperEval = mWrapper(x0)
```

```
[x,fval] = fminsearch(mWrapper,x0)
```

```
function className = displayObj(h)
```

```
h
className = class(h)
whos('h')
methods(h)
```

- 3 Build the .NET component with the **Library Compiler** app or `compiler.build.dotNETAssembly` using the following information:

Field	Value
Library Name	OptimizeComp
Class Name	OptimizeComp.OptimizeClass
Files to Compile	doOptim.m displayObj.m

For example, if you are using `compiler.build.dotNETAssembly`, type:

```
buildResults = compiler.build.dotNETAssembly(["doOptim.m","displayObj.m"], ...
'AssemblyName','OptimizeComp', ...
'ClassName','OptimizeComp.OptimizeClass');
```

For more details, see the instructions in “Generate .NET Assembly and Build .NET Application”.

- 4 Decide whether you are using C# or Visual Basic to access the component.

- **C#**

If you are using C#, write source code for a class that implements an object function to optimize.

The sample application for this example is in `OptimizeExample\OptimizeCSApp\BananaFunction.cs`

BananaFunction.cs

```
using System;

namespace MathWorks.Examples.Optimize
{
    public class BananaFunction
    {
        public BananaFunction() {}

        public double evaluateFunction(double[] x)
        {
            double term1= 100*Math.Pow((x[1]-Math.Pow(x[0],2.0)),2.0);
            double term2= Math.Pow((1-x[0]),2.0);
            return term1+term2;
        }
    }
}
```

- **Visual Basic**

If you are using Visual Basic, write source code for a class that implements an object function to optimize.

The sample application for this example is in `OptimizeExample\OptimizeVBApp\BananaFunction.vb`.

BananaFunction.vb

```
Imports System

Namespace MathWorks.Examples.Optimize

    Class BananaFunction

        #Region "Methods"
        Public Sub BananaFunction()
            End Sub

        Public Function evaluateFunction(ByVal x As Double()) As Double

            Dim term1 As Double = 100 * Math.Pow((x(1) - Math.Pow(x(0),
                2.0)), 2.0)

            Dim term2 As Double = Math.Pow((1 - x(0)), 2.0)
            Return term1 + term2
        End Function
        #End Region

    End Class
End Namespace
```

The `BananaFunction` class implements the Rosenbrock banana function described in the `fminsearch` documentation.

- 5 Open the .NET project file that corresponds to your application language using Visual Studio.

- **C#**

If you are using C#, the `OptimizeCSApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `OptimizeCSApp.csproj` in Windows Explorer. You can also open it from the desktop by right-clicking **OptimizeCSApp.csproj** and selecting **Open Outside MATLAB**.

- **Visual Basic**

If you are using Visual Basic, the `OptimizeVBApp` folder contains a Visual Studio .NET project file for this example. Open the project in Visual Studio .NET by double-clicking `OptimizeVBApp.vbproj` in Windows Explorer. You can also open it from the desktop by right-clicking **OptimizeVBApp.vbproj** and selecting **Open Outside MATLAB**.

- 6 Add a reference to your assembly file `OptimizeComp.dll`.
- 7 Add a reference to the `MWArray` API.

If MATLAB is installed on your system	<code>matlabroot\toolbox\dotnetbuilder\bin\win64\<framework_version>\MWArray.dll</code>
If MATLAB Runtime is installed on your system	<code><MATLAB_RUNTIME_INSTALL_DIR>\toolbox\dotnetbuilder\bin\win64\<framework_version>\MWArray.dll</code>

- 8 Build and run the `OptimizeApp` application in Visual Studio .NET.

The program displays the following output:

Using initial points= -1.2000 1

```
*****
**                Properties of .NET Object                **
*****
```

h =

```
MathWorks.Examples.Optimize.BananaFunction handle
with no properties.
Package: MathWorks.Examples.Optimize
```

className =

MathWorks.Examples.Optimize.BananaFunction

```

Name   Size   Bytes  Class  Attributes
-----
h      1x1    60    MathWorks.Examples.Optimize.BananaFunction
```

Methods for class `MathWorks.Examples.Optimize.BananaFunction`:

```

BananaFunction  addlistener  findprop  lt
Equals          delete      ge        ne
GetHashCode     eq         gt        notify
GetType         evaluateFunction  isvalid
ToString       findobj    le
```

```
***** Finished displayObj *****
```

```
*****  
** Performing unconstrained nonlinear optimization **  
*****
```

```
directEval =
```

```
    24.2000
```

```
wrapperEval =
```

```
    24.2000
```

```
x =
```

```
    1.0000    1.0000
```

```
fval =
```

```
    8.1777e-010
```

```
***** Finished doOptim *****
```

```
Location of minimum: 1.0000    1.0000  
Function value at minimum: 8.1777e-010
```

See Also

[libraryCompiler](#) | [compiler.build.dotNETAssembly](#) | [deploytool](#)

Related Examples

- “Generate .NET Assembly and Build .NET Application”

More About

- “Use Multiple Classes in .NET Assembly” on page 3-19

Build .NET Core Application That Runs on Linux and macOS

Supported Platform: Windows (Authoring), Linux® (Execution), and macOS (Execution).

This example shows how to create a .NET assembly using the **Library Compiler** and integrate it into a .NET Core application that can run on Linux or macOS.

Prerequisites

- 1 Create a new work folder that is visible to the MATLAB search path. This example uses `C:\Work` as the new work folder.
- 2 Install MATLAB Runtime on Windows and on additional platforms where you plan on running your .NET Core application. For details, see “Install and Configure MATLAB Runtime”.
- 3 For Linux and macOS platforms, after installing MATLAB Runtime, you need to set the `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` environment variables respectively. For more information, see “Set MATLAB Runtime Path for Deployment”.
- 4 Verify that you have Visual Studio and .NET Core 2.0 or higher installed. If you have version 15.8.2 of Visual Studio 2017 installed, then you do not need to install .NET Core 2.0 or higher separately.

Create .NET Assembly

Package the function into a .NET assembly using the **Library Compiler** app. Alternatively, if you want to create a .NET assembly from the MATLAB command window using a programmatic approach, see `compiler.build.dotNETAssembly`.

- 1 Create a new MATLAB file named `mymagic.m` with the following code in the work folder:

```
function out = mymagic(in)
out = magic(in);
```

- 2 Type `libraryCompiler` at the MATLAB command line to launch the Library Compiler app.
- 3 In the **TYPE** section of the toolstrip, select **.NET Assembly**, and in the **EXPORTED FUNCTIONS** section, click the **Add** button to add the file `mymagic.m` to the project.
- 4 In the **Library information** section, name the library `MyMatrixFunctions`.
- 5 Double-click the class `Class1` and rename it as `MyMagic`.
- 6 Save the deployment project with the default project name `MyMatrixFunctions`.
- 7 Select **Package** to create a .NET assembly. For information about the created files, see “Files Generated After Packaging MATLAB Functions”.

Create .NET Core Application

- 1 Open the command prompt in Windows and navigate to the folder `C:\Work`.
- 2 At the command line, type:

```
dotnet new console --name MyDotNetCoreApplication
```

This creates a folder named `MyDotNetCoreApplication` that has the following contents:

- obj folder
- MyDotNetCoreApp.csproj project file
- Program.cs C# source file

3 Open the project file in a text editor.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Add the following references to the project using the <ItemGroup> tag:

- .NET assembly file MyMatrixFunctions.dll created by the Library Compiler app
- MWArray.dll, which is located in <MATLAB_RUNTIME_INSTALL_DIR>\toolbox\dotnetbuilder\bin\win64\<framework_version>

Once you add the references, your project file should resemble the following:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <Reference Include="MyMatrixFunctions">
      <HintPath>C:\work\MyMatrixFunctions\for_redistribution_files_only\MyMatrixFunctions.dll</HintPath>
      <!--Path to .NET Assembly created by Library Compiler app-->
    </Reference>
    <Reference Include="MWArray">
      <HintPath>C:\Program Files\MATLAB\MATLAB Runtime\v97\toolbox\dotnetbuilder\bin\win64\v4.0\MWArray.dll</HintPath>
      <!--Path to MWArray.dll in the MATLAB Runtime-->
    </Reference>
  </ItemGroup>
</Project>
```

4 Open the C# source file Program.cs and replace the existing code with the following code:

Program.cs

```
// *****
//
// Program.cs
//
// This example demonstrates how to use MATLAB .NET Assembly to build a simple
// component returning a magic square and how to convert MWNumericArray types
// to native .NET types.
//
// Copyright 2001-2019 The MathWorks, Inc.
//
// *****
```

```
using System;
```

```
using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;
```

```
using MyMatrixFunctions;

namespace MathWorks.Examples.MagicSquare
{
    /// <summary>
    /// The MagicSquareApp class computes a magic square of the user specified size.
    /// </summary>
    /// <remarks>
    /// args[0] - a positive integer representing the size of the magic square.
    /// </remarks>
    class Program
    {
        #region MAIN

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            MWNumericArray arraySize= null;
            MWNumericArray magicSquare= null;

            try
            {
                // Get user specified command line arguments or set default
                arraySize= (0 != args.Length) ? Double.Parse(args[0]) : 4;

                // Create the magic square object
                MyMagic magic= new MyMagic();

                // Compute the magic square and print the result
                magicSquare= (MWNumericArray)magic.mymagic(arraySize);

                Console.WriteLine("Magic square of order {0}\n\n{1}", arraySize, magicSquare);

                // Convert the magic square array to a two dimensional native double array
                double[,] nativeArray= (double[,])magicSquare.ToArray(MWArrayComponent.Real);

                Console.WriteLine("\nMagic square as native array:\n");

                // Display the array elements:
                for (int i= 0; i < (int)arraySize; i++)
                    for (int j= 0; j < (int)arraySize; j++)
                        Console.WriteLine("Element({0},{1})= {2}", i, j, nativeArray[i,j]);

                Console.ReadLine(); // Wait for user to exit application
            }

            catch(Exception exception)
            {
                Console.WriteLine("Error: {0}", exception);
            }
        }

        #endregion
    }
}
```


- 5 At the command line, build your .NET Core project by typing:

```
dotnet build MyDotNetCoreApp.csproj
```

- 6 At the command line, run your application by typing:

```
dotnet run -- 3
```

The application displays a 3x3 magic square.

- 7 Publish the project as a self-contained deployment to execute the application on either Linux or macOS.

- To publish to Linux, type the following command on a single line:

```
dotnet publish --configuration Release --framework netcoreapp2.2
  --runtime linux-x64 --self-contained true MyDotNetCoreApp.csproj
```

- To publish to macOS, type the following command on a single line:

```
dotnet publish --configuration Release --framework netcoreapp2.2
  --runtime osx.10.11-x64 --self-contained true MyDotNetCoreApp.csproj
```

Run .NET Core Application on UNIX

- 1 Copy the Release folder from C:\Work\MyDotNetCoreApp\bin on Windows to ~/Work on a Linux or macOS machine.
- 2 On the Linux machine, verify that you have installed MATLAB Runtime and set up your library path environment variable. For more information, see “Prerequisites” on page 3-40.
- 3 Open a command shell and navigate to:

```
~/Work/Release/netcoreapp2.2/<os-architecture>/publish
```

- 4 Run the .NET Core application by typing:

```
./MyDotNetCoreApp 3
```

Magic square of order 3

8	1	6
3	5	7
4	9	2

Magic square as native array:

```
Element(0,0)= 8
Element(0,1)= 1
Element(0,2)= 6
Element(1,0)= 3
Element(1,1)= 5
Element(1,2)= 7
Element(2,0)= 4
Element(2,1)= 9
Element(2,2)= 2
```

See Also

Related Examples

- [“Install and Configure MATLAB Runtime”](#)
- [“Set MATLAB Runtime Path for Deployment”](#)

Microsoft Visual Basic Integration Examples

Note The examples for the MATLAB Compiler SDK product are in `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion`, where `matlabroot` is the folder where the MATLAB product is installed and `VSVersion` specifies the version of Microsoft Visual Studio .NET you are using. If you have Microsoft Visual Studio .NET installed, you can load projects for all the examples by opening the following solution:

`matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\DotNetExamples.sln`

Integrate a .NET Assembly Into a Visual Basic Application

To create the component for this example, see the first several steps in “Generate .NET Assembly and Build .NET Application”. After you build the `MagicSquareComp` component, you can build an application that accesses the component as follows.

- 1 For this example, the application is `MagicSquareApp.vb`.

You can find `MagicSquareApp.vb` in:

```
matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\MagicSquareExample\MagicSquareVBAApp
```

The program listing is as follows.

MagicSquareApp.vb

```
Imports System
Imports System.Reflection
Imports MathWorks.MATLAB.NET.Utility
Imports MathWorks.MATLAB.NET.Arrays

Imports MagicSquareComp

Namespace MathWorks.Examples.MagicSquare
    ' <summary>
    ' The MagicSquareApp class computes a magic square of the user specified size.
    ' </summary>
    ' <remarks>
    ' args[0] - a positive integer representing the array size.
    ' </remarks>
    Class MagicSquareApp
#Region " MAIN "
        ' <summary>
        ' The main entry point for the application.
        ' </summary>
        Shared Sub Main(ByVal args() As String)

            Dim arraySize As MWNumericArray = Nothing
            Dim magicSquare As MWNumericArray = Nothing

            Try
                ' Get user specified command line arguments or set default
                If (0 <> args.Length) Then
                    arraySize = New MWNumericArray(Int32.Parse(args(0)), False)
                Else
                    arraySize = New MWNumericArray(4, False)
                End If

                ' Create the magic square object
                Dim magic As MagicSquareClass = New MagicSquareClass

                ' Compute the magic square and print the result
                magicSquare = magic.makesquare(arraySize)

                Console.WriteLine("Magic square of order {0}{1}{2}{3}", arraySize,
                    Chr(10), Chr(10), magicSquare)

                ' Convert the magic square array to a two dimensional native double array
                Dim nativeArray(,) As Double =
                    CType(magicSquare.ToArray(MWArrayComponent.Real), Double(,))

                Console.WriteLine("{0}Magic square as native array:{1}", Chr(10), Chr(10))

                ' Display the array elements:
                Dim index As Integer = arraySize.ToScalarInteger()

                For i As Integer = 0 To index - 1
                    For j As Integer = 0 To index - 1
                        Console.WriteLine("Element({0},{1})= {2}", i, j, nativeArray(i, j))
                    Next j
                Next i
            End Try
        End Sub
    End Class
End Namespace
```

```

        Console.ReadLine() 'Wait for user to exit application
    Catch exception As Exception
        Console.WriteLine("Error: {0}", exception)
    End Try
End Sub
#End Region

End Class

End Namespace

```

The application you build from this source file does the following:

- Lets you pass a dimension for the magic square from the command line.
- Converts the dimension argument to a MATLAB integer scalar value.
- Declares variables of type `MWNumericArray` to handle data required by the encapsulated `makesquare` function.

Note For information about these data conversion classes, see the *MWArray Class Library Reference*, which is also available in the `matlabroot\help\dotnetbuilder\MWArrayAPI` folder, where `matlabroot` represents your MATLAB installation folder.

- Creates an instance of the `MagicSquare` class named `magic`.
 - Calls the `makesquare` method, which belongs to the `magic` object. The `makesquare` method generates the magic square using the MATLAB `magic` function.
 - Displays the array elements on the command line.
- 2** Build the application using Visual Studio .NET.
- a** The `MagicSquareVbApp` folder contains a Visual Studio .NET project file for each example. Open the project in Visual Studio .NET for this example by double-clicking `MagicSquareVbApp.vbproj` in Windows Explorer.
 - b** Add a reference to the `MWArray` component, which is `matlabroot\toolbox\dotnetbuilder\bin\architecture\framework_version\mwarray.dll`.
 - c** If necessary, add a reference to the `MagicSquareComp` component, which is in the `distrib` subfolder.
 - d** Build and run the application in Visual Studio.NET.

Distribute Integrated .NET Applications

- “Package .NET Applications” on page 5-2
- “About the MATLAB Runtime” on page 5-3
- “Install and Configure MATLAB Runtime” on page 5-4

Package .NET Applications

1 Gather and package the following files for installation on end user computers:

- MATLAB Runtime installer

See “Install and Configure MATLAB Runtime” on page 5-4.

- MATLAB generated .NET assembly
- Executable for the application

2 Include directions for installing MATLAB Runtime.

See “Install and Configure MATLAB Runtime” on page 5-4.

About the MATLAB Runtime

In this section...

“How is the MATLAB Runtime Different from MATLAB?” on page 5-3

“Performance Considerations and the MATLAB Runtime” on page 5-3

The MATLAB Runtime is a standalone set of shared libraries, MATLAB code, and other files that enables the execution of MATLAB files on computers without an installed version of MATLAB. Applications that use artifacts built with MATLAB Compiler SDK require access to an appropriate version of the MATLAB Runtime to run.

End-users of compiled artifacts without access to MATLAB must install the MATLAB Runtime on their computers or know the location of a network-installed MATLAB Runtime. The installers generated by the compiler apps may include the MATLAB Runtime installer. If you compiled your artifact using `mcc`, you should direct your end-users to download the MATLAB Runtime installer from the website <https://www.mathworks.com/products/compiler/mcr>.

See “Install and Configure MATLAB Runtime” on page 5-4 for more information.

How is the MATLAB Runtime Different from MATLAB?

The MATLAB Runtime differs from MATLAB in several important ways:

- In the MATLAB Runtime, MATLAB files are encrypted and immutable.
- MATLAB has a desktop graphical interface. The MATLAB Runtime has all the MATLAB functionality without the graphical interface.
- The MATLAB Runtime is version-specific. You must run your applications with the version of the MATLAB Runtime associated with the version of MATLAB Compiler SDK with which it was created. For example, if you compiled an application using version 6.3 (R2016b) of MATLAB Compiler, users who do not have MATLAB installed must have version 9.1 of the MATLAB Runtime installed. Use `mcrversion` to return the version number of the MATLAB Runtime.
- The MATLAB paths in a MATLAB Runtime instance are fixed and cannot be changed. To change them, you must first customize them within MATLAB.

Performance Considerations and the MATLAB Runtime

MATLAB Compiler SDK was designed to work with a large range of applications that use the MATLAB programming language. Because of this, run-time libraries are large.

Since the MATLAB Runtime technology provides full support for the MATLAB language, including the Java programming language, starting a compiled application takes approximately the same amount of time as starting MATLAB. The amount of resources consumed by the MATLAB Runtime is necessary in order to retain the power and functionality of a full version of MATLAB.

Calls into the MATLAB Runtime are serialized so calls into the MATLAB Runtime are threadsafe. This can impact performance.

Install and Configure MATLAB Runtime

Supported Platforms: Windows, Linux, macOS

MATLAB Runtime contains the libraries needed to run MATLAB applications on a target system without a licensed copy of MATLAB.

Download MATLAB Runtime Installer

Download MATLAB Runtime using one of the following options:

- Download the MATLAB Runtime installer at the latest update level for the selected release from the website at <https://www.mathworks.com/products/compiler/matlab-runtime.html>. This option is best for end users who want to run deployed applications.
- Use the MATLAB function `compiler.runtime.download` to download the MATLAB Runtime installer matching the version and update level of MATLAB from where the command is executed. If the installer has already been downloaded to the machine, it returns the path to the MATLAB Runtime installer. If the machine is offline, it returns a URL to the MATLAB Runtime installer. This option is best for developers who want to create application installers that contain MATLAB Runtime.

Install MATLAB Runtime Interactively

To install MATLAB Runtime:

- 1 Extract the archive containing the MATLAB Runtime installer.

Platform	Steps
Windows	Unzip the MATLAB Runtime installer. Right-click the ZIP file <code>MATLAB_Runtime_R2021b_win64.zip</code> and select Extract All .
Linux	Unzip the MATLAB Runtime installer at the terminal using the <code>unzip</code> command. For example, if you are unzipping the R2021b MATLAB Runtime installer, at the terminal, type: <code>unzip MATLAB_Runtime_R2021b_glnxa64.zip</code>
macOS	Unzip the MATLAB Runtime installer at the terminal using the <code>unzip</code> command. For example, if you are unzipping the R2021b MATLAB Runtime installer, at the terminal, type: <code>unzip MATLAB_Runtime_R2021b_maci64.zip</code>

Note The release part of the installer file name (`_R2021b_`) changes from one release to the next.

- 2 Start the MATLAB Runtime installer.

Platform	Steps
Windows	Double-click the file <code>setup.exe</code> from the extracted files to start the installer.
Linux	At the terminal, type: <pre>sudo -H ./install</pre> <p>Note The <code>-H</code> option sets the <code>HOME</code> environment variable to the home directory of the root user and should be used for graphical applications such as installers.</p>
macOS	At the terminal, type: <pre>./install</pre> <p>Note You may need to enter an administrator user name and password after you run <code>./install</code>.</p>

Note If you are running the MATLAB Runtime installer on a shared folder, be aware that other users of the share may need to alter their system configuration.

- 3 When the MATLAB Runtime installer starts, it displays a dialog box. Read the information and then click **Next** to proceed with the installation.
- 4 In the **Folder Selection** dialog box, specify the folder in which you want to install MATLAB Runtime.

Note You can have multiple versions of MATLAB Runtime on your computer, but only one installation for any particular version. If you already have an existing installation, the MATLAB Runtime installer does not display the **Folder Selection** dialog box because it overwrites the existing installation in the same folder.

- 5 Confirm your choices and click **Next**.

The MATLAB Runtime installer starts copying files into the installation folder.

- 6 On Linux and macOS platforms, after copying files to your disk, the MATLAB Runtime installer displays the **Product Configuration Notes** dialog box. This dialog box contains information necessary for setting your path environment variables. Copy the path information from this dialog box, save it to a text file, and then click **Next**. For information on setting environment variables, see “Set MATLAB Runtime Path for Deployment”.
- 7 Click **Finish** to exit the installer.

The default MATLAB Runtime installation directory for **R2021b** is specified in the following table:

Operating System	MATLAB Runtime Installation Directory
Windows	C:\Program Files\MATLAB\MATLAB Runtime\v911
Linux	/usr/local/MATLAB/MATLAB_Runtime/v911
macOS	/Applications/MATLAB/MATLAB_Runtime/v911

Install MATLAB Runtime Noninteractively

To install MATLAB Runtime without having to interact with the installer dialog boxes, use one of these noninteractive modes:

- Silent — The installer runs as a background task and does not display any dialog boxes.
- Automated — The installer displays the dialog boxes but does not wait for user interaction.

When run in silent or automated mode, the MATLAB Runtime installer uses default values for installation options. You can override these values by using MATLAB Runtime installer command-line options or an installer control file.

Note When running in silent or automated mode, the installer overwrites the installation location.

Run Installer in Silent Mode

To install MATLAB Runtime in silent mode:

- 1 Extract the contents of the MATLAB Runtime installer archive to a temporary folder.
- 2 In your system command prompt, navigate to the folder where you extracted the installer.
- 3 Run the MATLAB Runtime installer, specifying the `-mode silent` and `-agreeToLicense yes` options on the command line.

Note On most platforms, the installer is located at the root of the folder into which the archive was extracted. On 64-bit Windows, the installer is located in the archive `bin` folder.

Platform	Command
Windows	<code>setup -mode silent -agreeToLicense yes</code>
Linux	<code>./install -mode silent -agreeToLicense yes</code>
macOS	<code>./install -mode silent -agreeToLicense yes</code>

Note If you do not include the `-agreeToLicense yes` option, the installer does not install MATLAB Runtime.

- 4 View a log of the installation.

On Windows systems, the MATLAB Runtime installer creates a log file named `mathworks_username.log`, where `username` is your Windows login name, in the location defined by your `TEMP` environment variable.

- 5 On Linux and macOS systems, the MATLAB Runtime installer displays the log information at the command prompt and also saves it to a file if you use the `-outputFile` option.

Customize a Noninteractive Installation

When run in one of the noninteractive modes, the installer uses the default values unless you specify otherwise. Like the MATLAB installer, the MATLAB Runtime installer accepts a number of command-line options that modify the default installation properties.

Option	Description
-destinationFolder	Specifies where MATLAB Runtime is installed.
-outputFile	Specifies where the installation log file is written.
-tmpdir	Specifies where temporary files are stored during installation. Caution The installer deletes everything inside the specified folder.
-automatedModeTimeout	Specifies how long, in milliseconds, that each dialog box is displayed when run in automatic mode.
-inputFile	Specifies an installer control file that contains your command-line options and values. Omit the dashes and put each option and value pair on a separate line.

Note The MATLAB installer archive includes an example installer control file called `installer_input.txt`. This file contains all of the options available for a full MATLAB installation. The options listed in this section are valid for the MATLAB Runtime installer.

Install MATLAB Runtime without Administrator Rights

To install MATLAB Runtime as a user without administrator rights on Windows:

- 1 Use the MATLAB Runtime installer to install it on a Windows machine where you have administrator rights.
- 2 Copy the folder where MATLAB Runtime was installed to the machine without administrator rights. You can compress the folder into a zip file for distribution.
- 3 On the machine without administrator rights, add the `<MATLAB_RUNTIME_INSTALL_DIR>\runtime\arch` directory to the user's PATH environment variable. For more information, see "Set MATLAB Runtime Path for Deployment".

Install Multiple MATLAB Runtime Versions on Single Machine

MCRInstaller supports the installation of multiple versions of MATLAB Runtime on a target machine. This capability allows applications compiled with different versions of MATLAB Runtime to execute side by side on the same machine.

If you do not want multiple MATLAB Runtime versions on the target machine, you can remove the unwanted ones. On Windows, run **Add or Remove Programs** from the Control Panel to remove a specific version. On Linux, manually delete the unwanted MATLAB Runtime directories. You can remove unwanted versions before or after installation of a more recent version of MATLAB Runtime because versions can be installed or removed in any order.

Note Installing multiple versions of MATLAB Runtime on the same machine is not supported on macOS.

Install MATLAB and MATLAB Runtime on Same Machine

To test your deployed component on your development machine, you do not need an installation of MATLAB Runtime. The MATLAB installation that you use to compile the component can act as a MATLAB Runtime replacement.

You can, however, install MATLAB Runtime for debugging purposes.

Modify Path

If you install MATLAB Runtime on a machine that already has MATLAB on it, you must adjust the system library path according to your needs.

To run deployed MATLAB code against MATLAB Runtime rather than MATLAB, ensure that your library path lists the MATLAB Runtime directories before any MATLAB directories.

For information on setting environment variables, see “Set MATLAB Runtime Path for Deployment”.

Uninstall MATLAB Runtime

The method you use to uninstall MATLAB Runtime from your computer varies depending on your platform.

Windows

- 1 Start the uninstaller.

From the Windows Start menu, search for the **Add or Remove Programs** control panel, and double-click MATLAB Runtime in the list.

You can also start the MATLAB Runtime uninstaller from the `<MATLAB_RUNTIME_INSTALL_DIR>\uninstall\bin\<arch>` folder, where `<MATLAB_RUNTIME_INSTALL_DIR>` is your MATLAB Runtime installation folder and `<arch>` is an architecture-specific folder, such as `win32` or `win64`.

- 2 Select MATLAB Runtime from the list of products in the Uninstall Products dialog box and click **Next**.
- 3 Click **Finish**.

Linux

- 1 Close all instances of MATLAB and MATLAB Runtime.
- 2 Enter this command at the Linux terminal:

```
rm -rf <MATLAB_RUNTIME_INSTALL_DIR>
```

Caution Be careful when using the `rm` command, as deleted files cannot be recovered.

macOS

- 1 Close all instances of MATLAB and MATLAB Runtime.
- 2 Navigate to your MATLAB Runtime installation folder. For example, the installation folder might be named `MATLAB_Compiler_Runtime.app` in your Applications folder.
- 3 Drag your MATLAB Runtime installation folder to the trash, and then select **Empty Trash** from the Finder menu.

See Also

`compiler.runtime.download`

More About

- [About MATLAB Runtime](#)
- [“MATLAB Runtime Startup Options”](#)
- [“Set MATLAB Runtime Path for Deployment”](#)

Distribute to End Users

- “Deploy Components to End Users” on page 6-2
- “MATLAB Runtime Run-Time Options” on page 6-5
- “MATLAB Runtime User Data Interface” on page 6-7
- “MATLAB Runtime Component Cache and Deployable Archive Embedding” on page 6-11
- “Impersonation Implementation Using ASP.NET” on page 6-12
- “Enhanced XML Documentation Files” on page 6-15

Deploy Components to End Users

In this section...

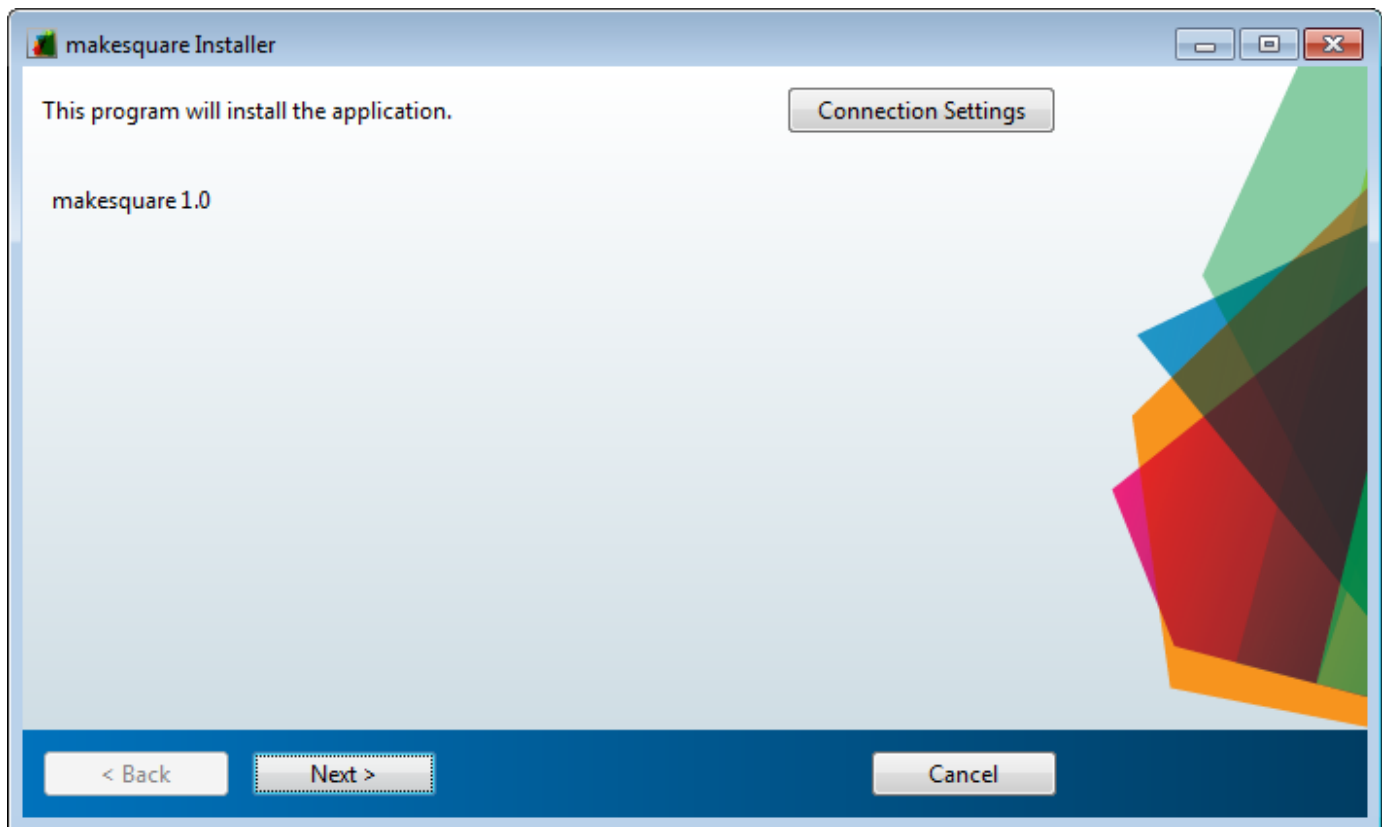
“Running the Component Installer” on page 6-2

“MATLAB Runtime” on page 6-3

Running the Component Installer

The **Library Compiler** app creates an installer for the generated .NET component. After compilation is complete, you can find this installer in the `for_redistribution` folder in your project folder. By default, the compiler names the installer `MyAppInstaller_web.exe` or `MyAppInstaller_mcr.exe`, depending on which packaging option you chose. Using the Application Information area of the Library Compiler, you can customize the look of the installer.

For example, when an end user double-clicks the component installer, the first screen identifies your component by name and version number.



By clicking **Next** on each screen, the installer leads you through the installation process. During installation, you can specify the installation folder.

The installer also automatically downloads and installs MATLAB Runtime if needed.

MATLAB Runtime

MATLAB Runtime is an execution engine made up of the same shared libraries MATLAB uses to enable execution of MATLAB files on systems without an installed version of MATLAB.

MATLAB Runtime is available to download from the web to simplify the distribution of your applications created using the MATLAB Compiler or the MATLAB Compiler SDK. Download the MATLAB Runtime from the MATLAB Runtime product page or use the `compiler.runtime.download` MATLAB function.

The MATLAB Runtime installer performs the following actions:

- 1 Install the MATLAB Runtime.
- 2 Install the component assembly in the folder from which the installer is run.
- 3 Copy the `MWArray` assembly to the Global Assembly Cache (GAC).

MATLAB Runtime Prerequisites

- 1 The MATLAB Runtime installer requires administrator privileges to run.
- 2 The version of MATLAB Runtime that runs your application on the target computer must be the same as the version of MATLAB Compiler or MATLAB Compiler SDK that built the deployed code, at the same update level or newer.
- 3 Do not install the MATLAB Runtime in MATLAB installation directories.
- 4 The MATLAB Runtime installer requires approximately 2 GB of disk space.

Add the MATLAB Runtime Installer to the Installer

This example shows how to include the MATLAB Runtime in the generated installer using one of the compiler apps. The generated installer contains all files needed to run the standalone application or shared library built with MATLAB Compiler or MATLAB Compiler SDK and properly lays them out on a target system.

- 1 On the **Packaging Options** section of the compiler interface, select one or both of the following options:
 - **Runtime downloaded from web** — This option builds an installer that downloads the MATLAB Runtime installer from the MathWorks website.
 - **Runtime included in package** — The option includes the MATLAB Runtime installer in the generated installer.
- 2 Click **Package**.
- 3 Distribute the installer to end users.

Install the MATLAB Runtime

For instructions on how to install the MATLAB Runtime on a system, see “Install and Configure MATLAB Runtime”.

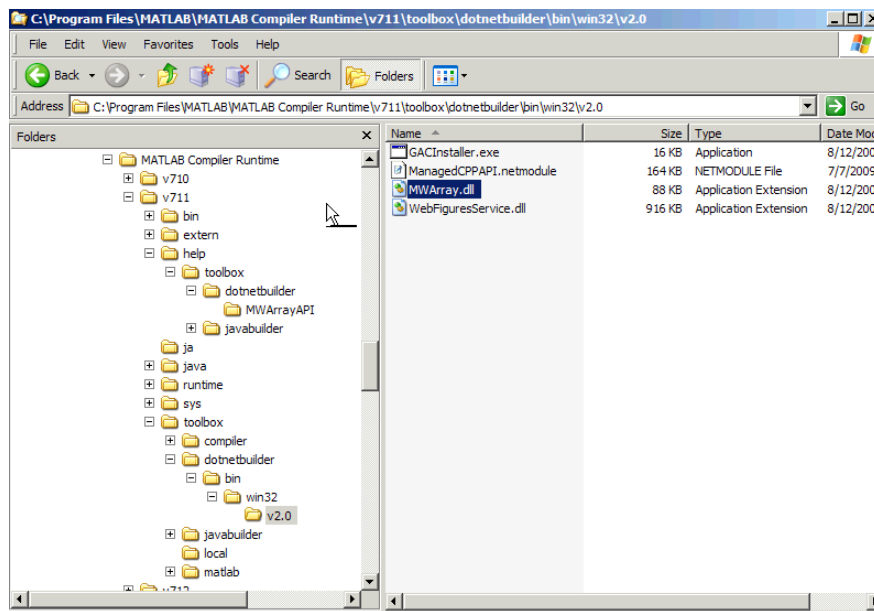
If you are given an installer containing the compiled artifacts, then MATLAB Runtime is installed along with the application or shared library. If you are given just the raw binary files, you must download and run the MATLAB Runtime installer.

Note On Windows, paths are set automatically by the installer. If you are running on a platform other than Windows, you must either modify the path on the target machine or use a shell script to launch the compiled application. Setting the paths enables your application executable to find MATLAB Runtime. For more information on setting the path, see “Set MATLAB Runtime Path for Deployment”.

Where to find the MWArray API

MATLAB Runtime also includes `MWArray.dll`, which contains an API for exchanging data between your applications and MATLAB Runtime. You can find documentation for this API in the `Help` folder of the installation.

On target machines where the MATLAB Runtime installer is run, it puts the `MWArray` assembly in `<MATLAB_RUNTIME_INSTALL_DIR>\toolbox\dotnetbuilder\bin\<ARCH>\<FRAMEWORK_VERSION>`.



Sample Directory Structure of the MATLAB Runtime Including MWArray.dll

MATLAB Runtime Run-Time Options

In this section...

“What Run-Time Options Can You Specify?” on page 6-5

“Getting MATLAB Runtime Option Values Using MWMCR” on page 6-5

What Run-Time Options Can You Specify?

You can pass the options `-nojvm` and `-logfile` to MATLAB Compiler SDK from a .NET client application using the assembly-level attributes `NOJVM` and `LOGFILE`. You retrieve values of these attributes by calling methods of the `MWMCR` class to access MATLAB Runtime attributes and state.

Getting MATLAB Runtime Option Values Using MWMCR

The `MWMCR` class provides several methods to get MATLAB Runtime option values. The following table lists methods supported by this class.

MWMCR Method	Purpose
<code>MWMCR.IsMCRInitialized()</code>	Returns <code>true</code> if the MATLAB Runtime run-time is initialized, otherwise returns <code>false</code> .
<code>MWMCR.IsMCRJVMEEnabled()</code>	Returns <code>true</code> if the MATLAB Runtime run-time is started with .NET Virtual Machine (JVM™), otherwise returns <code>false</code> .
<code>MWMCR.GetMCRLogFileName()</code>	Returns the name of the log file passed with the <code>LOGFILE</code> attribute.

Default MATLAB Runtime Options

If you pass no options, MATLAB Runtime starts with default option values:

MATLAB Runtime Run-Time Option	Default Option Values
.NET Virtual Machine (JVM)	<code>NOJVM(false)</code>
Log file usage	<code>LOGFILE(null)</code>

These options are all write-once, read-only properties.

Use the following attributes to represent the MATLAB Runtime options you want to modify.

MWMCR Attribute	Purpose
<code>NOJVM</code>	Lets users start MATLAB Runtime with or without a JVM. It takes a Boolean as input. For example, <code>NOJVM(true)</code> starts MATLAB Runtime without a JVM.
<code>LOGFILE</code>	Lets users pass the name of a log file, taking the file name as input. For example, <code>LOGFILE("logfile3.txt")</code> .

Passing MATLAB Runtime Option Values from a C# Application

Following is an example of how MATLAB Runtime option values are passed from a client-side C# application:

```
[assembly: NOJVM(false), LOGFILE("logfile3.txt")]
namespace App1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("In side main...");
            try
            {
                myclass cls = new myclass();
                cls.hello();
                Console.WriteLine("Done!!");
                Console.ReadLine();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

MATLAB Runtime User Data Interface

This feature allows data to be shared between a MATLAB Runtime instance, the MATLAB code running on that MATLAB Runtime instance, and the wrapper code that created the MATLAB Runtime instance. Through calls to the MATLAB Runtime User Data interface API, you access MATLAB Runtime data by creating a per-instance associative array of `mxArrays`, consisting of a mapping from string keys to `mxArray` values. Reasons for doing this include, but are not limited to:

- You need to supply MATLAB Runtime profile information to a client running an application created with the Parallel Computing Toolbox™ software. Profiles may be supplied (and changed) on a per-execution basis. For example, two instances of the same application may run simultaneously with different profiles.
- You want to initialize MATLAB Runtime with constant values that can be accessed by all your MATLAB applications.
- You want to set up a global workspace — a global variable or variables that MATLAB and your client can access.
- You want to store the state of any variable or group of variables.

MATLAB Compiler SDK software supports a per-MATLAB Runtime instance state access through an object-oriented API. Access to a per-instance state is optional. You can access this state by adding `setmcruserdata.m` and `getmcruserdata.m` to your deployment project or by specifying them on the command line. Alternatively, you can use a helper function to call these methods as shown in “Supplying Cluster Profiles for Parallel Computing Toolbox Applications” on page 6-7.

For more information, see “Using MATLAB Runtime User Data Interface”.

Supplying Cluster Profiles for Parallel Computing Toolbox Applications

Following is a complete example of how you can use the MATLAB Runtime User Data Interface as a mechanism to specify a cluster profile for Parallel Computing Toolbox applications.

Step 1: Write Your Parallel Computing Toolbox Code

- 1 Compile `sample_pct.m` in MATLAB.

This example code uses the cluster defined in the default profile.

The output assumes that the default profile is `local`.

```
function speedup = sample_pct (n)
warning off all;
tic
if(ischar(n))
    n=str2double(n);
end
for ii = 1:n
    (cov(sin(magic(n)+rand(n,n))));
end
time1 =toc;
parpool;
tic
parfor ii = 1:n
    (cov(sin(magic(n)+rand(n,n))));
```

```

end
time2 =toc;
disp(['Normal loop times: ' num2str(time1) ...
      ',parallel loop time: ' num2str(time2) ]);
disp(['parallel speedup: ' num2str(1/(time2/time1)) ...
      ' times faster than normal']);
delete(gcf);
disp('done');
speedup = (time1/time2);

```

- 2 Run the code as follows after changing the default profile to `local`, if needed.

```
a = sample_pct(200)
```

- 3 Verify that you get the following results:

```

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
Normal loop times: 0.7587,parallel loop time: 2.9988
parallel speedup: 0.253 times faster than normal
Parallel pool using the 'local' profile is shutting down.
done

```

```

a =

    0.2530

```

Step 2: Set the Parallel Computing Toolbox Profile

In order to compile MATLAB code to a .NET component and utilize Parallel Computing Toolbox, the `mcruserdata` must be set directly from MATLAB. There is no .NET API available to access the `MCRUserdata` as there is for C and C++ applications built with MATLAB Compiler SDK.

To set the `mcruserdata` from MATLAB, create an `init` function in your .NET class. This is a separate MATLAB function that uses `setmcruserdata` to set the Parallel Computing Toolbox profile once. You then call your other functions to utilize the Parallel Computing Toolbox functions.

Create the following `init` function:

```

function init_sample_pct
% Set the Parallel Profile:
if(isdeployed)
    [profile] = uigetfile('*.settings');
                    % let the USER select file
    setmcruserdata('ParallelProfile',
                    [profile]);
end

```

Step 3: Compile Your Function

You can compile your function from the command line by entering the following:

```

mcc -W 'dotnet:netPctComp,NetPctClass'
     init_sample_pct.m sample_pct.m -T link:lib

```

Alternately, you can use the Library Compiler app as follows:

- 1 Follow the steps in “Generate .NET Assembly and Build .NET Application” to compile your application. When the compilation finishes, a new folder (with the same name as the project) is created. This folder contains two subfolders: `distrib` and `src`.

Library Name	netPctComp
Class Name	NetPctClass
Files to Compile	sample_pct.m and init_sample_pct.m

Note If you are using the GPU feature of Parallel Computing Toolbox, you need to manually add the PTX and CU files.

If you are using the Library Compiler app, click **Add files/directories** on the **Build** tab.

If you are using the `mcc` command, use the `-a` option.

- To deploy the compiled application, copy the `for_redistribution_files_only` folder, which contains the following, to your end users.
 - `netPctComp.dll`
 - `MWArray.dll`
 - MATLAB Runtime Installer on page 6-3
 - Cluster profile

Note The end user's target machine must have access to the cluster.

Step 4: Write the .NET Driver Application

After adding references to your component and to `MWArray` in your Microsoft Visual Studio project, write the following .NET driver application to use the component, as follows. See “Integrate Simple MATLAB Function Into .NET Application” on page 3-2 for more information.

Note This example code was written using Microsoft Visual Studio 2008.

```
using System;
using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;
using netPctComp;
namespace PctNet
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                NetPctClass A = new NetPctClass();
                // Initialize the PCT set up
                A.init_sample_pct();
                double var = 300;
                MWNumericArray out1;
                MWNumericArray in1 = new MWNumericArray(300);
                out1 = (MWNumericArray)A.sample_pct(in1);
                Console.WriteLine("The speedup is {0}", out1);
                Console.ReadLine();
                // Wait for user to exit application
            }
        }
    }
}
```

```
        }  
        catch (Exception exception)  
        {  
            Console.WriteLine("Error: {0}", exception);  
        }  
    }  
}
```

The output is as follows:



A screenshot of a Windows command prompt window. The title bar is blue and contains the text "C:\ file:///C:/pct_compile/builderNE/deploy/PctNet/bin/Debug/PctNet.EXE". The main area of the window is black with white text that reads "The speedup is 2.1565". A small white cursor is visible at the bottom left of the window.

MATLAB Runtime Component Cache and Deployable Archive Embedding

Deployable archive data is automatically embedded directly in .NET Assemblies by default and extracted to a temporary folder.

Automatic embedding enables usage of the MATLAB Runtime component cache features through environment variables.

These variables allow you to specify the following:

- Define the default location where you want the deployable archive to be automatically extracted
- Add diagnostic error printing options that can be used when automatically extracting the deployable archive, for troubleshooting purposes
- Tuning the MATLAB Runtime component cache size for performance reasons.

Use the following environment variables to change these settings.

Environment Variable	Purpose	Notes
MCR_CACHE_ROOT	When set to the location of where you want the deployable archive to be extracted, this variable overrides the default per-user component cache location. This is true for embedded .ctf files only.	On macOS, this variable is ignored in MATLAB R2020a and later. The app bundle contains the files necessary for runtime.
MCR_CACHE_SIZE	When set, this variable overrides the default component cache size.	The initial limit for this variable is 32M (megabytes). This may, however, be changed after you have set the variable the first time. Edit the file <code>.max_size</code> , which resides in the file designated by running the <code>mrcachedir</code> command, with the desired cache size limit.

Note If you run `mcc` specifying conflicting wrapper and target types, the archive will not be embedded into the generated component. For example, if you run:

```
mcc -W lib:myLib -T link:exe test.m test.c
```

the generated `test.exe` will not have the archive embedded in it, as if you had specified a `-C` option to the command line.

Caution Do not extract the files within the .ctf file and place them individually under version control. Since the .ctf file contains interdependent MATLAB functions and data, the files within it must be accessed only by accessing the .ctf file. For best results, place the entire .ctf file under version control.

Impersonation Implementation Using ASP.NET

When running third-party software (for example, SQL Server®) there are times when it is necessary to use impersonation to perform Windows authentication in an ASP.NET application.

In deployed applications, impersonated credentials are passed in from IIS. However, since impersonation operates on a per-thread basis, this can sometimes present problems when processing the MATLAB Runtime thread in a multi-threaded deployed application.

Use the following examples to turn impersonation on and off in your MATLAB file, to avoid problems stemming from MATLAB Runtime thread processing issues.

Turning On Impersonation in a MATLAB MEX-file

```
#include mex.h
#include windows.h

/*
 *This mex function is called with a single int which
 *represents the user
 *identity token. We use this token to impersonate a
 *user on the interpreter
 *thread. This acts as a workaround for ASP.NET
 *applications that use
 *impersonation to pass the proper credentials
 *to SQL Server for windows
 *authentication. The function returns non zero status
 *for success, zero otherwise.
 */
void mexFunction( int          nlhs,
                  mxArray *    plhs[],
                  int          nrhs,
                  const mxArray * prhs[] )
{
    plhs[0] = mxCreateDoubleScalar(0); //return status

    HANDLE hToken =
        reinterpret_cast(*(mwSize *)mxGetData(prhs[0]));

    if(nrhs != 1)
    {
        mexErrMsgTxt("Incorrect number of input argument(s).
            Expecting 1.");
    }

    int hr;

    if(!(hr = ImpersonateLoggedOnUser(hToken)))
    {
        mexErrMsgTxt("Error impersonating.\n");
    }

    *(mxGetPr(plhs[0])) = hr;
}
```

Turning Off Impersonation in a MATLAB MEX-file

```

#include mex.h
#include windows.h

/*
 *This mex function reverts to the old identity on the
 interpreter thread **/
void mexFunction( int          nlhs,
                  mxArray *    plhs[],
                  int          nrhs,
                  const mxArray * prhs[] )
{
    if(!RevertToSelf())
    {
        mexErrMsgTxt("Failed to revert to the old
                    identity.");
    }
}

```

Code Added to Support Impersonation in ASP.NET Application

```

Monitor.Enter(someObj);

DeployedComponent.DeployedComponentClass myComp;

try
{
    System.Security.Principal.WindowsIdentity myIdentity =
        System.Security.Principal.WindowsIdentity.GetCurrent();

    //short circuit if user app is not impersonated
    if(myIdentity.isImpersonated())
    {
        myComp = new DeployedComponent.
            DeployedComponentClass ();

        //Run Users code

        MWArray[] output = myComp.impersonateUser(1,
            getToken());
    }
    else
    {
        //Run Users code
    }
}
Catch(Exception e)
{
}
finally
{
    if(myComp!=null)
        myComp.stopImpersonation();
    Monitor.Exit(someObj);
}

//

```

```
//  
//Utility method to read the token for the current user  
//and wraps it in a MWArray private MWNumericArray getToken()  
  
{  
    System.Security.Principal.WindowsIdentity myIdentity =  
        System.Security.Principal.WindowsIdentity.GetCurrent();  
  
    MWNumericArray a = null;  
  
    if (IntPtr.Size == 4)  
    {  
        int intToken = myIdentity.Token.ToInt32();  
        a = new MWNumericArray(intToken, false);  
    }  
    else  
    {  
        Int64 intToken = myIdentity.Token.ToInt64();  
        a = new MWNumericArray(intToken, false);  
    }  
    return a;  
}
```

Enhanced XML Documentation Files

Every MATLAB Compiler SDK .NET assembly produced is accompanied by a `readme.txt` file. This file outlines the contents of auto-generated documentation templates included with your built component. The documentation templates are HTML and XML files that can be read and processed by any number of third-party tools.

- `MWArray.xml` — This file describes the `MWArray` data conversion classes and their associated methods. Documentation for `MWArray` classes and their methods is available [here](#).
- `component_name.xml` — This file contains the code comments for your component. Using a third party documentation tool, you can combine this file with `MWArray.xml` to produce a complete documentation file that can be packaged with the component assembly for distribution to end users.
- `component_name_overview.html` — Optionally include this file in the generated documentation file. It contains an overview of the steps needed to access the component and how to use the data conversion classes, contained in the `MWArray` class hierarchy, to pass arguments to the generated component and return the results.

Type-Safe Interfaces, WCF, and MEF

- “Type-Safe Interfaces” on page 7-2
- “Implement Type-Safe Interface and Integrate into .NET Application” on page 7-7
- “Create Managed Extensibility Framework Plug-Ins” on page 7-12

Type-Safe Interfaces

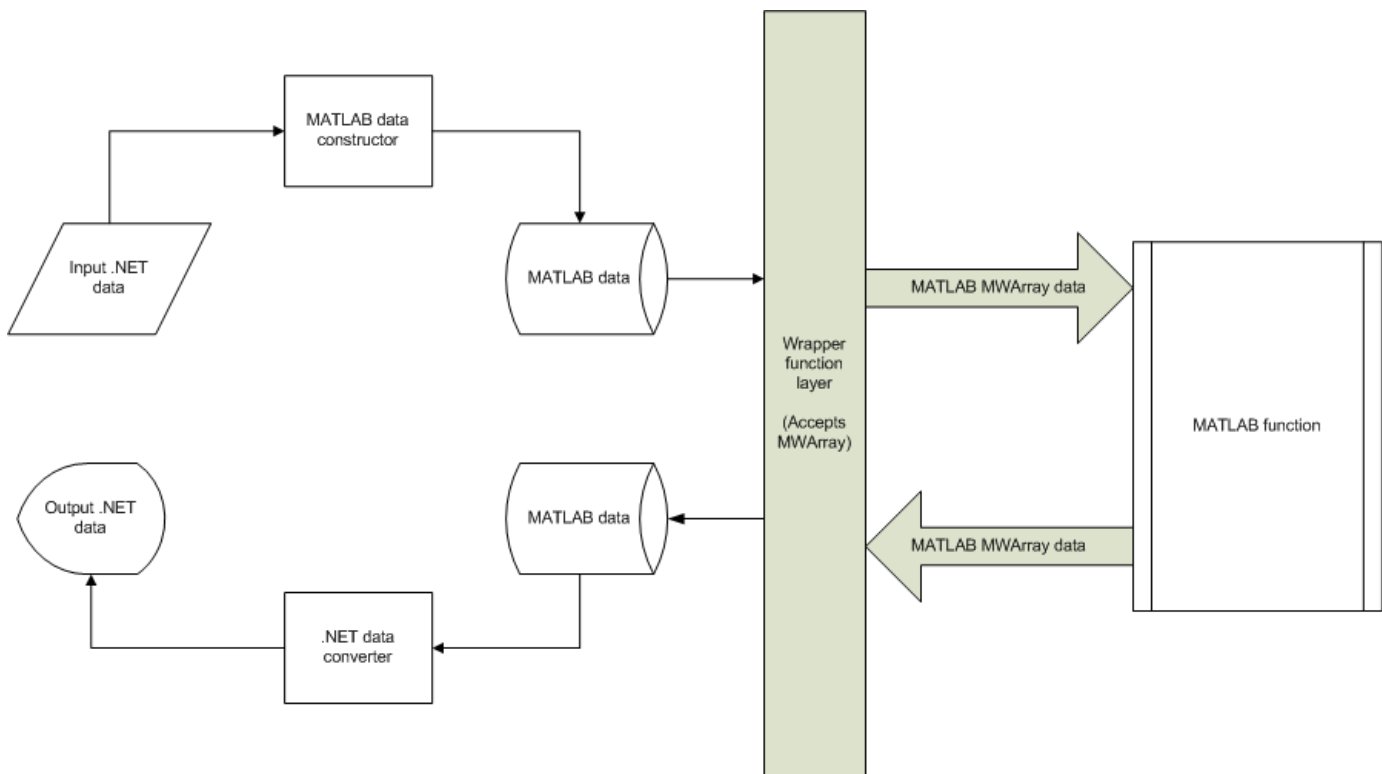
MATLAB data types are incompatible with native .NET types. To send data between your application and .NET, you perform these tasks:

- 1 Marshal data from .NET input data to a deployed function by creating an `MWArray` object from native .NET data. The `public` functions in a deployed component return `MWArray` objects.
- 2 Marshal the output MATLAB data in an `MWArray` into native .NET data by calling one of the `MWArray` marshaling methods (`ToArray()`, for example).

Manual Data Marshaling Without a Type-Safe Interface

Manually marshaling data adds complexity and potential failure points to the task of integrating deployed components into a .NET application. This is particularly true for these reasons:

- **Your application cannot detect type mismatch errors until run time.** For example, you might accidentally create an `MWArray` from a string and pass the array to a deployed function that expects a number. Because the wrapper code generated by MATLAB Compiler SDK expects an `MWArray`, the .NET compiler is unable to detect this error and the deployed function either throws an exception or returns the wrong answer.
- **Your end users must learn how to use the `MWArray` data type** or alternately mask the `MWArray` data type behind a manually written (and manually maintained) API. This introduces unwanted training time and places resource demands on a potentially overcommitted staff.

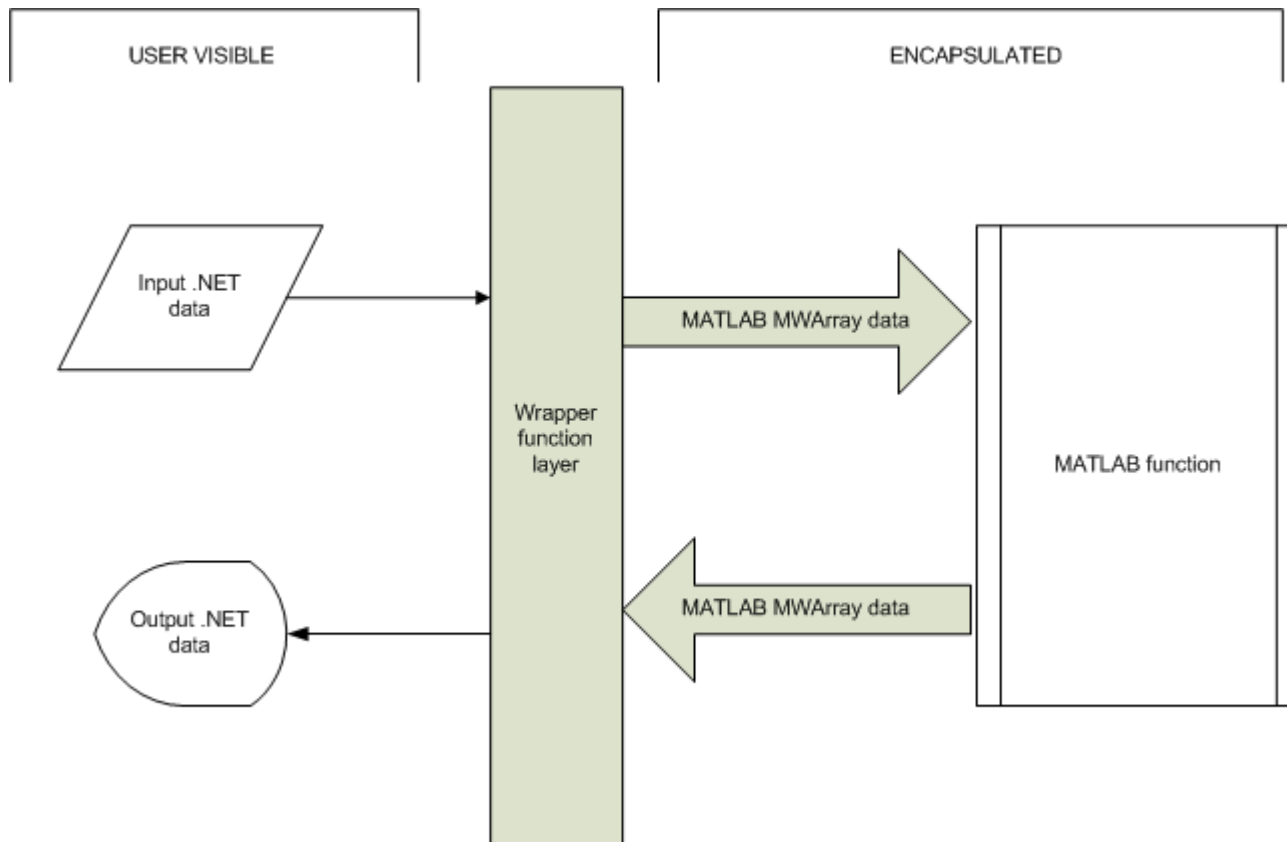


Data Marshaling Without a Type-Safe Interface

You may find `MWArray` methods more efficient when passing large data values in loops to one or more deployed functions. In such cases, creating an `MWArray` object allows you to marshal the data only once, whereas type-safe interfaces marshal inputs on every call.

Simplified Data Marshaling With a Type-Safe Interface

You can avoid performing `MWArray` data marshaling by using type-safe interfaces. Such interfaces minimize explicit type conversions by hiding the `MWArray` type from the calling application. Using type-safe interfaces allows .NET developers to work directly with familiar native data types.



Data Marshaling With a Type-Safe Interface

Some of the reasons to implement type-safe interfaces include:

- **You avoid training and coding costs** associated with teaching end users to work with the `MWArray` API.
- **You minimize cost of data you must marshal** by either placing `MWArray` objects in type-safe interfaces or by calling `MWArray` functions in the deployed MATLAB code.
- **Flexibility – you mix type-safe interfaces with manual data marshaling** to accommodate data of varying sizes and access patterns. For example, you may have a few large data objects (images, for example) that would incur excess cost to your organization if managed with a type-safe interface. By mixing type-safe interfaces and manual marshaling, smaller data types can be managed automatically with the type-safe interface and your large data can be managed on an as-needed basis.

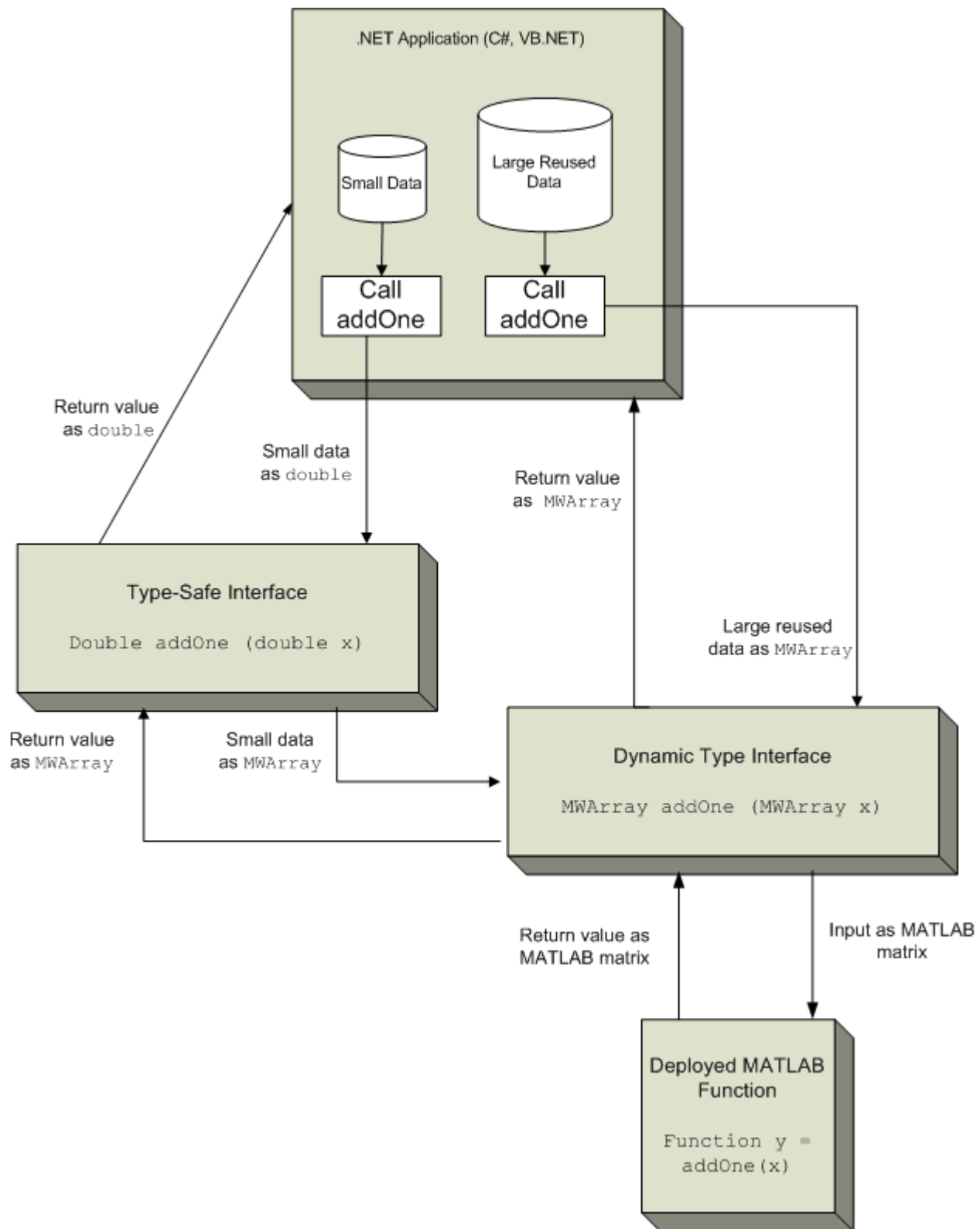
For details on implementing type-safe interfaces, see “Implement Type-Safe Interface and Integrate into .NET Application” on page 7-7.

How Type-Safe Interfaces Work

Every MATLAB Compiler SDK .NET assembly exports one or more public methods that accept and return data using `MWArray` objects. Adding a type-safe interface to a MATLAB Compiler SDK assembly creates another set of methods (with the same names) that accept and return native .NET types.

You may create multiple type-safe interface methods for a single MATLAB function. Type-safe interface methods follow the standard .NET methods for overloading.

The following figure illustrates the data paths between the .NET host application and the deployed MATLAB function through a type-safe interface.



The MATLAB function `addOne` returns its input plus one. Deploying `addOne` with a type-safe interface creates two .NET `addOne` methods:

- One that accepts and returns .NET `double`
- One that accepts and returns `MWArray`

Notice that the type-safe methods co-exist with the `MWArray` methods. Your .NET application may mix and match calls to either type of method, as appropriate.

See Also

Related Examples

- “Implement Type-Safe Interface and Integrate into .NET Application” on page 7-7

Implement Type-Safe Interface and Integrate into .NET Application

This example shows how to implement a type-safe interface and integrate it into a .NET application.

Type-safe interfaces allow you to work directly with familiar native data types instead of using the `MWArray` API. Adding a type-safe interface to a .NET assembly creates an additional set of methods that accept and return native .NET types.

Write and Test Your MATLAB Code

Create your MATLAB program and then test the code before implementing a type-safe interface. The functions in your MATLAB program must match the declarations in your native .NET interface.

For this example, save the following code as `multiply.m`. The function returns the multiplication of the inputs `x` and `y`.

```
function z = multiply(x,y)
z = x * y;
```

Test the function at the MATLAB command prompt.

```
multiply([1 4 7; 2 5 8; 3 6 9],[1 4 7; 2 5 8; 3 6 9])
```

```
ans =
```

```
    30    66   102
    36    81   126
    42    96   150
```

Implement Type-Safe Interface

After you write and test your MATLAB code, develop a .NET type-safe interface in either C# or Visual Basic. This example uses provided C# source code for the interface.

- 1 Open Microsoft Visual Studio and create a new **Class Library (.NET Framework)** project named `IMultiply`.
- 2 In the Solution Explorer window, rename the `Class1.cs` file to `IMultiply.cs`. In this file, you write source code for the type-safe interface that accesses the component.

In this example, the `IMultiply` interface is written in C# and specifies three overloads of `multiply`:

```
using System;

public interface IMultiply
{
    // Scalar multiplication
    System.Double multiply(System.Double x, System.Double y);

    // Multiply vector by a scalar, return a vector
    System.Double[] multiply(System.Double[] x, System.Double y);

    // Matrix multiplication
```

```

        System.Double[,] multiply(System.Double[,] x, System.Double[,] y);
    }

```

Each method in the interface must exactly match a deployed MATLAB function.

All methods have two inputs and one output (to match the MATLAB `multiply` function), though the parameter data type varies.

- 3 Go to **Build** and then **Configuration Manager**, and change the platform from **Any CPU** to **x64**.
- 4 Build the project with Microsoft Visual Studio.

The file `IMultiply.dll` is generated in the build folder.

This example assumes your assembly contains only `IMultiply`. Realistically, it is more likely that the type-safe interface will already be part of a compiled assembly. The assembly can be compiled even before the MATLAB function is written.

Create .NET Assembly Using Library Compiler App

Generate the type-safe interface with the .NET assembly using the **Library Compiler** app. Alternatively, if you want to create a .NET assembly from the MATLAB command window using a programmatic approach, see “Create .NET Assembly Using `compiler.build.dotNETAssembly`” on page 7-8.

- 1 Create a Library Compiler project and select **.NET Assembly** from the **Type** list.
- 2 Specify the following values:

Field	Value
Library Name	Multiply
Class Name	Arithmetic
File to Compile	multiply.m

- 3 Expand the **Additional Runtime Settings** section.
- 4 In the **Type-Safe API** section, do the following:
 - a Select **Enable Type-Safe API**.
 - b In the **Interface assembly** field, specify the location of the type-safe interface assembly `IMultiply.dll` that you built.
 - c Select the `IMultiply` interface from the **.NET interface** drop-down box.
 - d Leave the **Namespace** field blank.
 - e Specify the `Arithmetic` class in the **Wrapped Class** field.
- 5 Click the **Package** button to build the project.

Create .NET Assembly Using `compiler.build.dotNETAssembly`

As an alternative to the **Library Compiler** app, you can generate the type-safe interface using a programmatic approach using the following steps. If you have already created an assembly using the **Library Compiler**, see “Integrate .NET Assembly Into .NET Application” on page 7-9.

- 1 Build the .NET assembly using the `compiler.build.dotNETAssembly` function. Use name-value arguments to specify the assembly name and class name.

```

compiler.build.dotNETAssembly('multiply.m', ...
    'AssemblyName','Multiply', ...
    'ClassName','Arithmetic');

```


- 2 Navigate to the generated MultiplydotNETAssembly folder.
- 3 Generate the type-safe interface by using the ntswrap command from MATLAB:

```
ntswrap('-c','Multiply.Arithmetic', ...
        '-a','IMultiply.dll', ...
        '-i','IMultiply');
```

Not all arguments are compatible with each other. See ntswrap for details on all command options.

Tip If the IMultiply.dll assembly is not in the current folder, specify the full path.

This command generates the assembly ArithmeticIMultiply.dll that contains a type-safe API for the MATLAB Compiler SDK class Arithmetic in the namespace MultiplyNative.

Integrate .NET Assembly Into .NET Application

After creating your .NET assembly, you can integrate it into any .NET application. You can use this example .NET application code as a guide to write your own .NET application.

Compile the .NET program using Microsoft Visual Studio by doing the following steps:

- 1 Open Microsoft Visual Studio and create a C# **Console App (.NET Framework)** called MultiplyApp.
- 2 Copy the following source code into the generated Program.cs in your project:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MultiplyApp
{
    class Program
    {
        delegate String VectorToString<T>(IEnumerable<T> v);
        delegate IEnumerable<String> MatrixToString<T>(T[,] m);

        static void Main(string[] args)
        {
            Console.WriteLine("\nStarting application...\n");

            //Create an instance of the Type-safe API
            IMultiply m = new ArithmeticIMultiply();

            // Scalar multiplication
            double x = 17, y = 3.14159;
            double z = m.multiply(x, y);
            System.Console.WriteLine("{0} * {1} = {2}\n", x, y, z);

            // Vector times scalar
            double[] v = new double[] { 2.5, 81, 64 };
            double s = 11;
            double[] d = m.multiply(v, s);

            VectorToString<double> vec2str = (vec =>
                vec.Select(n => n.ToString()).Aggregate((str, next) => str + " " + next));

            System.Console.WriteLine("[ {0} ] * {1} = [ {2} ]\n",
                vec2str(v), s, vec2str(d));

            // Matrix multiplication
            double[,] magic = new double[,] { // 3x3 magic square
                { 8, 1, 6 },
                { 3, 5, 7 },
                { 4, 9, 2 } };
            double[,] squareSquared = m.multiply(magic, magic);
```

```

        MatrixToString<double> mat2str = mat =>
            mat.EnumerateRows<double>().Select(r => vec2str(r));

        PrintParallel(mat2str(magic), " * ".Select(c => c.ToString()),
            mat2str(magic), " = ".Select(c => c.ToString()),
            mat2str(squareSquared));

        Console.WriteLine("\nClosing application...");
    }

    public static void PrintParallel<T>(params IEnumerable<T>[] sources)
    {
        int max = sources.Select(s => s.Count()).Max();
        for (int i = 0; i < max; i++)
        {
            foreach (var src in sources)
                System.Console.Write("{0} ", src.ElementAt(i));
            System.Console.WriteLine();
        }
    }
}

public static class ArrayExtensions
{
    public static IEnumerable<IEnumerable<T>> EnumerateRows<T>(this Array a)
    {
        return Enumerable.Range(0, a.GetLength(1)).Select(row =>
            a.ToIEnumerable<T>().Skip(row * a.GetLength(0)).Take(a.GetLength(0)));
    }

    public static IEnumerable<T> ToIEnumerable<T>(this Array a)
    {
        foreach (var item in a)
            yield return (T)item;
    }
}
}

```

- 3 Add references in the project to the following files.

This reference:	Defines:
IMultiply.dll	The .NET native type interface assembly IMultiply
ArithmeticIMultiply.dll	The generated type-safe API
MultiplyNative.dll	The generated .NET assembly

Note Unlike other .NET deployment scenarios, you do not need to reference `MWArray.dll` in the server program source code. The `MWArray` data types are hidden behind the type-safe API in `ArithmeticIMultiply`.

- 4 Go to **Build** and then **Configuration Manager**, and change the platform from **Any CPU** to **x64**.
 5 Compile and run the program with Microsoft Visual Studio.

The program displays the following output:

```
Starting application...
```

```
17 * 3.14159 = 53.40703
```

```
[ 2.5 81 64 ] * 11 = [ 27.5 891 704 ]
```

```
8 1 6   8 1 6   91 67 67
3 5 7 * 3 5 7 = 67 91 67
4 9 2   4 9 2   67 67 91
```

```
Closing application...
```

Tips

- In a MATLAB function, declaration outputs appear before inputs. For example, in the `multiply` function, the output `z` appears before the inputs `x` and `y`. This ordering is not required for .NET interface functions. Inputs may appear before or after outputs, or the two may be mixed together.
- MATLAB Compiler SDK matches .NET interface functions to public MATLAB functions by function name and argument count. In the `multiply` example, both the .NET interface function and the MATLAB function must be named `multiply`, and both functions must have an equal number of arguments defined.
- The number and relative order of input and output arguments is critical.
 - In evaluating parameter order, only the order of like parameters (inputs or outputs) is considered, regardless of where they appear in the parameter list.
 - A function in the interface may have fewer inputs than its corresponding MATLAB function, but not more.
- Argument mapping occurs according to argument order rather than argument name.
- The function return value, if specified, counts as the first output.
- You must use `out` parameters for multiple outputs.
 - Alternately, the `ref` parameter can be used for `out`, as `ref` and `out` are synonymous.
- MATLAB does not support overloading of functions. Thus, all user-supplied overloads of a function with a given name will map to a function generated by MATLAB Compiler SDK.

See “[.NET Types to MATLAB Types](#)” on page 11-3 for complete guidelines in managing data conversion with type-safe interfaces.

See Also

`compiler.build.dotNETAssembly` | `ntswrap`

Related Examples

- “[Type-Safe Interfaces](#)” on page 7-2

Create Managed Extensibility Framework Plug-Ins

In this section...

“Prerequisites” on page 7-12

“Addition and Multiplication Applications with MEF” on page 7-12

“Create an MEFHost Assembly” on page 7-13

“Create a Contract Interface Assembly” on page 7-13

“Create a Metadata Attribute Assembly” on page 7-14

“Add Contract and Attributes References to MEFHost” on page 7-15

“Compile Your Code in Microsoft Visual Studio” on page 7-15

“Write MATLAB Functions for MEF Parts” on page 7-15

“Create Metadata Files” on page 7-15

“Build .NET Components from MATLAB Functions and Metadata” on page 7-16

“Install MEF Parts” on page 7-17

“Run the MEF Host Program” on page 7-17

The Managed Extensibility Framework (MEF) is a library for creating lightweight, extensible applications.

For up-to-date information regarding MEF, refer to the MSDN article “Managed Extensibility Framework.”

Prerequisites

Before running this example, keep the following in mind:

- You must be running at least Microsoft Visual Studio 2010 to create MEF applications. If you can't use Microsoft Visual Studio 2010, you can't run this example code, or any other program that uses MEF. End users do not need Microsoft Visual Studio 2010 to run applications using MEF.
- You must be running at least Microsoft .NET Framework 4.0 to use the MEF feature.
- The easiest way to use MEF is through the type-safe API.

Addition and Multiplication Applications with MEF

This MEF example application consists of an MEF host and two parts. The parts implement a very simple interface (`ICompute`) which defines three overloads of a single function (`compute`).

Each part performs simple arithmetic. In one part, the `compute` function adds one (1) to its input. In the other part, `compute` multiplies its input by two (2). The MEF host loads both parts and calls their `compute` functions twice.

To run this example, you'll create a new solution containing three projects:

- MEF host
- Contract interface assembly
- Strongly-typed metadata attribute assembly

Where To Find Example Code for MEF

Selected example code can be found, along with some Microsoft Visual Studio projects, in *matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET*. This code has been tested to be compliant with Microsoft Visual Studio 2010 running on Microsoft .NET Framework version 4.0 or higher.

Create an MEFHost Assembly

- 1 Start Microsoft Visual Studio 2010.
- 2 Click **File > New > Project**.
- 3 In the **Installed Templates** pane, click **Visual C#** to filter the list of available templates.
- 4 Select the **Console Application** template from the list.
- 5 In the **Name** field, enter MEFHost.
- 6 Click **OK**. Your project is created.
- 7 Replace the contents of the default `Program.cs` with the `MEFHost.cs` code. For information about locating example code, see “Where to Find Example Code,” above.
- 8 In the **Solution Explorer** pane, select the project **MEFHost** and right-click. Select **Add Reference**.
- 9 Click **Assemblies > Framework** and add a reference to `System.ComponentModel.Composition`.
- 10 To prevent security errors, particularly if you have a non-local installation of MATLAB, add an application configuration file to the project. This XML file instructs the MEF host to trust assemblies loaded from the network. If your project does not include this configuration file, your application fails at run time.
 - a Select the **MEFHost** project in the **Solution Explorer** pane and right-click.
 - b Click **Add > New Item**.
 - c From the list of available items, select **Application Configuration File**.
 - d Name the configuration file `App.config` and click **Add**.
 - e Replace the automatically-generated contents of `App.config` with this configuration:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <loadFromRemoteSources enabled="true" />
  </runtime>
</configuration>
```

You have finished building the first project, which builds the MEF host.

Next, you add a C# class library project for the MEF contract interface assembly.

Create a Contract Interface Assembly

- 1 In Visual Studio, click **File > New > Project**.
- 2 In the **Installed Templates** pane, click **Visual C#** to filter the list of available templates.
- 3 Select the **Class Library** template from the list.

- 4 In the **Name** field, enter `Contract`.

Note Ensure `Add to solution` is selected in the **Solution** drop-down box.

- 5 Click **OK**. Your project is created.
- 6 Replace the contents of the default `Class1.cs` with the following `ICompute` interface code:

```
namespace Contract
{
    public interface ICompute
    {
        double compute(double y);
        double[] compute(double[] y);
        double[,] compute(double[,] y);
    }
}
```

You have finished building the second project, which builds the `Contract Interface Assembly`.

Since strongly-typed metadata requires that you decorate MEF parts with a custom metadata attribute, in the next step you add a C# class library project. This project builds an attribute assembly to your MEFHost solution.

Create a Metadata Attribute Assembly

- 1 In Visual Studio, click **File > New > Project**.
- 2 In the **Installed Templates** pane, click **Visual C#** to filter the list of available templates.
- 3 Select the **Class Library** template from the list.
- 4 In the **Name** field, enter `Attribute`.

Note Ensure `Add to solution` is selected in the **Solution** drop-down box.

- 5 Click **OK**. Your project is created.
- 6 In the generated assembly code, change the namespace from `Attribute` to `MEFHost`. Your namespace code should now look like the following:

```
namespace MEFHost
{
    public class Class1
    {
    }
}
```

- 7 In the **MEFHost** namespace, replace the contents of the default class `Class1.cs` with the following code for the `ComputationTypeAttribute` class:

```
using System.ComponentModel.Composition;
[MetadataAttribute]
[AttributeUsage(AttributeTargets.Class, AllowMultiple=false)]
public class ComputationTypeAttribute: ExportAttribute
{
}
```

```

    public ComputationTypeAttribute() :
        base(typeof(Contract.ICompute)) { }
    public Operation FunctionType { get; set; }
    public double Operand { get; set; }
}

public enum Operation
{
    Plus,
    Times
}

```

- 8 Navigate to the **.NET** tab and add a reference to `System.ComponentModel.Composition.dll`.

Add Contract and Attributes References to MEFHost

Before compiling your code in Microsoft Visual Studio:

- 1 In your **MEFHost** project, add references to the **Contract** and **Attribute** projects.
- 2 In your **Attribute** project, add a reference to the **Contract** project.

Compile Your Code in Microsoft Visual Studio

Build all your code by selecting the solution name **MEFHost** in the **Solution Explorer** pane, right-clicking, and selecting **Build Solution**.

In doing so, you create the following binaries in `MEFHost/bin/Debug`:

- `Attribute.dll`
- `Contract.dll`
- `MEFHost.exe`

Write MATLAB Functions for MEF Parts

Create two MATLAB functions. Each must be named `compute` and stored in separate folders, within your Microsoft Visual Studio project:

MEFHost/Multiply/compute.m

```
function y = compute(x)
    y = x * 2;
```

MEFHost/Add/compute.m

```
function y = compute(x)
    y = x + 1;
```

Create Metadata Files

Create a metadata file for each MATLAB function.

- 1 For `MEFHost/Add/compute.m`:

a Name the metadata file MEFHost/Add/Add.metadata.

b In this file, enter the following metadata on one line:

```
[MEFHost.ComputationType(FunctionType=MEFHost.Operation.Plus, Operand=1)]
```

2 For MEFHost/Multiply/compute.m:

a Name the metadata file MEFHost/Multiply/Multiply.metadata.

b In this file, enter the following metadata on one line:

```
[MEFHost.ComputationType(FunctionType=MEFHost.Operation.Times, Operand=2)]
```

Build .NET Components from MATLAB Functions and Metadata

In this step, use the **Library Compiler** app to create .NET components from the MATLAB functions and associated metadata.

Use the information in these tables to create both Addition and Multiplication projects.

Note Since you are deploying two functions, you need to run the **Library Compiler** app *twice*, once using the Addition.prj information and once using the Multiplication.prj information.


Addition.prj

Library Name	Addition
Class Name	Add
File to Compile	MEFHost/Add/compute.m

Multiplication.prj

Library Name	Multiplication
Class Name	Multiply
File to Compile	MEFHost/Multiply/compute.m

1 Create your component, following the instructions in “Generate .NET Assembly and Build .NET Application”.

2 Modify project settings ( > **Settings**) on the **Type Safe API** tab, for whatever project you are building (Addition or Multiplication).

Project Setting	Addition.prj	Multiplication.prj
Enable Type Safe API	Checked	Checked
Interface Assembly	MEFHost/bin/Debug/Contract.dll	MEFHost/bin/Debug/Contract.dll
MEF metadata	MEFHost/Add/Add.metadata	MEFHost/Multiply/Multiply.metadata
Attribute Assembly	MEFHost/bin/Debug/Attribute.dll	MEFHost/bin/Debug/Attribute.dll

Wrapped Class	Add	Multiply
---------------	-----	----------

- 3 Click the Package button.

Install MEF Parts

The two components you have built are MEF parts. You now need to move the generated parts into the catalog directory so your application can find them:

- 1 Create a parts folder named `MEFHost/Parts`.
- 2 If necessary, modify the path argument that is passed to the `DirectoryCatalog` constructor in your MEF host program. It must match the full path to the `Parts` folder that you just created.

Note If you change the path after building the MEF host a first time, you must rebuild the MEF host again to pick up the new `Parts` path.

- 3 Copy the two `componentNative.dlls` (`Addition` and `Multiplication`) and `AddICompute.dll` and `MultiplyICompute.dll` assemblies from your into `MEFHost/Parts`.

Note You do not need to reference any of your MEF part assemblies in the MEF host program. The host program uses a `DirectoryCatalog`, which means it automatically searches for (and loads) parts that it finds in the specified folder. You can add parts at any time, without having to recompile or relink the MEF host application. You do not need to copy `Addition.dll` or `Multiplication.dll` to the `Parts` directory.

Run the MEF Host Program

MATLAB-based MEF parts require MATLAB Runtime, like all deployed MATLAB code.

Before you run your MEF host, ensure that the correct version of MATLAB Runtime is available and that `matlabroot/runtime/arch` is on your path.

- 1 From a command window, run the following. This example assumes you are running from `c:\Work`.

```
c:\Work> MEFHost\bin\Debug\MEFHost.exe
```

- 2 Verify you receive the following output:

```
8 Plus 1 = 9
9 Times 2 = 18
16 Plus 1 = 17
1.5707963267949 Times 2 = 3.14159265358979
```

Note If you receive an exception indicating that a type initializer failed, ensure that you have .NET security permissions set to allow applications to load assemblies from a network.

Windows Communications Foundation Based Components

Create Windows Communications Foundation Component

In this section...

“Write and Test Your MATLAB Code” on page 8-2
 “Implement WCF Interface” on page 8-2
 “Create .NET Assembly Using Library Compiler App” on page 8-3
 “Create .NET Assembly Using compiler.build.dotNETAssembly” on page 8-4
 “Develop Server Program Using WCF Interface” on page 8-5
 “Generate Proxy Code for Clients” on page 8-7
 “Develop Client Program Using WCF Interface” on page 8-7
 “Tips” on page 8-9

The following example shows you how to implement a Windows Communications Foundation (WCF) component using a type-safe interface and integrate it into a client-server .NET application.

For an additional example and data conversion rules regarding type-safe interfaces, see “Implement Type-Safe Interface and Integrate into .NET Application” on page 7-7.

For up-to-date information regarding WCF, see What Is Windows Communication Foundation in the Microsoft documentation.

Write and Test Your MATLAB Code

Create your MATLAB program and then test the code before implementing a type-safe interface. The functions in your MATLAB program must match the declarations in your native .NET interface.

For this example, save the following code as `addOne.m`.

```
function y = addOne(x)
% Input must be either a scalar or a matrix of single or multiple dimensions

    if ~isnumeric(x)
        error('Input must be numeric. Input was %s.', class(x));
    end
    y = x + 1;

end
```

At the MATLAB command prompt, enter `addOne([1,2,3])`.

The output is:

```
    2    3    4
```

Implement WCF Interface

After you write and test your MATLAB code, develop an interface in either C# or Visual Basic that supports the native types through the API.

- 1 Open Microsoft Visual Studio and create a new **Class Library (.NET Framework)** project named `IAddOne`.

- 2 In the Solution Explorer window within Visual Studio, rename the `Class1.cs` file to `IAddOne.cs`. In this file, write source code for the WCF interface that accesses the component.

In this example, the `IAddOne` interface is written in C# and specifies six overloads of `addOne`:

IAddOne.cs

```
using System;
using System.ServiceModel;

[ServiceContract]
public interface IAddOne
{
    [OperationContract(Name = "addOne_1")]
    int addOne(int x);

    [OperationContract(Name = "addOne_2")]
    void addOne(ref int y, int x);

    [OperationContract(Name = "addOne_3")]
    void addOne(int x, ref int y);

    [OperationContract(Name = "addOne_4")]
    System.Double addOne(System.Double x);

    [OperationContract(Name = "addOne_5")]
    System.Double[] addOne(System.Double[] x);

    [OperationContract(Name = "addOne_6")]
    System.Double[][] addOne(System.Double[][] x);
}
```

Note that in the WCF implementation of `addOne`, you decorate the methods with the `OperationContract` property. You give each method a unique operation name, which you specify with the `Name` property of `OperationContract`

Note When using WCF, your overloaded functions *must* have unique names.

Each method in the interface must exactly match a deployed MATLAB function. All methods have one input and one output (to match the MATLAB `addOne` function), though the type and position of these parameters varies.

- 3 Go to **Build > Configuration Manager** and change the platform from **Any CPU** to **x64**.
- 4 Build the project with Microsoft Visual Studio. The file `IAddOne.dll` is generated in the build folder.

Note This example assumes your assembly contains only `IAddOne`. Realistically, it is more likely that `IAddOne` will already be part of a compiled assembly. The assembly may be complete even before the MATLAB function is written.

Create .NET Assembly Using Library Compiler App

The **Library Compiler** app generates the type-safe API when you build your component, if the following options are selected.

- 1 Create a Library Compiler project and select **.NET Assembly** from the **Type** list.
- 2 Use the following values:

Library Name	AddOneComp
Class Name	Mechanism
File to Compile	addOne.m

- 3 Expand the **Additional Runtime Settings** section.

In the **Type-Safe** API section, do the following:

- 1 Select **Enable Type-Safe API**.
- 2 In the **Interface assembly** field, specify the location of the type-safe/WCF interface assembly `IAddOne.dll` that you built.
- 3 Select the `IAddOne` interface from the **.NET interface** drop-down box.

Tip If the drop-down is blank, the Library Compiler may have been unable to find any .NET interfaces in the assembly you selected.

- 4 Leave the **Namespace** and **MEF metadata** fields blank.
- 5 Specify the `Mechanism` class in the **Wrapped Class** field.
- 4 Click the **Package** button to build the project.

The file `AddOneCompNative.dll` is generated in the `for_redistribution_files_only` folder.

Create .NET Assembly Using `compiler.build.dotNETAssembly`

Note If you have already created a .NET assembly using the **Library Compiler** app, you can skip this section. However, if you want to know how to create a .NET assembly from the MATLAB command window using a programmatic approach, follow these instructions.

To generate the type-safe API with your component build using the `compiler.build.dotNETAssembly` function, complete the following steps:

- 1 Build the .NET assembly using `compiler.build.dotNETAssembly`. Use name-value arguments to specify the assembly name and class name.

```
compiler.build.dotNETAssembly('addOne.m', ...
    'AssemblyName','AddOneComp', ...
    'ClassName','Mechanism');
```

- 2 Navigate to the generated `AddOneCompdotNETAssembly` directory.
- 3 Generate the type-safe API by using the `ntswrap` command from MATLAB:

```
ntswrap('-c','AddOneComp.Mechanism', ...
    '-a','IAddOne.dll', ...
    '-i','IAddOne');
```

Not all arguments are compatible with each other. See `ntswrap` for details on all command options.

Tip If the `IAddOne.dll` assembly is not in the current folder, specify the full path.

This command generates the assembly `MechanismIAddOne.dll` that contains a type-safe API for the MATLAB Compiler SDK class `Mechanism` in the namespace `AddOneCompNative`.

Develop Server Program Using WCF Interface

Develop a server program that provides access (via the `WCFServiceContract`) to the overloads of `addOne` defined by the WCF `IAddOne` interface. The program references an `App.config` XML configuration file.

The WCF server program loads the WCF-based `addOne.Mechanism` component and makes it available to SOAP clients via the type-safe `mechanismIAddOne` interface.

Tip When writing your interface, you will be coding to handle jagged arrays, as opposed to rectangular arrays. For more information, see “Jagged Array Processing” on page 2-17.

Compile the server program using Microsoft Visual Studio by doing the following steps:

- 1 Open Microsoft Visual Studio and create a C# **Console App (.NET Framework)** called `AddOneApp`.
- 2 Copy the following source code into the generated `Program.cs` in your project:

WCF Server Program

```
using System;
using System.Text;
using System.ServiceModel;

namespace AdditionServer
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                using (ServiceHost host = new ServiceHost(typeof(MechanismIAddOne)))
                {
                    host.Open();
                    Console.WriteLine("Addition Server is up running.....");
                    Console.WriteLine("Press any key to close the service.");
                    Console.ReadLine();
                    Console.WriteLine("Closing service...");
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

- 3 Add the following configuration file `App.config` to your project. You may need to modify the listed .NET Framework version.

App.config XML file

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6.1" />
  </startup>
  <system.web>
    <compilation debug="true" />
  </system.web>
  <system.serviceModel>
```

```

<services>
  <service behaviorConfiguration=
    "Addition.ServiceBehavior" name="MechanismIAddOne">
    <endpoint
      address=""
      binding="wsHttpBinding"
      contract="IAddOne"
      name="HttpBinding" />
    <endpoint
      address=""
      binding="netTcpBinding"
      contract="IAddOne"
      name="netTcpBinding" />
    <endpoint
      address="mex"
      binding="mexHttpBinding"
      contract="IMetadataExchange"
      name="MexHttpBinding" />
    <endpoint
      address="mex"
      binding="mexTcpBinding"
      contract="IMetadataExchange"
      name="MexTCPBinding" />
    <host>
      <baseAddresses>
        <add baseAddress=
          "http://localhost:8001/AdditionServer/" />
        <add baseAddress=
          "net.tcp://localhost:8002/AdditionServer/" />
      </baseAddresses>
    </host>
  </service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior name="Addition.ServiceBehavior">
      <serviceMetadata httpGetEnabled="True" httpGetUrl=
        "http://localhost:8001/AdditionServer/mex" />
      <!-- To receive exception details in faults for debugging purposes,
        set the value below to true. Set to false before
        deployment to avoid disclosing exception
        information -->
      <serviceDebug includeExceptionDetailInFaults="True" />
    </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

- 4 Add references in the project to the following files:

This reference:	Defines:
IAddOne.dll	The .NET native type interface IAddOne
MechanismIAddOne.dll	The generated type-safe API
AddOneCompNative.dll	The generated .NET assembly

Note Unlike other .NET deployment scenarios, you do not need to reference `MWArray.dll` in the server program source code. The `MWArray` data types are hidden behind the type-safe API in `MechanismIAddOne`.

- 5 Add a reference to `System.ServiceModel`, which is listed under **Assemblies**.
 6 Go to **Build > Configuration Manager** and change the platform from **Any CPU** to **x64**.
 7 Compile and run the server program with Microsoft Visual Studio.

The program displays the following output:

```
Addition Server is up running.....
Press any key to close the service.
```

Pressing a key results in the following.

Closing service....

Generate Proxy Code for Clients

Configure your clients to communicate with the server by running the automatic proxy generation tool `svcutil.exe`. Most versions of Microsoft Visual Studio can automatically generate client proxy code from server metadata.

Caution Before you generate your client proxy code using this step, the server *must* be available and running. Otherwise, the client will not find the server.

- 1 Create a client project in Microsoft Visual Studio.
- 2 Add references by using either of these two methods.

Method 1	Method 2
<p>a In the Solutions Explorer pane, right-click References.</p> <p>b Select Add Service Reference. The Add Service Reference dialog box appears.</p> <p>c In the Address field, enter: <code>http://localhost:8001/Addition/</code></p> <hr/> <p>Note Be sure to include the / following Addition.</p> <hr/> <p>d In the Namespace field, enter <code>AdditionProxy</code>.</p> <p>e Click OK.</p>	<p>a Enter the following command from your client application directory to generate <code>AdditionProxy.cs</code>, which contains client proxy code. This command also generates configuration file <code>App.config</code>.</p> <pre>svcutil.exe /t:code http://localhost:8001/AddMaster//out:AdditionProxy.cs /config:App.config</pre> <hr/> <p>Note Enter the above command on one line, without breaks.</p> <hr/> <p>b Add <code>AdditionProxy.cs</code> and <code>App.config</code> to your client project</p>

Note When running a self-hosted application, you may encounter issues with port reservations. Use the tool `netsh` to modify your port configurations, as necessary.

Develop Client Program Using WCF Interface

At start-up, the client program connects to the `AdditionService` provided by the `Addition` WCF service. Instead of directly invoking the methods of the type-safe mechanism `IAddOne` interface, the WCF client uses the method names defined in the `OperationContract` attributes of `IAddOne`.

Compile the client program using Microsoft Visual Studio by doing the following:

- 1 Open Microsoft Visual Studio and create a C# **Console App (.NET Framework)** called `AdditionClient`.
- 2 Copy the following source code into the generated `Program.cs` in your project:

WCF Client Program

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ServiceModel;

namespace AdditionClient
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // Connect to Addition Service
                Console.WriteLine("\nConnecting to Addition Service through Http connection...");

                AdditionClient.AdditionProxy.AddOneClient Addition = new AdditionClient.AdditionProxy.AddOneClient("HttpBinding");

                Console.WriteLine("\nConnected to Addition Service...\n");

                // Output as return value
                int one = 10;
                int two = Addition.addOne_1(one);
                Console.WriteLine("addOne({0}) = {1}",
                                one, two);

                // Output: first parameter
                int i16 = 16;
                int o17 = 0;
                Addition.addOne_2(ref o17, i16);
                Console.WriteLine("addOne({0}) = {1}", i16, o17);

                // Output: second parameter
                int three = 0;
                Addition.addOne_3(two, ref three);
                Console.WriteLine("addOne({0}) = {1}",
                                two, three);

                // Scalar doubles
                System.Double i495 = 495.0;
                System.Double third =
                    Addition.addOne_4(i495);
                Console.WriteLine("addOne({0}) = {1}",
                                i495, third);

                // Vector addition
                System.Double[] i = { 30, 60, 88 };
                System.Double[] o = Addition.addOne_5(i);
                Console.WriteLine(
                    "addOne([ {0} {1} {2} ]) = [ {3} {4} {5} ]",
                    i[0], i[1], i[2], o[0], o[1], o[2]);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }

            Console.WriteLine("\nPress any key to close the client application.");
            Console.ReadLine();
            Console.WriteLine("Closing client...");
        }
    }
}

```

- 3** If you are not already referencing `System.ServiceModel`, add it to your Visual Studio project.
- 4** Go to **Build > Configuration Manager** and change the platform from **Any CPU** to **x64**.
- 5** Compile the WCF client program with Microsoft Visual Studio.
- 6** Run the program from the command line with administrator access.

The program displays the following output:

```

Connecting to Addition Service through Http connection...
Connected to Addition Service...

```

```
addOne(1) = 2
addOne(16) = 17
addOne(2) = 3
addOne(495) = 496
addOne([30 60 88]) = [31 61 89]
addOne([0 2; 3 1]) = [1 3; 4 2]
Press any key to close the client application.
```

Pressing a key results in the following:

```
Closing client....
```

Tips

- If you want to use WCF, the easiest way to do so is through the type-safe API.
- WCF and .NET remoting are not compatible in the same deployment project or component.
- This example requires both client and server to use message sizes larger than the WCF defaults. For information about changing the default message size, see the MSDN article regarding the `maxreceivedmessagesize` property.

See Also

`compiler.build.dotNETAssembly | ntswrap` | “.NET Remoting and Windows Communications Foundation” on page 9-2 | “Type-Safe Interfaces” on page 7-2

Related Examples

- “Implement Type-Safe Interface and Integrate into .NET Application” on page 7-7

.NET Remoting

- “.NET Remoting and Windows Communications Foundation” on page 9-2
- “Compare MWArray and Native .NET API for Remotable Assemblies” on page 9-4
- “Create Remotable .NET Assembly” on page 9-6
- “Access Remotable .NET Assembly Using MWArray API” on page 9-9
- “Access Remotable .NET Assembly Using Native .NET API: Magic Square” on page 9-14
- “Access Remotable .NET Assembly Using Native .NET API: Cell and Struct” on page 9-19

.NET Remoting and Windows Communications Foundation

In this section...

“.NET Remoting” on page 9-2

“Windows Communications Foundation” on page 9-2

“What’s the Difference Between WCF and .NET Remoting?” on page 9-3

.NET Remoting

Remotable .NET components allow you to access MATLAB functionality remotely as part of a distributed system consisting of multiple applications, domains, browsers, or machines.

Before you enable .NET remoting for your deployable component, be aware that you cannot enable both .NET remoting and Windows Communication Foundation.

Benefits of .NET Remoting

There are many reasons to create remotable components:

- **Cost savings** — Changes to business logic do not require you to roll out new software to every client. Instead, you can confine new updates to a small set of business servers.
- **Increased security for web applications** — .NET Remoting allows your database, for example, to reside behind one or more firewalls.
- **Software Compatibility** — Remotable components employ standard formatting protocols like SOAP (Simple Object Access Protocol), which can significantly enhance the compatibility of the component with libraries and applications.
- **Ability to run applications as Windows services** — To run as a Windows service, you must have access to a remotable component hosted by the service. Applications implemented as a Windows service provide many benefits to application developers who require an automated server running as a background process independent of a particular user account.
- **Flexibility to isolate native code binaries that were previously incompatible** — Mix native and managed code without restrictions.

Windows Communications Foundation

Windows Communication Foundation (WCF) is an application programming interface in the .NET Framework for building service-oriented applications. Servers implement multiple services that can be consumed by multiple clients. Services are loosely coupled to each other.

Services typically have a WSDL interface (Web Services Description Language), which any WCF client can use to consume the service. A WCF client connects to a service via an endpoint. Each service exposes itself via one or more endpoints. An endpoint has an address, which is a URL specifying where the endpoint can be accessed, and binding properties that specify how the data will be transferred.

For More information About WCF

For up-to-date information regarding WCF, see What Is Windows Communication Foundation in the Microsoft documentation.

What's the Difference Between WCF and .NET Remoting?

WCF is an end-to-end web service. Many of the advantages afforded by .NET Remoting—a wide selection of protocol interoperability, for instance—can be achieved with a WCF interface, in addition to having access to a richer, more flexible set of native data types. .NET Remoting can only support native objects.

WCF offers more robust choices in most every aspect of web-based development, even implementation of a Java client, for example.

Compare MWArray and Native .NET API for Remotable Assemblies

The two data conversion APIs that marshal and format data across the managed (.NET) and unmanaged (MATLAB) code boundary are `MWArray` and the native .NET API. Each API has advantages, limitations, and particular applications for which it is best suited.

The `MWArray` API, which consists of the `MWArray` class and several derived types that map to MATLAB data types, is the standard API that has been used since the introduction of MATLAB Compiler SDK. It provides full marshaling and formatting services for all basic MATLAB data types including sparse arrays, structures, and cell arrays. This API requires MATLAB Runtime to be installed on the target machine, as it makes use of several primitive MATLAB functions. For information about using this API, see “Access Remotable .NET Assembly Using `MWArray` API” on page 9-9.

The Native API was designed especially, though not exclusively, to support .NET remoting. It allows you to pass arguments and return values using standard .NET types when calling the deployed MATLAB function. Here, data marshaling is still used, but it is not explicit in the client code. This feature is especially useful for clients that access a remotable component using the native interface API, as it does not require the client machine to have MATLAB Runtime installed. In addition, as only native .NET types are used in this API, there is no need to learn the semantics of a new set of data conversion classes. This API does not directly support .NET analogs for the MATLAB structure and cell array types. For information about using this API, see “Access Remotable .NET Assembly Using Native .NET API: Magic Square” on page 9-14.

Features of the `MWArray` API Compared With the Native .NET API

	MWArray API	Native .NET API
Marshaling/formatting for all basic MATLAB types	X	
Pass arguments and return values using standard .NET types		X
Access to remotable component from client without installed MATLAB		X
Access to remotable component from client without installed MATLAB Runtime (see “Access Remotable .NET Assembly Using Native .NET API: Cell and Struct” on page 9-19).		X

Using Native .NET Structure and Cell Arrays

The MATLAB Compiler SDK native .NET API accepts standard .NET data types for inputs and outputs to MATLAB function calls.

These standard .NET data types are wrapped by the `Object` class—the base class for all .NET data types. This object representation is sufficient as long as the MATLAB functions have numeric, logical,

or string inputs or outputs. It does not work well for MATLAB data types like structure (struct) and cell arrays, since the native representation of these array types results in a multi-dimensional Object array that is difficult to comprehend or process. Instead, MATLAB Compiler SDK provides a special class hierarchy for struct and cell array representation designed to easily interface with the native .NET API. See “Access Remotable .NET Assembly Using Native .NET API: Cell and Struct” on page 9-19 for details.

See Also

Related Examples

- “Create Remotable .NET Assembly” on page 9-6
- “Create Windows Communications Foundation Component” on page 8-2

Create Remotable .NET Assembly

In this section...

“Preparation” on page 9-6

“Build Remotable Component Using Library Compiler App” on page 9-6

“Build Remotable Component Using compiler.build.dotNETAssembly” on page 9-7

“Files Generated by the Compilation Process” on page 9-7

This example shows how to create a remotable .NET assembly using MATLAB Compiler SDK.

Preparation

- 1 Decide whether you plan to use the `MWArray` API or the native .NET API. For more information, see “Compare MWArray and Native .NET API for Remotable Assemblies” on page 9-4.
 - **If using the MWArray API**, copy the following folder that ships with the MATLAB product to your working folder:

```
matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\MagicRemoteExample\MWArrayAPI\MagicSquareRemoteComp
```

After you copy the files, at the MATLAB command prompt, change the working directory (`cd`) to the new `MagicSquareRemoteComp` subfolder in your working folder.

- **If using the native .NET API**, copy the following folder that ships with the MATLAB product to your working folder:

```
matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET\MagicRemoteExample\NativeAPI\MagicSquareRemoteComp
```

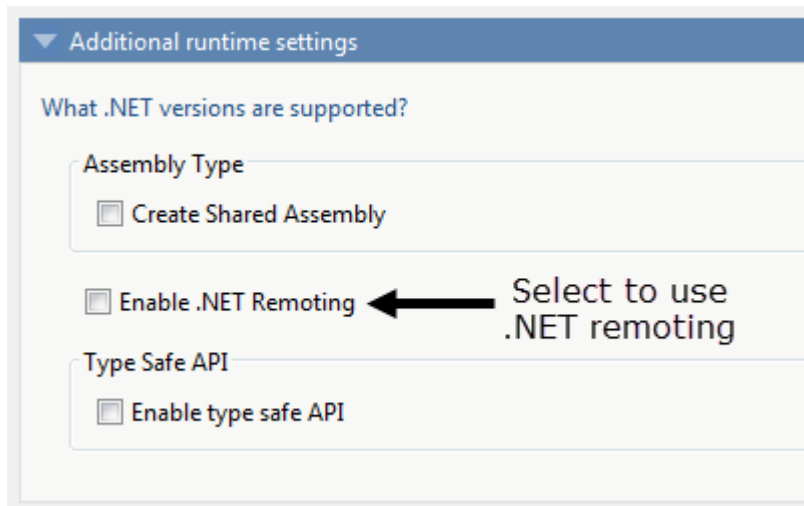
After you copy the file, at the MATLAB command prompt, change the working directory (`cd`) to the new `MagicSquareRemoteComp` subfolder in your working folder.

- 2 Write the MATLAB function. Your MATLAB code does not require any additions to support .NET remoting. The following code for the `makesquare` function is in the file `makesquare.m` in the `MagicSquareRemoteComp` subfolder:

```
function y = makesquare(x)
y = magic(x);
```

Build Remotable Component Using Library Compiler App

- 1 Click the **Library Compiler** app in the apps gallery, or type `libraryCompiler` at the MATLAB command prompt.
- 2 In the **Additional Runtime Settings** area, select **Enable .NET Remoting**.



- 3 Build the .NET component using the following values.

Field	Value
Library Name	MagicRemoteComp
Class Name	MagicClass
File to Compile	makesquare.m

For more details, see the instructions in “Generate .NET Assembly and Build .NET Application”.

Build Remotable Component Using `compiler.build.dotNETAssembly`

As an alternative to the **Library Compiler** app, you can create a .NET assembly using a programmatic approach using the following steps.

From the MATLAB prompt, issue the following command:

```
buildResults = compiler.build.dotNETAssembly('makesquare.m', ...
    'AssemblyName','MagicRemoteComp', ...
    'ClassName','MagicClass', ...
    'EnableRemoting','on');
```

Note The generated assembly does not include MATLAB Runtime or an installer. To create an installer using the `buildResults` object, see `compiler.package.installer`.

Files Generated by the Compilation Process

After compiling the components, ensure you have the following files in your `for_redistribution_files_only` folder or designated output folder:

- `MagicRemoteComp.dll` — The MArray API component implementation assembly used by the server.
- `IMagicRemoteComp.dll` — The MArray API component interface assembly used by the client .
- `MagicRemoteCompNative.dll` — The native .NET API component implementation assembly used by the server.

- `IMagicRemoteCompNative.dll` — The native .NET API component interface assembly used by the client. You do not need to install MATLAB Runtime on the client when using this interface.

See Also

`libraryCompiler` | `compiler.build.dotNETAssembly` | `mcc` | `deploytool`

Access Remotable .NET Assembly Using MWArray API

After you create the remotable component, you can set up a console server and client using the MWArray API. For more information on choosing the right API for your access needs, see “Compare MWArray and Native .NET API for Remotable Assemblies” on page 9-4.

Coding and Building the Hosting Server Application and Configuration File

The server application hosts the remote component built in “Create Remotable .NET Assembly” on page 9-6.

Build the server using the Microsoft Visual Studio project file `MagicSquareServer\MagicSquareMWServer.csproj`:

- 1 Change the references for the generated component assembly to `MagicSquareComp\for_redistribution_files_only\MagicSquareComp.dll`.
- 2 Select the appropriate build platform.
- 3 Select **Debug** or **Release** mode.
- 4 Build the `MagicSquareMWServer` project.
- 5 Supply the configuration file for the `MagicSquareMWServer`.

MagicSquareServer Code

Use the C# code for the server located in the file `MagicSquareServer\MagicSquareServer.cs`:

MagicSquareServer.cs

```
using System;
using System.Runtime.Remoting;

namespace MagicSquareServer
{
    class MagicSquareServer
    {
        static void Main(string[] args)
        {
            RemotingConfiguration.Configure(@"..\..\..\..\MagicSquareServer.exe.config");
            Console.WriteLine("Magic Square Server started...");
            Console.ReadLine();
        }
    }
}
```

This code does the following processing:

- Reads the associated configuration file to determine
 - The name of the component that it will host
 - The remoting protocol and message formatting to use
 - The lease time for the remote component
- Signals that the server is active and waits for a carriage return to be entered before terminating.

MagicSquareServer Configuration File

The configuration file for the `MagicSquareServer` is in the file `MagicSquareServer\MagicSquareServer.exe.config`. The entire configuration file, written in XML, follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="SingleCall"
          type="MagicSquareComp.MagicSquareClass, MagicSquareComp"
          objectUri="MagicSquareClass.remote" />
      </service>
      <lifetime leaseTime= "5M" renewOnCallTime="2M"
        leaseManagerPollTime="10S" />
      <channels>
        <channel ref="tcp" port="1234">
          <serverProviders>
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
        </channel>
      </channels>
    </application>
    <debug loadTypes="true"/>
  </system.runtime.remoting>
</configuration>
```

This code specifies:

- The mode in which the remote component will be accessed—in this case, single call mode
- The name of the remote component, the component assembly, and the object URI (uniform resource identifier) used to access the remote component
- The lease time for the remote component
- The remoting protocol (TCP/IP) and port number
- The message formatter (binary) and the permissions for the communication channel (full trust)
- The server debugging option

Build Client Application and Configuration File

The client application, running in a separate process, accesses the remote component running in the server application you built previously. (See “Coding and Building the Hosting Server Application and Configuration File” on page 9-9.

Next, build the remote client using the Microsoft Visual Studio project file `MagicSquareClient\MagicSquareMWClient.csproj`. This file references both the shared data conversion assembly `matlabroot\toolbox\dotnetbuilder\bin\win64\v4.0\MWArray.dll` and the generated component interface assembly `MagicSquareComp\for_redistribution_files_only\IMagicSquareComp`.

To create the remote client using Microsoft Visual Studio:

- 1 Select the appropriate build platform.
- 2 Select **Debug** or **Release** mode.
- 3 Build the `MagicSquareMWClient` project.
- 4 Supply the configuration file for the `MagicSquareMWServer`.

MagicSquareClient Code

Use the C# code for the client located in the file `MagicSquareClient\MagicSquareClient.cs`.

MagicSquareClient.cs

```

using System;
using System.Configuration;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

using System.Collections;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Remoting.Channels.Tcp;

using MathWorks.MATLAB.NET.Utility;
using MathWorks.MATLAB.NET.Arrays;

using IMagicSquareComp;

namespace MagicSquareClient
{
    class MagicSquareClient
    {
        static void Main(string[] args)
        {
            try
            {
                RemotingConfiguration.Configure
                    ("MagicSquareClient.exe.config");

                String urlServer=
                    ConfigurationSettings.AppSettings["MagicSquareServer"];

                IMagicSquareClass magicSquareComp=
                    (IMagicSquareClass)Activator.GetObject
                        (typeof(IMagicSquareClass),
                         urlServer);

                // Get user specified command line arguments or set default
                double arraySize= (0 != args.Length)
                    ? Double.Parse(args[0]) : 4;

                // Compute the magic square and print the result
                MWNumericArray magicSquare=
                    (MWNumericArray)magicSquareComp.makesquare
                        (arraySize);

                Console.WriteLine("Magic square of order {0}\n\n{1}",
                    arraySize, magicSquare);
            }
            catch (Exception exception)
            {
                Console.WriteLine(exception.Message);
            }

            Console.ReadLine();
        }
    }
}

```

This code does the following:

- The client reads the associated configuration file to get the name and location of the remotable component.
- The client instantiates the remotable object using the static `Activator.GetObject` method
- From this point, the remoting client calls methods on the remotable component exactly as it would call a local component method.

MagicSquareClient Configuration File

The configuration file for the magic square client is in the file `MagicSquareClient\MagicSquareClient.exe.config`. The configuration file, written in XML, is shown here:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>

```

```
<appSettings>
  <add key="MagicSquareServer"
    value="tcp://localhost:1234/MagicSquareClass.remote"/>
</appSettings>
<system.runtime.remoting>
  <application>
    <channels>
      <channel name="MagicSquareChannel" ref="tcp" port="0">
        <clientProviders>
          <formatter ref="binary" />
        </clientProviders>
        <serverProviders>
          <formatter ref="binary" typeFilterLevel="Full" />
        </serverProviders>
      </channel>
    </channels>
  </application>
</system.runtime.remoting>
</configuration>
```

This code specifies:

- The name of the remote component server and the remote component URI (uniform resource identifier)
- The remoting protocol (TCP/IP) and port number
- The message formatter (binary) and the permissions for the communication channel (full trust)

Start Server Application

Starting the server by doing the following:

- 1 Open a DOS or UNIX command window and cd to MagicSquareServer\bin\x86\v4.0\Debug.
- 2 Run MagicSquareServer.exe. You will see the message:
Magic Square Server started...

Start Client Application

Start the client by doing the following:

- 1 Open a DOS or UNIX command window and cd to MagicSquareClient\bin\x86\v4.0\Debug.
- 2 Run MagicSquareClient.exe. After MATLAB Runtime initializes, you should see the following output:

```
Magic square of order 4
```

```
162313
511108
97612
414151
```


See Also

Related Examples

- “Access Remotable .NET Assembly Using Native .NET API: Magic Square” on page 9-14
- “Access Remotable .NET Assembly Using Native .NET API: Cell and Struct” on page 9-19

Access Remotable .NET Assembly Using Native .NET API: Magic Square

Why Use the Native .NET API?

After the remotable component has been created, you can set up a server application and client using the native .NET API. For more information on choosing the right API for your access needs, see “Compare MArray and Native .NET API for Remotable Assemblies” on page 9-4.

Some reasons you might use the native .NET API instead of the MArray API are:

- You want to pass arguments and return values using standard .NET types, and you or your users don't work extensively with data types specific to MATLAB.
- You want to access your component from a client machine without an installed version of MATLAB.

For information on accessing your component using the MArray API, see “Access Remotable .NET Assembly Using MArray API” on page 9-9.

Coding and Building the Hosting Server Application and Configuration File

The server application will host the remote component you built in “Create Remotable .NET Assembly” on page 9-6.

The client application, running in a separate process, will access the remote component hosted by the server application. Build the server with the Microsoft Visual Studio project file `MagicSquareServer\MagicSquareServer.csproj`:

- 1 Change the reference for the generated component assembly to `MagicSquareComp\for_redistribution_files_only\MagicSquareCompNative.dll`.
- 2 Select the appropriate build platform.
- 3 Select **Debug** or **Release** mode.
- 4 Build the `MagicSquareServer` project.
- 5 Supply the configuration file for the `MagicSquareServer`.

MagicSquareServer Code

The C# code for the server is in the file `MagicSquareServer\MagicSquareServer.cs`. The `MagicSquareServer.cs` server code is shown here:

```
using System;
using System.Runtime.Remoting;

namespace MagicSquareServer
{
    class MagicSquareServer
    {
        static void Main(string[] args)
        {
            RemotingConfiguration.Configure
                (@"..\\..\\..\\..\\MagicSquareServer.exe.config");
        }
    }
}
```

```

        Console.WriteLine("Magic Square Server started...");
        Console.ReadLine();
    }
}

```

This code does the following:

- Reads the associated configuration file to determine the name of the component that it will host, the remoting protocol and message formatting to use, as well as the lease time for the remote component.
- Signals that the server is active and waits for a carriage return to be entered before terminating.

MagicSquareServer Configuration File

The configuration file for the MagicSquareServer is in the file MagicSquareServer \MagicSquareServer.exe.config. The entire configuration file, written in XML, is shown here:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="SingleCall"
          type="MagicSquareCompNative.MagicSquareClass,
            MagicSquareCompNative"
          objectUri="MagicSquareClass.remote" />
      </service>
      <lifetime leaseTime= "5M" renewOnCallTime="2M"
        leaseManagerPollTime="10S" />
      <channels>
        <channel ref="tcp" port="1234">
          <serverProviders>
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
        </channel>
      </channels>
    </application>
    <debug loadTypes="true"/>
  </system.runtime.remoting>
</configuration>

```

This code specifies:

- The mode in which the remote component will be accessed—in this case, single call mode
- The name of the remote component, the component assembly, and the object URI (uniform resource identifier) used to access the remote component
- The lease time for the remote component
- The remoting protocol (TCP/IP) and port number
- The message formatter (binary) and the permissions for the communication channel (full trust)
- The server debugging option

Coding and Building the Client Application and Configuration File

The client application, running in a separate process, accesses the remote component running in the server application built in “Coding and Building the Hosting Server Application and Configuration File” on page 9-14. Build the remote client using the Microsoft Visual Studio project file MagicSquareClient\MagicSquareClient.csproj. To create the remote client using Microsoft Visual Studio:

- 1 Change the reference for the generated component assembly to `MagicSquareComp\for_redistribution_files_only\MagicSquareCompNative.dll`.
- 2 Change the reference for the generated interface assembly to `MagicSquareComp\for_redistribution_files_only\IMagicSquareCompNative.dll`.
- 3 Select the appropriate build platform.
- 4 Select **Debug** or **Release** mode.
- 5 Build the `MagicSquareClient` project.
- 6 Supply the configuration file for the `MagicSquareServer`.

MagicSquareClient Code

The C# code for the client is in the file `MagicSquareClient\MagicSquareClient.cs`.

MagicSquareClient.cs

```
using System;
using System.Configuration;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

using System.Collections;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Remoting.Channels.Tcp;

using IMagicSquareCompNative;

namespace MagicSquareClient
{
    class MagicSquareClient
    {
        static void Main(string[] args)
        {
            try
            {
                RemotingConfiguration.Configure(
                    @"MagicSquareClient.exe.config");

                String urlServer=
                    ConfigurationSettings.AppSettings["MagicSquareServer"];

                IMagicSquareClassNative magicSquareComp=
                    (IMagicSquareClassNative)Activator.GetObject(
                        (typeof(IMagicSquareClassNative), urlServer));

                // Get user specified command line arguments or set default
                double arraySize= (0 != args.Length)
                    ? Double.Parse(args[0]) : 4;

                // Compute the magic square and print the result
                double[,] magicSquare=
                    (double[,])magicSquareComp.makesquare(arraySize);

                Console.WriteLine("Magic square of order {0}\n", arraySize);

                // Display the array elements:
                for (int i = 0; i < (int)arraySize; i++)
                    for (int j = 0; j < (int)arraySize; j++)
                        Console.WriteLine(
                            "Element({0},{1})= {2}", i, j, magicSquare[i, j]);
            }
            catch (Exception exception)
            {
                Console.WriteLine(exception.Message);
            }

            Console.ReadLine();
        }
    }
}
```

This code does the following:

- The client reads the associated configuration file to get the name and location of the remotable component.
- The client instantiates the remotable object using the static `Activator.GetObject` method
- From this point, the remoting client calls methods on the remotable component exactly as it would call a local component method.

MagicSquareClient Configuration File

The configuration file for the magic square client is in the file `MagicSquareClient\MagicSquareClient.exe.config`. The configuration file, written in XML, is shown here:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="MagicSquareServer"
        value="tcp://localhost:1234/MagicSquareClass.remote"/>
  </appSettings>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel name="MagicSquareChannel" ref="tcp" port="0">
          <clientProviders>
            <formatter ref="binary" />
          </clientProviders>
          <serverProviders>
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

This code specifies:

- The name of the remote component server and the remote component URI (uniform resource identifier)
- The remoting protocol (TCP/IP) and port number
- The message formatter (binary) and the permissions for the communication channel (full trust)

Starting the Server Application

Start the server by doing the following:

- 1 Open a DOS or UNIX command window and `cd` to `MagicSquareServer\bin\x86\v4.0\Debug`.
- 2 Run `MagicSquareServer.exe`. You will see the message:
Magic Square Server started...

Starting the Client Application

Start the client by doing the following:

- 1 Open a DOS or UNIX command window and `cd` to `MagicSquareClient\bin\x86\v4.0\Debug`.

- 2 Run `MagicSquareClient.exe`. After MATLAB Runtime initializes, you should see the following output:

Magic square of order 4

```
Element(0,0)= 16  
Element(0,1)= 2  
Element(0,2)= 3  
Element(0,3)= 13  
Element(1,0)= 5  
Element(1,1)= 11  
Element(1,2)= 10  
Element(1,3)= 8  
Element(2,0)= 9  
Element(2,1)= 7  
Element(2,2)= 6  
Element(2,3)= 12  
Element(3,0)= 4  
Element(3,1)= 14  
Element(3,2)= 15  
Element(3,3)= 1
```

Access Remotable .NET Assembly Using Native .NET API: Cell and Struct

Why Use the .NET API With Cell Arrays and Structs?

Using .NET representations of MATLAB struct and cell arrays is recommended if both of these are true:

- You have MATLAB functions on a server with MATLAB struct or cell data types as inputs or outputs
- You do not want or need to install MATLAB Runtime on your client machines

The native `MWArray`, `MWStructArray`, and `MWCellArray` classes are members of the `MathWorks.MATLAB.NET.Arrays.native` namespace.

The class names in this namespace are identical to the class names in `MathWorks.MATLAB.NET.Arrays`. The difference is that the native representations of struct and cell arrays have no methods or properties that require MATLAB Runtime.

The `matlabroot\toolbox\dotnetbuilder\Examples\VSVersion\NET` folder has example solutions you can practice building. The `NativeStructCellExample` folder contains native struct and cell examples.

Building Your Component

This example demonstrates how to deploy a remotable component using native struct and cell arrays. Before you set up the remotable client and server code, build a remotable component using the instructions in “Create Remotable .NET Assembly” on page 9-6.

The Native .NET Cell and Struct Example

The server application hosts the remote component.

The client application, running in a separate process, accesses the remote component hosted by the server application. Build the server with the Microsoft Visual Studio project file `NativeStructCellServer.csproj`:

- 1 Change the references for the generated component assembly to `component_name\for_redistribution_files_only\component_nameNative.dll`.
- 2 Select the appropriate build platform.
- 3 Select **Debug** or **Release** mode.
- 4 Build the `NativeStructCellServer` project.
- 5 Supply the configuration file for the `NativeStructCellServer`. The C# code for the server is in the file `NativeStructCellServer.cs`:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Remoting;

namespace NativeStructCellServer
```

```

{
    class NativeStructCellServer
    {
        static void Main(string[] args)
        {
            RemotingConfiguration.Configure(
                @"NativeStructCellServer.exe.config");

            Console.WriteLine("NativeStructCell Server started...");

            Console.ReadLine();
        }
    }
}

```

This code reads the associated configuration file to determine:

- Name of the component to host
- Remoting protocol and message formatting to use
- Lease time for the remote component

In addition, the code also signals that the server is active and waits for a carriage return before terminating.

Coding and Building the Client Application and Configuration File

The client application, running in a separate process, accesses the remote component running in the server application built in “The Native .NET Cell and Struct Example” on page 9-19. Build the remote client using the Microsoft Visual Studio project file `NativeStructCellClient\NativeStructCellClient.csproj`. To create the remote client using Microsoft Visual Studio:

- 1 Change the references for the generated component assembly to *component_name\for_redistribution_files_only\component_nameNative.dll*.
- 2 Change the references for the generated interface assembly to *component_name\for_redistribution_files_only\Icomponent_nameNative.dll*.
- 3 Select the appropriate build platform.
- 4 Select **Debug** or **Release** mode.
- 5 Build the `NativeStructCellClient` project.
- 6 Supply the configuration file for the `NativeStructCellClient`.

NativeStructCellClient Code

The C# code for the client is in the file `NativeStructCellClient\NativeStructCellClient.cs`:

NativeStructCellClient.cs

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Remoting;
using System.Configuration;
using MathWorks.MATLAB.NET.Arrays.native;

using INativeStructCellCompNative;

// This is a simple example that demonstrates the use
// of MathWorks.MATLAB.NET.Arrays.native package.
namespace NativeStructCellClient
{

```



```

class NativeStructCellClient
{
    static void Main(string[] args)
    {
        try
        {
            RemotingConfiguration.Configure(
                @"NativeStructCellClient.exe.config");
            String urlServer =
                ConfigurationSettings.AppSettings["NativeStructCellServer"];
            INativeStructCellClassNative nativeStructCell =
                (INativeStructCellClassNative)Activator.GetObject(typeof
                    (INativeStructCellClassNative), urlServer);

            MWCellArray field_names = new MWCellArray(1, 2);
            field_names[1, 1] = "Name";
            field_names[1, 2] = "Address";

            Object[] o = nativeStructCell.createEmptyStruct(1, field_names);
            MWStructArray S1 = (MWStructArray)o[0];
            Console.WriteLine("\nEVENT 2: Initialized structure as
                received in client applications:\n\n{0}", S1);

            //Convert "Name" value from char[,] to a string since there's
            //no MWCharArray constructor on server that accepts
            //char[,] as input.
            char c = ((char[,])S1["Name"])[0, 0];
            S1["Name"] = c.ToString();

            MWStructArray address = new MWStructArray(new int[] { 1, 1 },
                new String[] { "Street", "City", "State", "Zip" });
            address["Street", 1] = "3, Apple Hill Drive";
            address["City", 1] = "Natick";
            address["State", 1] = "MA";
            address["Zip", 1] = "01760";

            Console.WriteLine("\nUpdating the 'Address' field to
                :\n\n{0}", address);
            Console.WriteLine("\n#####\n");
            S1["Address",1] = address;

            Object[] o1 = nativeStructCell.updateField(1, S1, "Name");
            MWStructArray S2 = (MWStructArray)o1[0];

            Console.WriteLine("\nEVENT 5: Final structure as
                received by client:\n\n{0}", S2);
            Console.WriteLine("\nAddress field: \n\n{0}", S2["Address",1]);
            Console.WriteLine("\n#####\n");
        }
        catch (Exception exception)
        {
            Console.WriteLine(exception.Message);
        }
        Console.ReadLine();
    }
}

```

This code does the following:

- The client reads the associated configuration file to get the name and location of the remotable component.
- The client instantiates the remotable object using the static `Activator.GetObject` method
- From this point, the remoting client calls methods on the remotable component exactly as it would call a local component method.

NativeStructCellClient Configuration File

The configuration file for the `NativeStructCellClient` is in the file `NativeStructCellClient\NativeStructCellClient.exe.config`:

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>

```

```

    <add key="NativeStructCellServer" value=
        "tcp://localhost:1236/NativeStructCellClass.remote"/>
</appSettings>
<system.runtime.remoting>
  <application>
    <channels>
      <channel name="NativeStructCellChannel" ref="tcp" port="0">
        <clientProviders>
          <formatter ref="binary" />
        </clientProviders>
        <serverProviders>
          <formatter ref="binary" typeFilterLevel="Full" />
        </serverProviders>
      </channel>
    </channels>
  </application>
</system.runtime.remoting>
</configuration>

```

This code specifies:

- Name of the remote component server and the remote component URI (uniform resource identifier)
- Remoting protocol (TCP/IP) and port number
- Message formatter (binary) and the permissions for the communication channel (full trust)

Starting the Server Application

Start the server by doing the following:

- 1 Open a DOS or UNIX command window and cd to NativeStructCellServer\bin\x86\v4.0\Debug.
- 2 Run NativeStructCellServer.exe. The following output appears:

```

EVENT 1: Initializing the structure on server and sending
         it to client:
         Initialized empty structure:

```

```

         Name: ' '
         Address: []

```

```

#####

```

```

EVENT 3: Partially initialized structure as
         received by server:

```

```

         Name: ' '
         Address: [1x1 struct]

```

```

Address field as initialized from the client:

```

```

         Street: '3, Apple Hill Drive'
         City: 'Natick'
         State: 'MA'
         Zip: '01760'

```

```

#####

```

```
EVENT 4: Updating 'Name' field before sending the
         structure back to the client:
```

```
    Name: 'The MathWorks'
  Address: [1x1 struct]
```

```
#####
```

Starting the Client Application

Start the client by doing the following:

- 1 Open a DOS or UNIX command window and cd to NativeStructCellClient\bin\x86\v4.0\Debug.
- 2 Run NativeStructCellClient.exe. After MATLAB Runtime initializes, the following output appears:

```
EVENT 2: Initialized structure as
         received in client applications:
```

```
1x1 struct array with fields:
    Name
  Address
```

```
Updating the 'Address' field to :
```

```
1x1 struct array with fields:
    Street
    City
    State
    Zip
```

```
#####
```

```
EVENT 5: Final structure as received by client:
```

```
1x1 struct array with fields:
    Name
  Address
```

```
Address field:
```

```
1x1 struct array with fields:
    Street
    City
    State
    Zip
```

```
#####
```

Coding and Building the Client Application and Configuration File with the Native MWArray, MWStructArray, and MWCellArray Classes

createEmptyStruct.m

Initialize the structure on the server and send it to the client with the following MATLAB code:

```
function PartialStruct = createEmptyStruct(field_names)
fprintf('EVENT 1: Initializing the structure on server
        and sending it to client:\n');

PartialStruct = struct(field_names{1}, ' ', field_names{2}, []);

fprintf('        Initialized empty structure:\n\n');
disp(PartialStruct);
fprintf('\n#####\n');
```

updateField.m

Receive the partially updated structure from the client and add more data to it, before passing it back to the client, with the following MATLAB code:

```
function FinalStruct = updateField(st, field_name)

fprintf('\nEVENT 3: Partially initialized structure as
        received by server:\n\n');

disp(st);
fprintf('Address field as initialized from the client:\n\n');
disp(st.Address);
fprintf('#####\n');

fprintf(['\nEVENT 4: Updating ''', field_name, ''
        field before sending the structure back to the client:\n\n']);
st.(field_name) = 'The MathWorks';
FinalStruct = st;
disp(FinalStruct);
fprintf('\n#####\n');
```

NativeStructCellClient.cs

Create the client C# code:

Note In this case, you do not need MATLAB Runtime on the system path.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Remoting;
using System.Configuration;
using MathWorks.MATLAB.NET.Arrays.native;

using INativeStructCellCompNative;

// This is a simple example that demonstrates the use of
// MathWorks.MATLAB.NET.Arrays.native package.
namespace NativeStructCellClient
{
    class NativeStructCellClient
    {
        static void Main(string[] args)
        {
            try
            {
```

```

RemotingConfiguration.Configure
    (@"NativeStructCellClient.exe.config");
String urlServer =
    ConfigurationSettings.AppSettings[
        "NativeStructCellServer"];
INativeStructCellClassNative nativeStructCell =
    (INativeStructCellClassNative)Activator.GetObject(typeof
        (INativeStructCellClassNative),
        urlServer);

MWCellArray field_names = new MWCellArray(1, 2);
field_names[1, 1] = "Name";
field_names[1, 2] = "Address";

Object[] o = nativeStructCell.createEmptyStruct(1,field_names);
MWStructArray S1 = (MWStructArray)o[0];
Console.WriteLine("\nEVENT 2: Initialized structure as received
    in client applications:\n\n{0}" , S1);

//Convert "Name" value from char[,] to a string since
// there's no MWCharArray constructor
// on server that accepts char[,] as input.
char c = ((char[,])S1["Name"])[0, 0];
S1["Name"] = c.ToString();

MWStructArray address =
    new MWStructArray(new int[] { 1, 1 },
        new String[] { "Street", "City", "State", "Zip" });
address["Street", 1] = "3, Apple Hill Drive";
address["City", 1] = "Natick";
address["State", 1] = "MA";
address["Zip", 1] = "01760";

Console.WriteLine("\nUpdating the
    'Address' field to :\n\n{0}", address);
Console.WriteLine("\n#####\n");
S1["Address",1] = address;

Object[] o1 = nativeStructCell.updateField(1, S1, "Name");
MWStructArray S2 = (MWStructArray)o1[0];

Console.WriteLine("\nEVENT 5: Final structure as received by
    client:\n\n{0}" , S2);
Console.WriteLine("\nAddress field: \n\n{0}" , S2["Address",1]);
Console.WriteLine("\n#####\n");
}
catch (Exception exception)
{
    Console.WriteLine(exception.Message);
}
Console.ReadLine();
}
}
}

```

NativeStructCellServer.cs

Create the server C# code:

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Remoting;

namespace NativeStructCellServer
{
    class NativeStructCellServer
    {
        static void Main(string[] args)
        {
            RemotingConfiguration.Configure(
                @"NativeStructCellServer.exe.config");
        }
    }
}

```

```
        Console.WriteLine("NativeStructCell Server started...");  
        Console.ReadLine();  
    }  
}
```

Troubleshooting

- “Failure to Find MATLAB Runtime Files” on page 10-2
- “Failure to Find MATLAB Classes” on page 10-3
- “Diagnostic Messages” on page 10-4

Failure to Find MATLAB Runtime Files

If your application generates a diagnostic message indicating that a module cannot be found, it could be that MATLAB Runtime is not properly located on your path. You fix this problem by ensuring that the MATLAB Runtime files are on your application path.

On a system with MATLAB installed, *matlabroot*\runtime*arch* must be on your system path ahead of any other MATLAB installations.

- *matlabroot* is your root MATLAB folder.
- *arch* is the architecture of your computer such as win64 for a 64-bit Windows computer.

On a system with MATLAB Runtime installed, *mcr_root**ver*\runtime*arch* is on your system path.

- *mcr_root* is your root MATLAB Runtime folder. *ver* is the version number of MATLAB Runtime.
- *arch* is the architecture of your computer such as win64 for a 64-bit Windows computer.

Failure to Find MATLAB Classes

If your application generates a diagnostic message indicating that `Mathworks.MATLAB.NET.name` cannot be found, it could be that you need to reference the `MWArray.dll` assembly in your Visual Studio project.

The `MWArray.dll` assembly is located at `matlabroot\toolbox\dotnetbuilder\bin\arch\v4.0`.

- `matlabroot` is your root MATLAB or MATLAB Runtime folder.
- `arch` is the architecture of your computer, such as `win64` for a 64-bit Windows computer.

Diagnostic Messages

The following table shows diagnostic messages you might encounter, probable causes for the message, and suggested solutions.

See the following table for information about some diagnostic messages.

Diagnostic Messages and Suggested Solutions

Message	Probable Cause	Suggested Solution
LoadLibrary("component_name_1_0.dll") failed - The specified module could not be found.	You may get this error message while registering the project DLL from the DOS prompt. This can occur if the MATLAB Runtime is not on the system path.	See "Failure to Find MATLAB Runtime Files" on page 10-2.
Error in component_name.class_name.x: Error getting data conversion options.	This is often caused by mwcomutil.dll not being registered.	<ol style="list-style-type: none"> 1 Open a DOS window. 2 Change folders to <i>matlabroot</i> \bin\win64. 3 Run the following command: mwregsvr mwcomutil.dll <p>(<i>matlabroot</i> is your root MATLAB folder.)</p>
Error in VBAProject: ActiveX component can't create object.	<ul style="list-style-type: none"> • Project DLL is not registered. • An incompatible MATLAB DLL exists somewhere on the system path. 	<p>If the DLL is not registered,</p> <ol style="list-style-type: none"> 1 Open a DOS window. 2 Change folders to <i>projectdir</i> \distrib. 3 Run the following command: mwregsvr <i>projectdll.dll</i> <p>(<i>projectdir</i> represents the location of your project files).</p>
object ref not set to instance of an object	This occurs when an object that has not been instantiated is called	Instantiate the object.
Error in VBAProject: Automation error The specified module could not be found.	This usually occurs if MATLAB is not on the system path.	See "Failure to Find MATLAB Runtime Files" on page 10-2.
Showing a modal dialog box or form when the application is not running in UserInteractive mode is not a valid operation. Specify the ServiceNotification or DefaultDesktopOnly style to display a notification from a service application.	<p>This warning occurs when ASP.NET code tries to bring up a dialog box.</p> <p>If occurs because getframe() makes the figure window visible before performing the capture and thus fails when running in IIS. msgbox() calls in MATLAB code cause the warning to appear also.</p>	<p>Work around this problem by doing the following:</p> <ol style="list-style-type: none"> 1 Open the Windows Control Panel. 2 Open Services. 3 From the list of services, select and open the IIS Admin service. 4 In the Properties dialog, on the Log On tab, select Local System Account. 5 Select the option Allow Service to Interact with Desktop.

Enhanced Error Diagnostics Using mstack Trace

Use this enhanced diagnostic feature to troubleshoot problems that occur specifically during MATLAB code execution.

To implement this feature, use .NET exception handling to invoke the MATLAB function inside of the .NET application, as demonstrated in this try-catch code block:

```
try
{
Magic magic = new Magic();
magic.callmakeerror();
}
catch(Exception ex)
{
Console.WriteLine("Error: {0}", exception);
}
```

When an error occurs, the MATLAB code stack trace is printed before the Microsoft .NET application stack trace, as follows:

```
... MATLAB code Stack Trace ...
    at
file C:\work\MagicDemoCSharpApp\bin\Debug\
CallldmakeerrComp_mcr\compiler\g388611\ca
thy\MagicDemoComp\dmakeerror.m,name
dmakeerror_error2,line at 14.
    at
file C:\work\MagicDemoCSharpApp\bin\Debug\
CallldmakeerrComp_mcr\compiler\work\MagicDemoComp\dmakeerror.m,name
dmakeerror_error1,line at 11.
    at
file C:\work\MagicDemoCSharpApp\bin\Debug\
CallldmakeerrComp_mcr\compiler\work\MagicDemoComp\dmakeerror.m,name dmakeerror,line at 4.
    at
file C:\work\MagicDemoCSharpApp\bin\Debug\
CallldmakeerrComp_mcr\compiler\work\MagicDemoComp\callldmakeerror.m,name
callldmakeerror,line at 2.

... .Application Stack Trace ...
    at MathWorks.MATLAB.NET.Utility.MWMCR.EvaluateFunction
(String functionName, Int32 numArgsOut, Int
32 numArgsIn, MWArray[] argsIn)
    at MathWorks.MATLAB.NET.Utility.MWMCR.EvaluateFunction
(Int32 numArgsOut, String functionName, MWA
rray[] argsIn)
    at CallldmakeerrComp.Callldmakeerr.callldmakeerror() in
C:\work\MagicDemoComp\src\
Callldmakeerr.cs:line 140
    at MathWorks.Demo.MagicSquareApp.MagicDemoApp.Main(String[]
args) in C:\work\Ma
gicDemoCSharpApp\MagicDemoApp.cs:line 52
```

Reference Information

- “Rules for Data Conversion Between .NET and MATLAB” on page 11-2
- “Interfaces Generated by the MATLAB Compiler SDK” on page 11-9

Rules for Data Conversion Between .NET and MATLAB

In this section...
"Managed .NET Types to MATLAB Arrays" on page 11-2
"MATLAB Arrays to Managed .NET Types" on page 11-2
".NET Types to MATLAB Types" on page 11-3
"Character and String Conversion" on page 11-7
"Unsupported MATLAB Array Types" on page 11-7

Note The conversion rules listed in the following tables apply to scalars, vectors, matrices, and multidimensional arrays of the native types listed.

Managed .NET Types to MATLAB Arrays

The following table lists the data conversion rules used when converting native .NET types to MATLAB arrays.

Conversion Rules: Managed Types to MATLAB Arrays

Native .NET Type	MATLAB Array	Comments
System.Double	double	—
System.Single	single	Available only when the <code>makeDouble</code> constructor argument is set to <code>false</code> . The default is <code>true</code> , which creates a MATLAB double type.
System.Int64	int64	
System.Int32	int32	
System.Int16	int16	
System.Byte	int8	
System.String	char	None
System.Boolean	logical	None

MATLAB Arrays to Managed .NET Types

The following table lists the data conversion rules used when converting MATLAB arrays to native .NET types.

Conversion Rules: MATLAB Arrays to Managed Types

MATLAB Type	.NET Type (Primitive)	.NET Type (Class)	Comments
cell	N/A	MWCellArray	Cell and struct arrays have no corresponding .NET type.
structure	N/A	MWStructArray	
char	System.String	MWCharArray	
double	System.Double	MWNumericArray	Default is type double.
single	System.Single	MWNumericArray	
uint64	System.Int64	MWNumericArray	Conversion to the equivalent unsigned type is not supported
uint32	System.Int32	MWNumericArray	Conversion to the equivalent unsigned type is not supported
uint16	System.Int16	MWNumericArray	Conversion to the equivalent unsigned type is not supported
uint8	System.Byte	MWNumericArray	None
logical	System.Boolean	MWLogicalArray	None
Function handle	N/A	N/A	None
Object	N/A	N/A	None

.NET Types to MATLAB Types

In order to create .NET interfaces that describe the type-safe API of a MATLAB Compiler SDK generated component, you must decide on the .NET types used for input and output parameters.

When choosing input types, consider how .NET inputs become MATLAB types. When choosing output types, consider the inverse conversion

The following tables list the data conversion results and rules used to convert .NET types to MATLAB arrays and MATLAB arrays to .NET types.

Note Invalid conversions result in a thrown `ArgumentException`

Conversion Results: .NET Types to MATLAB Types

.NET Type	Converts to MATLAB Type
NumericType <ul style="list-style-type: none"> System.Double System.Single System.Byte System.Int16 System.Int32 System.Int64 System.Int64 	numeric
System.Boolean	logical
System.Char	char
System.String	
NumericType[N]	NumericType[1,N]
NumericType[P _n ,...,P ₁ ,M,N]	NumericType[M,N,P ₁ ,...,P _n]
System.Boolean[N]	logical [1,N]
System.Boolean[P _n ,...,P ₁ ,M,N]	logical [M,N,P ₁ ,...,P _n]
System.Char[N]	char [1,N]
System.Char[P _n ,...,P ₁ ,M,N]	char [M,N,P ₁ ,...,P _n]
System.String[N]	char [N,max_string_length]
System.String[P _n ,...,P ₁ ,N]	char [N,max_string_length, P ₁ ,...,P _n]
Scalar .NET struct	MATLAB struct constructed from public instance fields of the .NET struct
.NET struct [N]	MATLAB struct [1,N] where each element is constructed from public instance fields of the .NET struct
.NET struct [M,N]	MATLAB struct [M,N] where each element is constructed from public instance fields of the .NET struct
native.MWStructArray	struct
native.MWCellArray	cell
Hashtable	struct
Dictionary <K,V> Where K = string and V = scalar or array of [Numeric, boolean, Char, String]	struct
ArrayList	cell
Any other .NET type in the default application domain	.NET object

.NET Type	Converts to MATLAB Type
Any other serializable .NET type in a non-default application domain	.NET object

Conversion Rules: MATLAB Numeric Types to .NET Types

To Convert This MATLAB Type:	To this:	Follow these rules:
numeric	Scalar	The type must be scalar in MATLAB. For example, a 1 X 1 int in MATLAB.
	Vector	The type must be a vector in MATLAB. For example, a 1 X <i>N</i> or <i>N</i> X 1 int array in MATLAB.
	<i>N</i> -dimensional array	The <i>N</i> -dimensional array type specified by the user must match the rank of the MATLAB numeric array.

Tip When converting MATLAB numeric arrays, widening conversions are allowed. For example, an int can be converted to a double. The type specified must be a numeric type that is equal or wider. Narrowing conversions throw an ArgumentException.

Caution .NET types are not as flexible as MATLAB types. Take care and test appropriately with .NET outputs before integrating data into your applications.

Conversion Rules: MATLAB Char Arrays to .NET Types

To Convert This MATLAB Type:	To this:	Follow these rules:
char	Char	The char must be scalar.
	Char array	The <i>N</i> -dimensional Char type must match the rank of the MATLAB char array.
	String	MATLAB char array must be [1, <i>N</i>]
	String array	The <i>N</i> -dimensional MATLAB char array can be converted to (<i>N</i> - 1) - dimensional array of type String.

Conversion Rules: MATLAB Logical Arrays to .NET Types

To Convert This MATLAB Type:	To this:	Follow these rules:
logical	Boolean	The logical must be scalar.
	Boolean[]	The MATLAB logical array must be [1,N] or [N,1].
	Boolean array	The N-dimensional Boolean array must match the rank of the MATLAB logical array.

Conversion Rules: Cell Array to .NET Types

To Convert This MATLAB Type:	To this:	Follow these rules:
cell	System.Array	The N-dimensional MATLAB cell array is converted to an N-dimensional System.Array of type object.
	ArrayList	The MATLAB cell array must be a vector.

Caution If the MATLAB cell array contains a struct, it is left unchanged. All other types are converted to native types. Any nested cell array is converted to a System.Array matching the dimension of the cell array, as illustrated in this code snippet:

```
Let C = {[1,2,3], {[1,2,3]}, 'Hello world'}
% be a cell
```

C can be converted to an object[1,3] where object[1,1] contains int[,], object[1,2] contains an object[1,1] whose first element is an int[,], and object[1,3] contains char[,].

Note Any nested cell array is converted to a System.Array that matches the dimension of the cell array

Conversion Rules: Struct to .NET Types

To Convert This MATLAB Type:	To this:	Follow these rules:
struct	.NET struct	The name and number of public fields in the specified .NET struct must match the name and number of fields in the MATLAB struct.
	Hashtable	A scalar struct can be converted to a Hashtable. Any nested struct will also be converted to a Hashtable. If the nested struct is not a scalar, then an ArgumentException is thrown. The dictionary key must be of type String.

Conversion Rules: .NET Objects in MATLAB to .NET Native Objects

To Convert this MATLAB Type:	To this:	Follow these rules:
.NET object	Type or super-type of the containing object	A .NET object in MATLAB can only be converted to a type or a super-type.

Character and String Conversion

A native .NET string is converted to a 1-by- N MATLAB character array, with N equal to the length of the .NET string.

An array of .NET strings (`string[]`) is converted to an M -by- N character array, with M equal to the number of elements in the string (`[]`) array and N equal to the maximum string length in the array.

Higher dimensional arrays of `String` are similarly converted.

In general, an N -dimensional array of `String` is converted to an $N+1$ dimensional MATLAB character array with appropriate zero padding where supplied strings have different lengths.

Unsupported MATLAB Array Types

The MATLAB Compiler SDK product does not support returning the following MATLAB array types because they are not CLS-compliant:

- `int8`
- `uint16`
- `uint32`
- `uint64`

However, it is permissible to pass these types as arguments to a MATLAB Compiler SDK component.

Interfaces Generated by the MATLAB Compiler SDK

In this section...

“Single Output API” on page 11-9

“Standard API” on page 11-10

“feval API” on page 11-11

For each MATLAB function that you specify as part of a .NET assembly, the MATLAB Compiler SDK product generates an API based on the MATLAB function signature.

- A single output on page 11-9 signature that assumes that only a single output is required and returns the result in a single `MWArray` rather than an array of `MWArray`.
- A standard on page 11-10 signature that specifies inputs of type `MWArray` and returns values as an array of `MWArray`.
- A feval on page 11-11 signature that includes both input and output arguments in the argument list rather than returning outputs as a return value. Output arguments are specified first, followed by the input arguments.

Single Output API

For each MATLAB function, the MATLAB Compiler SDK product generates a wrapper class that has overloaded methods to implement the various forms of the generic MATLAB function call. The single output API for a MATLAB function returns a single `MWArray`.

Typically, you use the single output interface for MATLAB functions that return a single argument. You can also use the single output interface when you want to use the output of a function as the input to another function.

The following table shows a generic function `foo` along with the single output API that the compiler generates for its several forms.

Generic MATLAB function	<code>function [Out1, Out2, ..., varargout] = foo(In1, In2, ..., InN, varargin)</code>
API if there are no input arguments	<code>public MWArray foo()</code>
API if there are one or more input arguments	<code>public MWArray foo(MWArray In1, MWArray In2...MWArray inN)</code>
API if there are optional input arguments	<code>public MWArray foo(MWArray In1, MWArray In2, ..., MWArray inN params MWArray[] varargin)</code>

In the example, the input arguments `In1`, `In2`, and `inN` are of type `MWArray`.

Similarly, in the case of optional arguments, the `params` arguments are of type `MWArray`. (The `varargin` argument is similar to the `varargin` function in MATLAB — it allows the user to pass a variable number of arguments.)

Note When you call a class method in your .NET application, specify all required inputs first, followed by any optional arguments.

Functions having a single integer input require an explicit cast to type `MWNumericArray` to distinguish the method signature from a standard interface signature that has no input arguments.

Standard API

Typically, you use the standard interface for MATLAB functions that return multiple output values.

The standard calling interface returns an array of `MWArray` rather than a single array object.

The following table shows the standard API for a generic function with none, one, more than one, or a variable number of arguments.

Generic MATLAB function	<code>function [Out1, Out2, ..., varargout] = foo(In1, In2, ..., InN, varargin)</code>
API if there are no input arguments	<code>public MWArray[] foo(int numArgsOut)</code>
API if there is one input argument	<code>public MWArray [] foo(int numArgsOut, MWArray In1)</code>
API if there are two to <i>N</i> input arguments	<code>public MWArray[] foo(int numArgsOut, MWArray In1, MWArray In2, \dots MWArray InN)</code>
API if there are optional arguments, represented by the <code>varargin</code> argument	<code>public MWArray[] foo(int numArgsOut, MWArray in1, MWArray in2, MWArray InN, params MWArray[] varargin)</code>

Details about the arguments for these samples of standard signatures are shown in the following table.

Argument	Description	Details
<code>numArgsOut</code>	Number of outputs	An integer indicating the number of outputs you want the method to return. The <code>numArgsOut</code> argument must always be the first argument in the list.
<code>In1, In2, ...InN</code>	Required input arguments	All arguments that follow <code>numArgsOut</code> in the argument list are inputs to the method being called. Specify all required inputs first. Each required input must be of type <code>MWArray</code> or one of its derived types.
<code>varargin</code>	Optional inputs	You can also specify optional inputs if your MATLAB code uses the <code>varargin</code> input: list the optional inputs, or put them in an <code>MWArray[]</code> argument, placing the array last in the argument list.
<code>Out1, Out2, ...OutN</code>	Output arguments	With the standard calling interface, all output arguments are returned as an array of <code>MWArray</code> .

feval API

In addition to the methods in the single API and the standard API, in most cases, the MATLAB Compiler SDK product produces an additional overloaded method. If the original MATLAB code contains no output arguments, then the compiler will not generate the `feval` method interface.

Consider a function with the following structure:

```
function [Out1, Out2, ..., varargout] = foo(In1, In2, ..., InN,
varargin)
```

The compiler generates the following API, known as the `feval` interface.

```
public void foo
    (int numArgsOut,
     ref MArray [] ArgsOut,
     MArray[] ArgsIn)
```

The interface accepts the following arguments:

<code>numArgsOut</code>	Number of outputs	An integer indicating the number of outputs you want to return. This number generally matches the number of output arguments that follow. The <code>varargout</code> array counts as just one argument, if present.
<code>ref MArray [] ArgsOut</code>	Output arguments	Following <code>numArgsOut</code> are all the outputs of the original MATLAB code, each listed in the same order as they appear on the left side of the original MATLAB code. A <code>ref</code> attribute prefaces all output arguments, indicating that these arrays are passed by reference.
<code>MArray[] ArgsIn</code>	Input arguments	<code>MArray</code> types or supported .NET primitive types. When you pass an instance of an <code>MArray</code> type, the underlying MATLAB array is passed directly to the called function. Native types are first converted to <code>MArray</code> types.

Functions

compiler.build.dotNETAssembly

Create .NET assembly for deployment outside MATLAB

Syntax

```
compiler.build.dotNETAssembly(Files)
compiler.build.dotNETAssembly(Files,Name,Value)
compiler.build.dotNETAssembly(ClassMap)
compiler.build.dotNETAssembly(ClassMap,Name,Value)
compiler.build.dotNETAssembly(opts)
results = compiler.build.dotNETAssembly( ___ )
```

Description

Caution This function is only supported on Windows operating systems.

`compiler.build.dotNETAssembly(Files)` creates a .NET assembly using the MATLAB functions specified by `Files`.

`compiler.build.dotNETAssembly(Files,Name,Value)` creates a .NET assembly with additional options specified using one or more name-value arguments. Options include the class name, output directory, and additional files to include.

`compiler.build.dotNETAssembly(ClassMap)` creates a .NET assembly with a class mapping specified using a container.Map object `ClassMap`.

`compiler.build.dotNETAssembly(ClassMap,Name,Value)` creates a .NET assembly using `ClassMap` and additional options specified using one or more name-value arguments. Options include the assembly name, output directory, and additional files to include.

`compiler.build.dotNETAssembly(opts)` creates a .NET assembly with options specified using a `compiler.build.DotNetAssemblyOptions` object `opts`. You cannot specify any other options using name-value arguments.

`results = compiler.build.dotNETAssembly(___)` returns build information as a `compiler.build.Results` object using any of the input argument combinations in previous syntaxes. The build information consists of the build type, paths to the compiled files, and build options.

Examples

Create .NET Assembly Using File

Create a .NET assembly on a Windows system using a function file that generates a magic square.

In MATLAB, locate the MATLAB function that you want to deploy as a .NET assembly. For this example, use the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

```
appFile = fullfile(matlabroot, 'extern', 'examples', 'compiler', 'magicsquare.m');
```

Build a .NET assembly using the `compiler.build.dotNETAssembly` command.

```
compiler.build.dotNETAssembly(appFile);
```

The build function generates the following files within a folder named `magicsquaredotNETAssembly` in your current working directory:

- `GettingStarted.html` — HTML file that contains information on integrating your assembly.
- `includedSupportPackages.txt` — Text file that lists all support files included in the assembly.
- `magicsquare.dll` — Dynamic-link library file that can be accessed using the `mwArray` API.
- `magicsquare.xml` — XML file that contains documentation for the `mwArray` assembly.
- `magicsquare_overview.html` — HTML file that contains requirements for accessing the component and for generating arguments using the `mwArray` class hierarchy.
- `magicsquareNative.dll` — Dynamic-link library file that can be accessed using the native API.
- `magicsquareNative.xml` — XML file that contains documentation for the native assembly.
- `magicsquareVersion.cs` — C# file that contains version information.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see *MATLAB Compiler Limitations*.
- `readme.txt` — Text file that contains packaging and interface information.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

Customize .NET Assembly

Create a .NET assembly on a Windows system and customize it using name-value arguments.

For this example, use the files `flames.m` and `flames.mat` located in `matlabroot\extern\examples\compiler`.

```
appFile = fullfile(matlabroot, 'extern', 'examples', 'compiler', 'flames.m');
MATFile = fullfile(matlabroot, 'extern', 'examples', 'compiler', 'flames.mat');
```

Build a .NET assembly using the `compiler.build.dotNETAssembly` command. Use name-value arguments to specify the assembly name and version, add a MAT-file, and enable verbose output.

```
compiler.build.dotNETAssembly(appFile, 'AssemblyName', 'FlamesComp', ...
    'AssemblyVersion', '2.0', ...
    'AdditionalFiles', MATFile, ...
    'Verbose', 'on');
```

Create .NET Assembly Using Class Map Input

Create a .NET assembly on a Windows system using a class map and multiple function files.

Create a `containers.Map` object whose keys are class names and whose values are the locations of function files.

```
cmap = containers.Map;
cmap('Class1') = {'exampleFcn1.m','exampleFcn2.m'};
cmap('Class2') = {'exampleFcn3.m','exampleFcn4.m'};
```

Build a .NET assembly using the `compiler.build.dotNETAssembly` command.

```
compiler.build.dotNETAssembly(cmap);
```

Alternatively, you can specify additional options using name-value arguments when you build the .NET assembly.

```
compiler.build.dotNETAssembly(cmap,...
    'AssemblyName','MyExampleComp',...
    'AssemblyVersion','2.0',...
    'Verbose','on');
```

Create Multiple Assemblies Using Options Object

Create multiple .NET assemblies on a Windows system using a `compiler.build.DotNETAssemblyOptions` object.

For this example, use the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
```

Create a `DotNETAssemblyOptions` object using `appFile`. Use name-value arguments to specify a common output directory, generate the assembly archive separately, and enable verbose output.

```
opts = compiler.build.DotNETAssemblyOptions(appFile,...
    'OutputDir','D:\Documents\MATLAB\work\dotNETBatch',...
    'EmbedArchive','off',...
    'Verbose','on')
```

```
opts =
```

DotNETAssemblyOptions with properties:

```
    AssemblyName: 'example.magicsquare'
    AssemblyVersion: '1.0.0.0'
    ClassMap: [1x1 containers.Map]
    DebugBuild: off
    EmbedArchive: off
    EnableRemoting: off
    SampleGenerationFiles: {}
    StrongNameKeyFile: ''
    AdditionalFiles: {}
    AutoDetectDataFiles: on
    SupportPackages: {'autodetect'}
    Verbose: on
    OutputDir: 'D:\Documents\MATLAB\work\dotNETBatch'
```

Class Map Information

```
magicsquareClass: {'C:\Program Files\MATLAB\R2021b\extern\examples\compiler\magicsquare
```

Build the .NET Assembly using the `DotNETAssemblyOptions` object.

```
compiler.build.dotNETAssembly(opts);
```

To compile using the function file `hello.m` with the same options, use dot notation to modify the `ClassMap` of the existing `COMComponentOptions` object before running the build function again.

```
remove(opts.ClassMap, keys(opts.ClassMap));
opts.ClassMap('helloClass') = fullfile(matlabroot, 'extern', 'examples', 'compiler', 'hello.m');
compiler.build.dotNETAssembly(opts);
```

By modifying the `ClassMap` argument and recompiling, you can compile multiple components using the same options object.

Get Build Information from .NET Assembly

Create a .NET assembly on a Windows system and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

```
results = compiler.build.dotNETAssembly('magicsquare.m')

results =

    Results with properties:

        BuildType: 'dotNETAssembly'
           Files: {4x1 cell}
IncludedSupportPackages: {}
           Options: [1x1 compiler.build.DotNETAssemblyOptions]
```

The `Files` property contains the paths to the following compiled files:

- `magicsquare.dll`
- `magicsquareNative.dll`
- `magicsquare_overview.dll`
- `GettingStarted.html`

Input Arguments

Files — Files implementing MATLAB functions

character vector | string scalar | cell array of character vectors | string array

Files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `["myfunc1.m", "myfunc2.m"]`

Data Types: `char` | `string` | `cell`

ClassMap — Class map

`containers.Map` object

Class map, specified as a `containers.Map` object. Map keys are class names and each value is the set of files mapped to the corresponding class. Files must have a `.m` extension.

Example: `cmap`

opts — .NET assembly build options

`compiler.build.DotNetAssemblyOptions` object

.NET assembly build options, specified as a `compiler.build.DotNETAssemblyOptions` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Verbose', 'on'`

AdditionalFiles — Additional files

character vector | string scalar | cell array of character vectors | string array

Additional files to include in the .NET assembly, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute.

Example: `'AdditionalFiles', ["myvars.mat", "data.txt"]`

Data Types: `char` | `string` | `cell`

AssemblyName — Name of .NET assembly

character vector | string scalar

Name of the .NET assembly, specified as a character vector or a string scalar. Specify `'AssemblyName'` as a namespace, which is a period-separated list, such as `companyname.groupname.component`. The name of the generated library is set to the last entry of the period-separated list. The name must begin with a letter and contain only alphabetic characters and periods.

Example: `'AssemblyName', 'mathworks.dotnet.mymagic'`

Data Types: `char` | `string`

AssemblyVersion — Assembly version

`'1.0.0.0'` (default) | character vector | string scalar

Assembly version, specified as a character vector or a string scalar. For information on versioning using MATLAB Compiler SDK, see “Versioning”.

Example: `'AssemblyVersion', '4.0'`

Data Types: `char` | `string`

AutoDetectDataFiles — Flag to automatically include data files

`'on'` (default) | on/off logical value

Flag to automatically include data files, specified as `'on'` or `'off'`, or as numeric or logical 1 (`true`) or 0 (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the .NET assembly.
- If you set this property to `'off'`, then you must add data files to the assembly using the `AdditionalFiles` option.

Example: `'AutoDetectDataFiles', 'off'`

Data Types: `logical`

ClassName — Name of .NET class

character vector | string scalar

Name of the .NET class, specified as a character vector or a string scalar. You cannot specify this option if you use a `ClassMap` input. Class names must meet the .NET class name requirements.

The default value is the name of the first file listed in the `Files` argument appended with `Class`.

Example: `'ClassName', 'magicsquareClass'`

Data Types: `char` | `string`

DebugBuild — Flag to enable debug symbols

'off' (default) | on/off logical value

Flag to enable debug symbols, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the compiled assembly contains debug symbols.
- If you set this property to 'off', then the compiled assembly does not contain debug symbols.

Example: `'DebugBuild', 'on'`

Data Types: `logical`

EmbedArchive — Flag to embed assembly archive

'on' (default) | on/off logical value

Flag to embed the assembly archive, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the function embeds the assembly archive in the .NET assembly.
- If you set this property to 'off', then the function generates the assembly archive as a separate file.

Example: `'EmbedArchive', 'off'`

Data Types: `logical`

EnableRemoting — Flag to control remoting type

'off' (default) | on/off logical value

Flag to control the remoting type of the assembly, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the function builds a remotable assembly.

- If you set this property to 'off', then the function builds an assembly that is not remotable.

Example: 'EnableRemoting', 'on'

Data Types: logical

OutputDir — Path to output directory

character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the assembly name appended with dotNETAssembly.

Example: 'OutputDir', 'D:\Documents\MATLAB\work\mymagicdotNETAssembly'

Data Types: char | string

SampleGenerationFiles — MATLAB sample files

character vector | string scalar | cell array of character vectors | string array

MATLAB sample files used to generate sample .NET driver files for functions included within the assembly, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a .m extension. For more information and limitations, see “Sample Driver File Creation”.

Example: 'SampleGenerationFiles', ["sample1.m", "sample2.m"]

Data Types: char | string | cell

StrongNameKeyFile — Path to encryption key

character vector | string scalar

Path to the encryption key file used to sign the shared assembly, specified as a character vector or a string scalar. If the value is empty, the function creates a private assembly. The file path can be relative to the current working directory or absolute.

Example: 'StrongNameKeyFile', 'sgKey.snk'

Data Types: char | string

SupportPackages — Support packages

'autodetect' (default) | 'none' | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- 'autodetect' (default) — The dependency analysis process detects and includes the required support packages automatically.
- 'none' — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

Example: 'SupportPackages', {'Deep Learning Toolbox Converter for TensorFlow Models', 'Deep Learning Toolbox Model for Places365-GoogLeNet Network'}

Data Types: char | string | cell

Verbose — Flag to control build verbosity

'off' (default) | on/off logical value

Flag to control build verbosity, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to 'off', then the command window does not display progress information.

Example: 'Verbose', 'on'

Data Types: logical

Output Arguments

results — Build results

`compiler.build.Results` object

Build results, returned as a `compiler.build.Results` object. The Results object consists of:

- Build type, which is 'dotNETAssembly'
- Paths to the following compiled files:
 - *AssemblyName.dll*
 - *AssemblyNameNative.dll*
 - *AssemblyName_overview.dll*
 - *GettingStarted.html*
- A list of included support packages
- Build options, specified as a `DotNETAssemblyOptions` object

Limitations

- This function is only supported on Windows operating systems.

See Also

`compiler.build.DotNETAssemblyOptions`

Introduced in R2021a

compiler.build.DotNETAssemblyOptions

Options for building .NET assemblies

Syntax

```
opts = compiler.build.DotNETAssemblyOptions(Files)
opts = compiler.build.DotNETAssemblyOptions(Files,Name,Value)
opts = compiler.build.DotNETAssemblyOptions(ClassMap)
opts = compiler.build.DotNETAssemblyOptions(ClassMap,Name,Value)
```

Description

`opts = compiler.build.DotNETAssemblyOptions(Files)` creates a `DotNETAssemblyOptions` object using MATLAB functions specified by `Files`. Use the `DotNETAssemblyOptions` object as an input to the `compiler.build.dotNETAssembly` function.

`opts = compiler.build.DotNETAssemblyOptions(Files,Name,Value)` creates a `DotNETAssemblyOptions` object with options specified using one or more name-value arguments. Options include the class name, output directory, and additional files to include.

`opts = compiler.build.DotNETAssemblyOptions(ClassMap)` creates a `DotNETAssemblyOptions` object with a class mapping specified using a `containers.Map` object `ClassMap`.

`opts = compiler.build.DotNETAssemblyOptions(ClassMap,Name,Value)` creates a `DotNETAssemblyOptions` object with a class mapping specified using `ClassMap` and options specified using one or more name-value arguments. Options include the assembly name, output directory, and additional files to include.

Examples

Create .NET Assembly Options Object Using File

Create a `DotNETAssemblyOptions` object using file input.

For this example, use the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
opts = compiler.build.DotNETAssemblyOptions(appFile)
```

```
opts =
```

```
DotNETAssemblyOptions with properties:
```

```
AssemblyName: 'example.magicsquare'
AssemblyVersion: '1.0.0.0'
ClassMap: [1x1 containers.Map]
DebugBuild: off
EmbedArchive: on
```

```

        EnableRemoting: off
    SampleGenerationFiles: {}
        StrongNameKeyFile: ''
        AdditionalFiles: {}
    AutoDetectDataFiles: on
        SupportPackages: {'autodetect'}
        Verbose: off
        OutputDir: '.\magicsquaredotNETAssembly'

```

Class Map Information

```
magicsquareClass: {'C:\Program Files\MATLAB\R2021b\extern\examples\compiler\magicsquare
```

You can modify property values of an existing `DotNETAssemblyOptions` object using dot notation.

```
opts.Verbose = 'on'
```

```
opts =
```

DotNETAssemblyOptions with properties:

```

        AssemblyName: 'example.magicsquare'
    AssemblyVersion: '1.0.0.0'
        ClassMap: [1x1 containers.Map]
    DebugBuild: off
    EmbedArchive: on
    EnableRemoting: off
    SampleGenerationFiles: {}
        StrongNameKeyFile: ''
        AdditionalFiles: {}
    AutoDetectDataFiles: on
        SupportPackages: {'autodetect'}
        Verbose: on
        OutputDir: '.\magicsquaredotNETAssembly'

```

Class Map Information

```
magicsquareClass: {'C:\Program Files\MATLAB\R2021b\extern\examples\compiler\magicsquare
```

Use the `DotNETAssemblyOptions` object as an input to the `compiler.build.dotNETAssembly` function to build a .NET assembly.

```
buildResults = compiler.build.dotNETAssembly(opts);
```

Customize .NET Assembly Options Object

Create a `DotNETAssemblyOptions` object and customize it using name-value arguments.

For this example, use the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`. Use name-value arguments to specify the output directory and disable automatic detection of data files.

```
opts = compiler.build.DotNETAssemblyOptions('magicsquare.m',...
    'OutputDir','D:\Documents\MATLAB\work\MagicDotNET',...
    'AutoDetectDataFiles','off')
```

```
opts =
```

DotNETAssemblyOptions with properties:

```

        AssemblyName: 'example.magicsquare'
        AssemblyVersion: '1.0.0.0'
        ClassMap: [1x1 containers.Map]
        DebugBuild: off
        EmbedArchive: on
        EnableRemoting: off
        SampleGenerationFiles: {}
        StrongNameKeyFile: ''
        AdditionalFiles: {}
        AutoDetectDataFiles: off
        SupportPackages: {'autodetect'}
        Verbose: off
        OutputDir: 'D:\Documents\MATLAB\work\MagicDotNET'

```

Class Map Information

```
magicsquareClass: {'C:\Program Files\MATLAB\R2021b\extern\examples\compiler\magicsquare
```

You can modify the property values of an existing `DotNETAssemblyOptions` object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on'
```

```
opts =
```

DotNETAssemblyOptions with properties:

```

        AssemblyName: 'example.magicsquare'
        AssemblyVersion: '1.0.0.0'
        ClassMap: [1x1 containers.Map]
        DebugBuild: off
        EmbedArchive: on
        EnableRemoting: off
        SampleGenerationFiles: {}
        StrongNameKeyFile: ''
        AdditionalFiles: {}
        AutoDetectDataFiles: off
        SupportPackages: {'autodetect'}
        Verbose: on
        OutputDir: 'D:\Documents\MATLAB\work\MagicDotNET'

```

Class Map Information

```
magicsquareClass: {'C:\Program Files\MATLAB\R2021b\extern\examples\compiler\magicsquare
```

Use the `DotNETAssemblyOptions` object as an input to the `compiler.build.dotNETAssembly` function to build a .NET assembly.

```
buildResults = compiler.build.dotNETAssembly(opts);
```

Create .NET Assembly Options Object Using Class Map

Create a `DotNETAssemblyOptions` object using a class map.

Create a `containers.Map` object whose keys are class names and whose values are MATLAB function files.

```

cmap = containers.Map;
cmap('Class1') = {'exampleFcn1.m', 'exampleFcn2.m'};
cmap('Class2') = {'exampleFcn3.m', 'exampleFcn4.m'};

```

Create the `DotNETAssemblyOptions` object using the class map `cmap`.

```
opts = compiler.build.DotNETAssemblyOptions(cmap)
opts =
    DotNETAssemblyOptions with properties:
        AssemblyName: 'example.exampleFcn1'
        AssemblyVersion: '1.0.0.0'
        ClassMap: [2x1 containers.Map]
        DebugBuild: off
        EmbedArchive: on
        EnableRemoting: off
        SampleGenerationFiles: {}
        StrongNameKeyFile: ''
        AdditionalFiles: {}
        AutoDetectDataFiles: on
        SupportPackages: {'autodetect'}
        Verbose: off
        OutputDir: '.\exampleFcn1dotNETAssembly'

    Class Map Information
        Class1: {2x1 cell}
        Class2: {2x1 cell}
```

You can also create a `DotNETAssemblyOptions` object using name-value arguments or modify an existing object using dot notation. For this example, specify an output directory, enable verbose output, and disable automatic detection of data files.

```
opts = compiler.build.DotNETAssemblyOptions(cmap,...
    'OutputDir','D:\Documents\MATLAB\work\MagicDotNET',...
    'Verbose','On');
opts.AutoDetectDataFiles = 'off'
opts =
    DotNETAssemblyOptions with properties:
        AssemblyName: 'example.exampleFcn1'
        AssemblyVersion: '1.0.0.0'
        ClassMap: [2x1 containers.Map]
        DebugBuild: off
        EmbedArchive: on
        EnableRemoting: off
        SampleGenerationFiles: {}
        StrongNameKeyFile: ''
        AdditionalFiles: {}
        AutoDetectDataFiles: off
        SupportPackages: {'autodetect'}
        Verbose: on
        OutputDir: 'D:\Documents\MATLAB\work\MagicDotNET'

    Class Map Information
        Class1: {2x1 cell}
        Class2: {2x1 cell}
```

Use the `DotNETAssemblyOptions` object as an input to the `compiler.build.dotNETAssembly` function to build a .NET assembly.

```
buildResults = compiler.build.dotNETAssembly(opts);
```

Input Arguments

Files — Files implementing MATLAB functions

character vector | string scalar | cell array of character vectors | string array

Files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `["myfunc1.m", "myfunc2.m"]`

Data Types: `char` | `string` | `cell`

ClassMap — Class map

`containers.Map` object

Class map, specified as a `containers.Map` object. Map keys are class names and each value is the set of files mapped to the corresponding class. Files must have a `.m` extension.

Example: `cmap`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Verbose', 'on'`

AdditionalFiles — Additional files

character vector | string scalar | cell array of character vectors | string array

Additional files to include in the `.NET` assembly, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute.

Example: `'AdditionalFiles', ["myvars.mat", "data.txt"]`

Data Types: `char` | `string` | `cell`

AssemblyName — Name of .NET assembly

character vector | string scalar

Name of the `.NET` assembly, specified as a character vector or a string scalar. Specify `'AssemblyName'` as a namespace, which is a period-separated list, such as `companyname.groupname.component`. The name of the generated library is set to the last entry of the period-separated list. The name must begin with a letter and contain only alphabetic characters and periods.

Example: `'AssemblyName', 'mathworks.dotnet.mymagic'`

Data Types: `char` | `string`

AssemblyVersion — Assembly version

`'1.0.0.0'` (default) | character vector | string scalar

Assembly version, specified as a character vector or a string scalar. For information on versioning using MATLAB Compiler SDK, see “Versioning”.

Example: 'AssemblyVersion', '4.0'

Data Types: char | string

AutoDetectDataFiles — Flag to automatically include data files

'on' (default) | on/off logical value

Flag to automatically include data files, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the .NET assembly.
- If you set this property to 'off', then you must add data files to the assembly using the `AdditionalFiles` option.

Example: 'AutoDetectDataFiles', 'off'

Data Types: logical

ClassName — Name of .NET class

character vector | string scalar

Name of the .NET class, specified as a character vector or a string scalar. You cannot specify this option if you use a `ClassMap` input. Class names must meet the .NET class name requirements.

The default value is the name of the first file listed in the `Files` argument appended with `Class`.

Example: 'ClassName', 'magicsquareClass'

Data Types: char | string

DebugBuild — Flag to enable debug symbols

'off' (default) | on/off logical value

Flag to enable debug symbols, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the compiled assembly contains debug symbols.
- If you set this property to 'off', then the compiled assembly does not contain debug symbols.

Example: 'DebugBuild', 'on'

Data Types: logical

EmbedArchive — Flag to embed assembly archive

'on' (default) | on/off logical value

Flag to embed the assembly archive, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the function embeds the assembly archive in the .NET assembly.
- If you set this property to 'off', then the function generates the assembly archive as a separate file.

Example: 'EmbedArchive', 'off'

Data Types: logical

EnableRemoting – Flag to control remoting type

'off' (default) | on/off logical value

Flag to control the remoting type of the assembly, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the function builds a remotable assembly.
- If you set this property to 'off', then the function builds an assembly that is not remotable.

Example: 'EnableRemoting', 'on'

Data Types: logical

OutputDir – Path to output directory

character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the assembly name appended with `dotNETAssembly`.

Example: 'OutputDir', 'D:\Documents\MATLAB\work\mymagicdotNETAssembly'

Data Types: char | string

SampleGenerationFiles – MATLAB sample files

character vector | string scalar | cell array of character vectors | string array

MATLAB sample files used to generate sample .NET driver files for functions included within the assembly, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a .m extension. For more information and limitations, see "Sample Driver File Creation".

Example: 'SampleGenerationFiles', ["sample1.m", "sample2.m"]

Data Types: char | string | cell

StrongNameKeyFile – Path to encryption key

character vector | string scalar

Path to the encryption key file used to sign the shared assembly, specified as a character vector or a string scalar. If the value is empty, the function creates a private assembly. The file path can be relative to the current working directory or absolute.

Example: 'StrongNameKeyFile', 'sgKey.snk'

Data Types: char | string

SupportPackages — Support packages

'autodetect' (default) | 'none' | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- 'autodetect' (default) — The dependency analysis process detects and includes the required support packages automatically.
- 'none' — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

.

Example: 'SupportPackages', {'Deep Learning Toolbox Converter for TensorFlow Models', 'Deep Learning Toolbox Model for Places365-GoogLeNet Network'}

Data Types: char | string | cell

Verbose — Flag to control build verbosity

'off' (default) | on/off logical value

Flag to control build verbosity, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to 'off', then the command window does not display progress information.

Example: 'Verbose', 'on'

Data Types: logical

Output Arguments**opts — .NET assembly build options**

DotNETAssemblyOptions object

.NET assembly build options, returned as a DotNETAssemblyOptions object.

See Also

`compiler.build.dotNETAssembly`

Introduced in R2021a

enableTSUtilsfromNetworkDrive

Set trust setting to load .NET assemblies from network drive

Syntax

```
enableTSUtilsfromNetworkDrive
```

Description

`enableTSUtilsfromNetworkDrive` sets the trust setting so that the MATLAB Compiler SDK module can load .NET assemblies from remote drives.

Note This is only required when using .NET 2.0 or 3.5.

Examples

Enable Load from Remote Drives

Enable MATLAB Compiler SDK to use network drives on a system.

Enter the following on the MATLAB command line after logging in with Administrator privileges:

```
enableTSUtilsfromNetworkDrive
```

Tips

- Administrator privileges are required to run this command.
- The setting is permanent and only needs to be executed once per machine.

Introduced in R2013a

ntswrap

Generate type-safe API

Syntax

```
ntswrap('-a','interfaceAssemblyFile','-c','componentName.className','-i','interfaceName')
ntswrap('-a','interfaceAssemblyFile','-c','componentName.className','-i','interfaceName','-s','-k')
ntswrap(___,'-b','assemblyFile','-d','-n','Namespace.Class','-o','outputDir','-v','version','-w','assemblyName')
```

Description

`ntswrap('-a','interfaceAssemblyFile','-c','componentName.className','-i','interfaceName')` generates an assembly that contains the type-safe API for the MATLAB Compiler SDK .NET assembly *componentName* wrapped in the class *className*.

`ntswrap('-a','interfaceAssemblyFile','-c','componentName.className','-i','interfaceName','-s','-k')` generates the source code for the type-safe API instead of an assembly.

`ntswrap(___,'-b','assemblyFile','-d','-n','Namespace.Class','-o','outputDir','-v','version','-w','assemblyName')` generates the type-safe API with additional options specified using one or more of the listed arguments. Options include the Microsoft .NET Framework version, assembly and class names, and output directory.

Examples

Generate Type-Safe Assembly

Generate an assembly that contains a type-safe API using a .NET assembly and a type-safe interface DLL.

Create the type-safe interface `IMultiply.dll` and .NET assembly `Multiply.dll` using the procedure outlined in “Implement Type-Safe Interface and Integrate into .NET Application” on page 7-7.

Generate the type-safe API using `ntswrap`.

```
ntswrap('-c','Multiply.Arithmetic', ...
        '-a','IMultiply.dll', ...
        '-i','IMultiply');
```

This syntax generates the .NET binary `ArithmeticIMultiply.dll` that contains a type-safe API for the MATLAB Compiler SDK class `Arithmetic` in the namespace `Multiply`.

Generate Source Code for Type-Safe Assembly

Generate source code for a type-safe API using a .NET assembly and a type-safe interface DLL.

Create the type-safe interface `IMultiply.dll` and .NET assembly `Multiply.dll` using the procedure outlined in “Implement Type-Safe Interface and Integrate into .NET Application” on page 7-7.

Generate the type-safe API using `ntswrap`. Use the additional arguments `-s` and `-k` to generate source code instead of an assembly.

```
ntswrap('-c','Multiply.Arithmetic', ...
        '-a','IMultiply.dll', ...
        '-i','IMultiply', ...
        '-s','-k');
```

This syntax generates the source code file `ArithmeticIMultiply.cs` for the type-safe API.

Input Arguments

-a — Assembly that contains statically typed interface

character vector | string scalar

Assembly that contains the statically typed interface referenced by the `-i` argument, specified as a character vector or string vector that contains the relative or absolute path to the assembly.

Example: `'-a','IMyInterface.dll'`

Data Types: `char` | `string`

-b — .NET assembly

character vector | string scalar

The .NET assembly that defines the component referenced by the `-c` argument, specified as a character vector that contains the relative or absolute path to the assembly. Use this option if `ntswrap` cannot find the specified .NET assembly.

Example: `'-b','MATLAB_NET_assembly.dll'`

Data Types: `char` | `string`

-c — Component class namespace

character vector | string scalar

Component class namespace of the .NET assembly, specified as a period-separated list that consists of the component name followed by the class name. If the assembly is scoped to a namespace, specify the full namespace-qualified name.

Example: `'-c','MydotNETComp.MyClass'`

Data Types: `char` | `string`

-d — Enable debugging

Enable debugging of the type-safe API assembly. This option is incompatible with `-s`.

-i — Interface name

character vector | string scalar

Interface name, specified by a character vector or string scalar. The interface name is usually prefixed by an `I` and correlates to the interface assembly identified by the `-a` option.

Example: `'-i', 'IMyInterface'`

Data Types: `char | string`

-k — Keep source code

Keep the generated type-safe API source code. If this argument is omitted, the source code is deleted after processing. This argument is optional.

-n — Namespace containing type-safe class

character vector | string scalar

Namespace containing the type-safe API class, specified as a character vector or string scalar. Use this option to override the namespace specified by the `-c` argument. This argument is optional.

Example: `'-n', 'Sample.Sample'`

Data Types: `char | string`

-o — Path to output folder

character vector | string scalar

Path to the output folder where the build files are saved, specified as a character vector or a string scalar that contains the relative or absolute path. This argument is optional.

Example: `'-o', 'D:\Documents\MATLAB\work\TypeSafeProject'`

Data Types: `char | string`

-s — Generate source code

Generate source code only; do not compile type-safe API source into an assembly. This argument is optional.

-v — Microsoft .NET Framework version

character vector | string scalar

Microsoft .NET Framework version (csc compiler) used to generate the type-safe API assembly, specified as a character vector or string scalar. This argument is optional and is incompatible with `-s`.

Example: `'-v', 'v4.0'`

Data Types: `char | string`

-w — Name of type-safe API

character vector | string scalar

Name of the generated type-safe API class and assembly, specified as a character vector or string scalar. Use this option to override the default name. This argument is optional.

Example: `'-w', 'TypeSafeMultiply'`

Data Types: `char | string`

Tips

- To use `ntswrap` from the Windows command prompt, use the following syntax:

```
ntswrap.exe -a interfaceAssembly -c className -i interfaceName
```

`ntswrap.exe` is located in `matlabroot\toolbox\dotnetbuilder\bin\<arch>`.

See Also

Topics

“Type-Safe Interfaces” on page 7-2

“Implement Type-Safe Interface and Integrate into .NET Application” on page 7-7

Introduced in R2011a

Deploying .NET Components With the F# Programming Language

Integrate .NET Assembly Into F# Application

The F# programming language offers the opportunity to implement the same solutions you usually implement using C#, but with less code. This can be helpful when scaling a deployment solution across an enterprise-wide installation, or in any situation where code efficiency is valued. The brevity of F# programs can also make them easier to maintain.

The following example shows you how to integrate the deployable MATLAB `magic` function into an F# application.

Prerequisites

You must be running Microsoft Visual Studio 2010 or higher to use this example.

If you build this example on a system running 64-bit Microsoft Visual Studio, you must add a reference to the 32-bit `MWArray` DLL due to a current imitation of Microsoft's F# compiler.

Step 1: Build the Component

Build the `MagicSquareComp` component using the instructions in “Generate .NET Assembly and Build .NET Application”.

Step 2: Integrate Component Into F# Application

- 1 Using Microsoft Visual Studio 2010 or higher, create an F# project.
- 2 Add references to your .NET component and `MWArray` in Visual Studio.
- 3 Make the .NET namespaces available for your component and `MWArray` libraries:

```
open MagicSquareComp
open MathWorks.MATLAB.NET.Arrays
```

- 4 Define the Magic Square function with an initial `let` statement, as follows:

```
let magic n =
```

Then, add the following statements to complete the function definition.

- a Instantiate the Magic Square component:

```
use magicComp = new MagicSquareComp.MagicSquareClass()
```

- b Define the input argument:

```
use inarg = new MWNumericArray((int) n)
```

- c Call MATLAB, get the output argument cell array, and extract the first element as a two-dimensional float array:

```
(magicComp.makesquare(1, inarg).[0].ToArray() :> float[,])
```

The complete function definition looks like this:

```
let magic n =
    // Instantiate the magic square component
    use magicComp = new MagicSquareComp.MagicSquareClass()
```



```
// Define the input argument
use inarg = new MWNumericArray((int) n)
// Call MATLAB, get the output argument cell array,
// extract the first element as a 2D float array
(magicComp.makesquare(1, inarg).[0].ToArray()
 :?> float[,])
```

- 5 Add another let statement to define the output display logic:

```
let printMagic n =
    let numArray = magic n
    // Display the output
    printfn "Number of [rows,cols]: [%d,%d]"
        (numArray.GetLength(0)) (numArray.GetLength(1))
    printfn ""
    for i in 0 .. numArray.GetLength(0)-1 do
        for j in 0 .. numArray.GetLength(1)-1 do
            printf "%3.0f " numArray.[i,j]
        printfn ""
    printfn "=====\n"

ignore(List.iter printMagic [1..19])
// Pause until keypress
ignore(System.Console.ReadKey())
```

The complete program listing follows:

The F# Magic Square Program

```
open MagicSquareComp
open MathWorks.MATLAB.NET.Arrays
let magic n =
    // Instantiate the magic square component
    use magicComp = new MagicSquareComp.MagicSquareClass()
    // Define the input argument
    use inarg = new MWNumericArray((int) n)
    // Call MATLAB, get the output argument cell array,
    // extract the first element as a 2D float array
    (magicComp.makesquare(1, inarg).[0].ToArray() :?> float[,])

let printMagic n =
    let numArray = magic n
    // Display the output
    printfn "Number of [rows,cols]: [%d,%d]"
        (numArray.GetLength(0)) (numArray.GetLength(1))
    printfn ""
    for i in 0 .. numArray.GetLength(0)-1 do
        for j in 0 .. numArray.GetLength(1)-1 do
            printf "%3.0f " numArray.[i,j]
        printfn ""
    printfn "=====\n"

ignore(List.iter printMagic [1..19])
// Pause until keypress
ignore(System.Console.ReadKey())
```

Step 3: Deploy the Component

For information about deploying your component to end users, see “MATLAB Runtime” on page 6-3.