MATLAB® Builder for COM

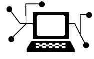
The Language of Technical Computing

- Computation
- Visualization
- Programming

User's Guide

Version 1

How to Contact The MathWorks:

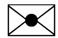


www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup

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

508-647-7000 Phone

508-647-7001 Fax



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

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

MATLAB Builder for COM

© COPYRIGHT 2002 – 2005 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

July 2002	Online only	New for Version 1.0 (Release 13)
June 2004	Online only	Revised for Version 1.1 (Release 14) Name changed from MATLAB COM Builder
August 2004	Online only	Revised for Version 1.1.1 (Release 14+)
October 2004	Online only	Revised for Version 1.1.2 (Release 14SP1)
September 2005	Online only	Revised for Version 1.1.5 (Release 14SP3)

Contents

1 Getting Started

Building a Deployable Application **1-2**

What Is a Project? **1-10**
Classes and Methods **1-10**
Versions **1-10**

Using the Command Line Interface **1-12**

Requirements for MATLAB Builder for COM **1-15**
System Requirements **1-15**
Compiler Requirements **1-15**
Limitations and Restrictions **1-15**

2 Graphical User Interface

MATLAB Builder **2-2**

Project Settings Window **2-6**

Component Information Window **2-7**
Sample Component Information Window **2-7**

Package Files Window **2-8**

3 Programming with COM Objects Created by MATLAB Builder for COM

General Techniques **3-3**

Registering and Referencing the Utility Library **3-5**

Creating an Instance of a Class in Visual Basic **3-6**
CreateObject Function **3-6**
Visual Basic New Operator **3-7**
Advantages of Each Technique **3-7**
Declaring a Reusable Class Instance **3-8**

Calling the Methods of a Class Instance **3-9**
Variant **3-10**
Examples of Passing Input and Output **3-10**

Calling a COM Object in a C++ Program **3-12**
Using COM Builder to Create the Object **3-12**
Using the Component in a C++ Program **3-15**

Add Events to MATLAB Builder for COM Objects **3-17**
Using a Callback with a Visual Basic Event **3-18**

Passing Arguments **3-21**
Creating and Using a varargin Array in Visual Basic Programs **3-21**
Creating and using varargout in Visual Basic programs . . **3-22**

Using Flags to Control Array Formatting and Data Conversion **3-23**
Overview **3-23**

Using MATLAB Global Variables **3-30**
Using MATLAB Global Variables in Visual Basic **3-30**

Obtaining Registry Information **3-33**

Handling Errors During a Method Call 3-35

4 Usage Examples

Magic Square Example 4-2
Creating the M-File 4-2
Creating the Project 4-2
Building the Project 4-4
Creating the Visual Basic Project 4-5
Creating the User Interface 4-5
Creating the Executable 4-9
Testing the Application 4-9
Packaging the Component 4-9

Spectral Analysis Example 4-11
Building the Component 4-11
Integrating the Component with VBA 4-13
Creating the Visual Basic Form 4-16
Adding The Spectral Analysis Menu Item to Excel 4-21
Saving the Add-in 4-23
Testing The Add-in 4-24
Package the Component 4-26

Univariate Interpolation 4-27
Building the Component 4-27
Building the Project 4-28
Using the Component in Visual Basic 4-30
Creating the Visual Basic Form 4-32

Matrix Calculator 4-39
Building the Component 4-39
Building the Project 4-41
Using the Component in Visual Basic 4-41
Creating the Visual Basic Form 4-43

Curve Fitting 4-52
Building the Component 4-52
Building the Project 4-53

Using the Component in Visual Basic 4-55
Creating the Visual Basic Form 4-56

Bouncing Ball Simulation **4-63**
Building the Component 4-63
Building the Project 4-65
Using the Component in Visual Basic 4-65
Creating the Visual Basic Form 4-67

5 Troubleshooting

6 How MATLAB Builder for COM Works Internally

Overview of Internal Processes **6-2**
Code Generation 6-2
Create Interface Definitions 6-2
C++ Compilation 6-3
Linking and Resource Binding 6-3
Component Registration 6-3

Component Registration **6-4**
Self-Registering Components 6-4
Globally Unique Identifier (GUID) 6-5
Versioning 6-6

Data Conversion Rules **6-8**
Array Formatting Flags 6-18
Data Conversion Flags 6-20

Calling Conventions **6-22**
Producing a COM Class 6-22
IDL Mapping 6-23
Visual Basic Mapping 6-24

7 Functions — Alphabetical List

8 Utility Library

Referencing the Utility Classes **8-2**

Utility Library Classes **8-3**
Class MWUtil **8-3**
Class MWFlags **8-10**
Class MWStruct **8-16**
Class MWField **8-23**
Class MWComplex **8-24**
Class MWSparse **8-26**
Class MWArg **8-29**

Enumerations **8-31**
Enum mwArrayFormat **8-31**
Enum mwDataType **8-31**
Enum mwDateFormat **8-32**

A Examples

Calling a COM Object in a C++ Program **A-2**

Passing Arguments **A-3**

Using MATLAB Global Variables **A-4**

Querying the Registry **A-5**

Basic Usage Example: Visual Basic **A-6**

Creating a Comprehensive Excel Add-in **A-7**

Comprehensive Examples **A-8**

Index

Getting Started

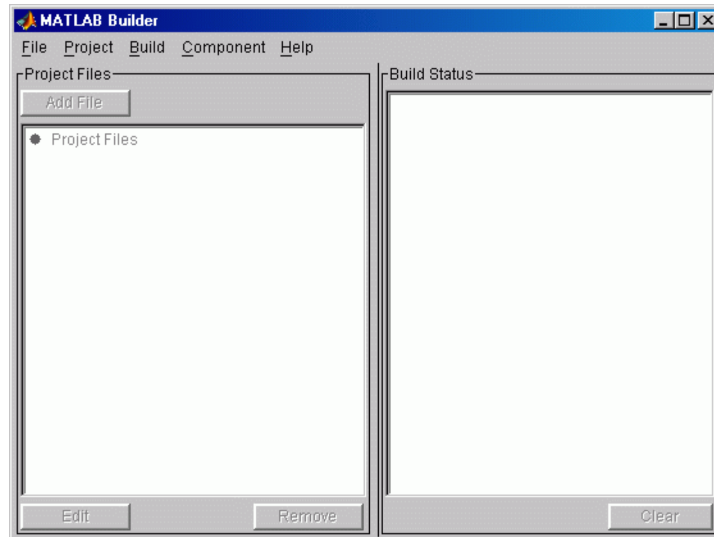
Building a Deployable Application (p. 1-2)	How to create and package a COM component
What Is a Project? (p. 1-10)	How MATLAB Builder for COM uses the specifications in a project
Using the Command Line Interface (p. 1-12)	How you can use the `mcc` command instead of the GUI to build COM objects
Requirements for MATLAB Builder for COM (p. 1-15)	Software requirements for using MATLAB Builder for COM

Building a Deployable Application

To create a deployable COM component and package it for distribution, follow these steps:

1 Type `comtool` at the command line in MATLAB.

   The MATLAB Builder window appears.



2 Click **File > New > Project** to open the New Project Settings window.

New Project Settings

3. Specify names for the component and at least one class that will be part of the component.

Using the New Project Settings Window

Area of Window	How To Use This Area
Component name	Type the name that you want to use for the component to be created. This name is also used for the Dynamic Link Library (DLL) that is created to implement the component. A component name cannot match the name of any M-files or MEX-files added to the project.
Class name	After you type the component name, the GUI automatically enters a default class name; the default is the component name with `class` appended. For instance, the default class name for a component named `MATLABfunction` would be `MATLABfunctionclass` To change the name of the class, add a new class with the name that you want to use and remove the class that has the default name. To add another class to your component, enter the class name in the **Class name** box, and click **Add>>**. The added class appears in the **Classes** text box. Any new classes become part of the newly built component. Removed classes are removed from the component on the subsequent build.
Project version	Default value is 1.0. See “Versions” on page 1-10 for additional information about version numbers.

Area of Window	How To Use This Area
Project directory	Specifies where project and build files are written when compiling and packaging your components. COM Builder specifies the project directory automatically, based on the name of your current directory and the component name. You can accept the automatically generated project directory path or specify a different path. Once you click **OK** in the Project Settings Window, the specified path is saved. If you decide to move the project or change anything on its path, you need to repeat the entire project specification process, including adding files to the project.
Create a singleton MCR option	Tells COM Builder to generate code that creates only one instance of the MCR per application. If you click this option for a component, and a programmer uses the component more than once in an application, each instance of the component uses the same MCR. MCR stands for MATLAB Component Run-time, which is required to run applications on machines that do not have MATLAB installed.
Build Debug version	Enables you to trace back to the point where where a failure occurred — in the initialization of MCR, the function call, or the termination routine. This setting does not affect M-file debugging.
Display verbose output	Tells COM Builder to show each step in the build process. This output is saved in `project_dir\build.log`.

4 Accept the current settings by clicking **OK**.

COM Builder creates and saves a project file named `component_name.cbl` in the project directory..

The project file is part of your project workspace. It contains the names of any M-files or MEX- files you subsequently add to the project. Adding these files is the next step.

5 Click **Add File** to add methods to the component classes.

Understanding the MATLAB Builder Window

When you save a new project or open an existing project, COM Builder opens the MATLAB Builder window, which has the following fields and settings:

Area of Window	How To Use This Area
Menu bar commands	File
	Project
	Build
	Component
	Help
Add File	Add M-files and/or MEX-files to the project by clicking **Add File** or clicking **Project > Add File**. You can add only one file at a time to the project. The name of any file added to the project cannot duplicate the name of any function existing in the library of precompiled functions.
Project Files	List of folders and files in the project. The project folder contains folders that correspond to classes that you have specified in the Project Settings window. The files in each class folder represent MATLAB code that COM Builder will encapsulate into methods for the classes.For instance, here is an illustration of a project folder

Area of Window	How To Use This Area
Build Status	Look at the **Build Status** panel to view the output of the build process and discover any problems. To clear the **Build Status** panel, click **Build > Clear Status**. The output of the build process is saved in the file <project_dir>\build.log. To open the Build Log, click **Build > Open Build Log**. If you have reason to contact MathWorks Technical Support with a question about the build process, you will be asked to provide a copy of this log.
Edit	To open an M-file for editing, do any of the following: • Select it and click **Edit**. • Double-click the file you want to edit. • Click **Project > Edit File**. You cannot edit MEX-files.
Remove	To remove a file from the project, do any of the following: • Select it and click **Remove**. • Double-click the file you want to edit. • Click **Project > Remove File** You can highlight multiple files for removal.
Clear	Click **Clear** in the MATLAB Builder window to remove the process log from the status panel.

6 Click **Build > COM Object** to build the project.

The build process sends intermediate source files to the \src sub-directory and output files necessary for deployment to the \distrib sub-directory of your project directory. The files in \distrib are DLLs, which are automatically registered on your system.

You probably want to test your component before packaging, which is the next step. After testing the component outside of the MATLAB environment you can reopen the project and proceed to the next step.

7. Click **Component > Package Component** to create a self-extracting executable. COM Builder names this file `componentname.exe`.

Files in the self-extracting executable

File	Purpose
`componentname.ctf`	Component Technology File (`ctf`) archive. This is a platform-dependent file that must correspond to the end user's platform.
`componentname_projectversion`	Component that encapsulates M-code
`_install.bat`	Script run by the self-extracting executable
`MCRInstaller.exe`	Self-extracting MATLAB Component Runtime library utility; platform-dependent file that must correspond to the end user's platform. `MCRInstaller.exe` installs MATLAB Component Runtime (MCR), which users of your component need to install on the target machine once per release.

8. Distribute the self-extracting executable to your users.

For More Information

Concepts involved in using COM Builder projects	"What Is a Project?" on page 1-10 "Classes and Methods" on page 1-10
Example of creating a simple component	"Calling a COM Object in a C++ Program" on page 3-12"Calling a COM Object in a C++ Program" on page 3-12
Using the command line instead of the GUI	"Using the Command Line Interface" on page 1-12

What Is a Project?

A project contains the files and settings needed by COM Builder to create a deployable component, or COM object.

COM stands for Component Object Model, which is a software architecture developed by Microsoft to build component-based applications. COM objects expose interfaces that allow applications and other components to access the features of the objects. COM objects are accessible through Visual Basic, C++, or any language that supports COM objects.

Classes and Methods

The components that COM Builder creates are implemented as classes. Each class contains a set of functions called methods. COM Builder transforms MATLAB functions that are included in the component's project into methods.

When creating a component, you must provide one or more class names as well as a component name. The component name specifies the name of the DLL file to be created; this is the file that implements the component. The class name, on the other hand, denotes the name of the class that performs a call on a specific method at run-time. To use a component you create, programmers need to instantiate the class and call its methods.

Typically you should specify names for components and classes that will be clear to programmers who use your COM objects. For example, if you are compiling 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.

Versions

COM Builder supports a simple versioning mechanism for components by attaching a version number to a component when it is created. COM Builder automatically includes the version number in the DLL filename and also in the system registry information. The default value for the first version of a component is 1.0.

COM Builder treats classes in different versions of the same component as distinct, even if the classes have the same name.

Using Version Numbers

For changes that you make before packaging the component you should not change the version number.

After deployment, change the version number for subsequent changes, so that you can manage the new and old versions.

Using the Command Line Interface

You can use the MATLAB command line interface instead of the GUI to create COM objects. Do this by issuing the `mcc` command with options. If you use `mcc`, you do not create a project.

Note See the MATLAB Compiler documentation for a complete description of the `mcc` command and its options.

The following table provides an overview of some `mcc` options related to components, along with syntax and examples of their usage.

How To Use COM Builder on the Command Line

<table>
<tr><th>Action to Perform</th><th>mcc Option to Use</th><th>Description</th></tr>
<tr><td rowspan="3">Create component that has one class.</td><td><code>-W com</code></td><td>The <code>W</code> option with <code>com</code> as the type controls the generation of wrapper files, which you can use to support components.</td></tr>
<tr><td colspan="2">Syntax
<code>mcc -W com:<component_name>[,<class_name>[,<major>.<minor>]]</code>
An unspecified <code><class_name></code> defaults to <code><component_name></code>, and an unspecified version number defaults to the latest version built or 1.0, if there is no previous version.</td></tr>
<tr><td colspan="2">Example
<code>mcc -W com:mycomponent,myclass,1.0 -T link:lib foo.m bar.m</code>
The example creates a COM component called <code>mycomponent</code>, which contains a single COM class named <code>myclass</code> with methods <code>foo</code> and <code>bar</code>, and a version of 1.0.</td></tr>
</table>

How To Use COM Builder on the Command Line (Continued)

Action to Perform	mcc Option to Use	Description
Add additional classes to a COM component.	Not needed	A separate COM class with name <class_name> is created for each class argument that is passed. Following the <class_name> parameter is a comma-separated list of source files that are encapsulated as methods for the class.
	Syntax class{<class_name>:[file, [file,...]]}	
	Example mcc -B ccom:mycomponent,myclass,1.0 foo.m bar.m class{myclass2:foo2.m, bar2.m} The example creates a COM component named mycomponent with two classes: myclass has methods foo and bar, and myclass2 has methods foo2 and bar2. The version is version 1.0.	
Simplify the command line input for components.	-B ccom:	Uses the bundle file.
	Syntax mcc -B <filename>[:<a1>,<a2>,...,<an>]	
	Example mcc -B ccom:mycomponent,myclass,1.0 foo.m bar.m	
Control how each COM class uses the MCR.	-S	By default, a new MCR instance is created for each instance of each COM class in the component. Use -S to change the default. This option tells COM Builder to create a single MCR is when the first COM class is instantiated. This MCR is reused and shared among all subsequent class instances, resulting in more efficient memory usage and eliminating the MCR startup cost in each subsequent class instantiation. When using -S, note that all class instances share a single MATLAB workspace and share global variables in the M-files used to build the

How To Use COM Builder on the Command Line (Continued)

Action to Perform	mcc Option to Use	Description
		component. This makes properties of a COM class behave as static properties instead of instance-wise properties.
	Example `mcc -S -B ccom:mycomponent,myclass,1.0 foo.m bar.m` The example creates a COM component called `mycomponent` containing	
Create subdirectories needed for deployment and copy associated files to them.	`-d`	The `\src` and `\distrib` subdirectories are needed to package components.
	Syntax `-d` *`directoryname`*	

Requirements for MATLAB Builder for COM

System Requirements

System requirements and restrictions on use for MATLAB Builder for COM are almost identical to those listed in the MATLAB Compiler documentation.

Compiler Requirements

You must have MATLAB and the MATLAB Compiler installed to install MATLAB Builder for COM.

MATLAB Builder for COM is available only on Windows. The compilers that support the building of COM objects are Microsoft Visual C/C++ (versions 6.0 and 7.1).

For an up-to-date list of all the compilers supported by MATLAB and the MATLAB Compiler, see the MathWorks Technical Support Department's Technical Notes at `http://www.mathworks.com/support/tech-notes/1600/1601.shtml`.

Limitations and Restrictions

In general, limitations and restrictions on the use of COM Builder are the same as those for the MATLAB Compiler. See the MATLAB Compiler documentation for details.

2

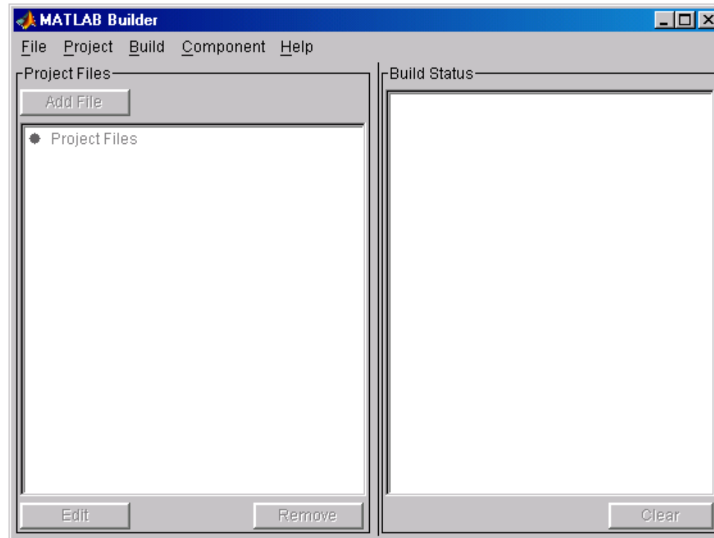
Graphical User Interface

MATLAB Builder for COM has the following primary windows:

MATLAB Builder (p. 2-2)	Serves as the hub of operations with COM Builder
Project Settings Window (p. 2-6)	Helps you specify project properties for a new or existing project
Component Information Window (p. 2-7)	Shows details about components created with COM Builder
Package Files Window (p. 2-8)	Helps you specify and create a self-extracting executable for your component

MATLAB Builder

The MATLAB function `comtool` displays the MATLAB Builder graphical user interface (GUI) window.



The MATLAB Builder window has the following menus:

- "File Menu" on page 2-2
- "Project Menu" on page 2-3
- "Build Menu" on page 2-4
- "Component Menu" on page 2-4
- "Help Menu" on page 2-4

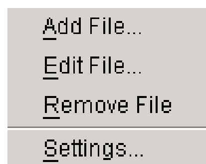
File Menu

Use the **File** menu to create and manage COM Builder projects.

- Click **New Project** to open the project settings window, which you can use to creates a project containing M-files and MEX-files to encapsulate in COM components.
- Click **Open Project** to load a previously saved project.
- Click **Save Project** to save the current project. If you have not yet saved the current project, you are prompted for a filename.
- Click **Save As Project** to save the current project after prompting for a filename.
- Click **Close Project** to close the current project.
- **Close COMTOOL** closes the MATLAB Builder window.

Project Menu

Use the Project menu commands to control and manage the the files and settings, or properties, for a project.



- Click **Add File** to add an M-file or MEX-file to the current project.
- Click **Edit File** to edit the selected M-file. (You can also click **Edit File** button to perform this task).
- Click **Remove File** to remove the selected M-file. (You can also click **Remove** button to perform this task).

- Click **Settings** to view the current project settings. See “Project Settings Window” on page 2-6 for details.

Build Menu

Use the **Build** menu to control the building of the project’s files into a COM object.

- Click **COM Object** to build a COM object from the current project files.
- Click **Clear Status** to clear the **Build Status** pane.
- Click **Open Build Log** to display the output of the build process that has been saved in the log file.

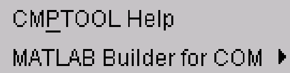
Component Menu

Use the **Component** menu to package a component and view details about a component.

- Click **Package Component** to complete the process of producing a deployable application after you have built and tested the components in a project.
- Click **Component Info** to view detailed information about a component that you have built with COM Builder.

Help Menu

The **Help** menu provides access to the context-sensitive help for the COM Builder graphical user interface.



The MATLAB Builder window also includes the following buttons:

- Click **Add File** to add files to a selected project.
- Click **Edit** to edit a selected file.
- Click **Remove** to remove selected files.
- Click **Clear** to remove output from the **Build Status** pane.

The **Build Status** pane shows the current build log.

Project Settings Window

For new projects, click **File > New > Project** to open the New Project Settings window.

For existing projects, click **File > Open > Project > Settings** to open the Project Settings window.

The project settings are as follows:

Setting	Description
Component name	Name of the component you are creating with this project.
Class name	Name of a class that you want to add.
Classes	List of classes currently in this project.
Project version	Version number of this project.
Project directory	Location for output files generated by COM Builder, including the project file
Compiler options	Select **Create a singleton MCR** if your users can share a single installation of the MCR. Select **Build debug version** to add debugging information to the classes generated by COM Builder. Select **Show verbose output** to display all details and create a complete log of the build process.

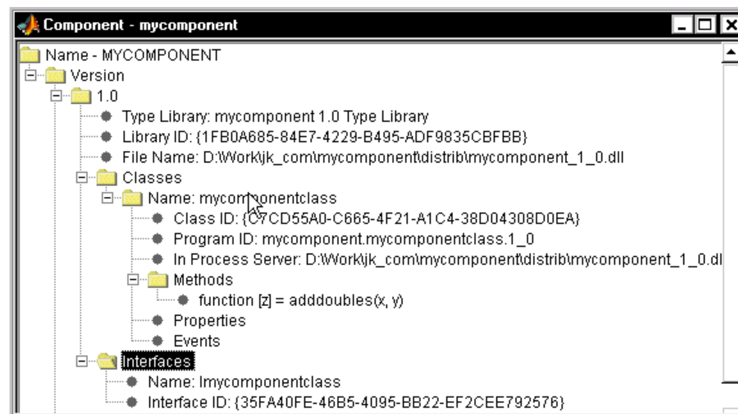
Component Information Window

This listing presents the component information that is stored in the registry.

See `componentinfo` for an explanation of these fields.

Sample Component Information Window

The following window displays information for `mycomponent`, the component created in "Calling a COM Object in a C++ Program" on page 3-12.



For this particular component there is just one version. The **Classes** folder contains information about the DLL, including the Program ID (`PROGID`). The `Methods` folder contains a list of the functions in MATLAB that are encapsulated and can be called as methods in the class.

Package Files Window

Use the Package Files window to specify the files and properties that COM Builder should use to create a self-extracting executable for the component that you have built.

Click **Add File** to add files to the package. You do not need to add any files that are in the project.

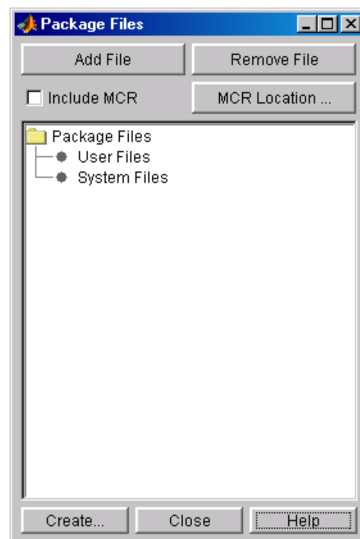
Click **Remove File** to delete files from the package.

Click or clear the **Include MCR** check box to include or exclude MATLAB Component Runtime (MCR) from the package. MCR is a stand-alone set of shared libraries that enables the execution of M-files. MCR provides complete support for all MATLAB language features.

Click **MCR Location** to specify the directory location of `MCRInstaller.zip`.

After you have specified the files that you want to include in the package, click **Create**.

Here is an illustration of the Package Files window:



3

Programming with COM Objects Created by MATLAB Builder for COM

General Techniques (p. 3-3)	Describes the integration of MATLAB Builder for COM components into COM-compliant programs
Registering and Referencing the Utility Library (p. 3-5)	How to register and reference the utilities you need in your program
Creating an Instance of a Class in Visual Basic (p. 3-6)	Describes two techniques for calling a class method (encapsulated MATLAB function).
Calling the Methods of a Class Instance (p. 3-9)	Describes how you call the class methods to access the encapsulated MATLAB functions.
Calling a COM Object in a C++ Program (p. 3-12)	How to use COM Builder to integrate a COM object into a C++ program
Add Events to MATLAB Builder for COM Objects (p. 3-17)	Describes how you can turn a MATLAB function into an event function.
Passing Arguments (p. 3-21)	Describes how you can pass multiple arguments as a varargin array by creating a Variant array, assigning each element of the array to the respective input argument.

Using Flags to Control Array Formatting and Data Conversion (p. 3-23)	Describes array formatting and data conversion flags.
Using MATLAB Global Variables (p. 3-30)	Describes class properties, which allow an object to retain an internal state between method calls.
Obtaining Registry Information (p. 3-33)	How to use MATLAB function `componentinfo` to query the system registry for any installed MATLAB Builder for COM components.
Handling Errors During a Method Call (p. 3-35)	Describes the Visual Basic exception handling capability.

General Techniques

After you package and install a COM component created by MATLAB Builder for COM, you can access the component in any program that supports COM, such as Visual Basic, Visual C++, or Visual C#.

Your code module must

- Load the components created by COM Builder
  - “Registering and Referencing the Utility Library” on page 3-5
  - “Creating an Instance of a Class in Visual Basic” on page 3-6
- Call methods of the component class
  - “Calling the Methods of a Class Instance” on page 3-9
  - “Calling a COM Object in a C++ Program” on page 3-12
  - “Add Events to MATLAB Builder for COM Objects” on page 3-17
  - “Obtaining Registry Information” on page 3-33
- Deal with data conversion and parameter passing
  - “Passing Arguments ” on page 3-21
  - “Using Flags to Control Array Formatting and Data Conversion” on page 3-23
  - “Using MATLAB Global Variables” on page 3-30
- Process errors
  - “Handling Errors During a Method Call” on page 3-35

Note These topics provide general information on how to integrate COM Builder components into your COM-compliant programs. The presentation focuses on the special programming techniques needed for components based on MATLAB and generated by COM Builder. It assumes that you have a working knowledge of the programming language used in these programs.

For information about programming with COM objects in Microsoft Visual Studio, see articles in the MSDN library, such as Calling COM Components from .NET Clients.

Registering and Referencing the Utility Library

The MWComUtil library provided with the MATLAB Builder for COM is freely distributable. The MWComUtil library includes seven classes and three enumerated types. These utilities are required for array processing, and they provide type definitions used in data conversion.

The library is contained in the file mwcomutil.dll. It must be registered once on each machine that uses components created with COM Builder.

Register the MWComUtil library at the DOS command prompt with the command

```
mwregsvr mwcomutil.dll
```

To use the types in the library, make sure that you reference the MWComUtil library in your current project:

1. Click **Tools > References**.

2. Click **MWComUtil 7.1 Type Library**.

Creating an Instance of a Class in Visual Basic

Before calling a class method that encapsulates MATLAB functions, you must create an instance of the class. You can do this in Visual Basic using the following techniques.

- Using the "CreateObject Function" on page 3-6
- Using the "Visual Basic New Operator" on page 3-7
- "Declaring a Reusable Class Instance" on page 3-8

Each technique has advantages and disadvantages.

For an example of creating a class instance in Visual C++, see "Calling a COM Object in a C++ Program" on page 3-12.

CreateObject Function

This method uses the Visual Basic application program interface (API) `CreateObject` function to create an instance of the class.

1. Dimension a variable of type `Object` to hold a reference to the class instance.

2. Call `CreateObject` with the Program ID (`ProgID`) for the class as an argument.

Here is a programming example:

```
Function foo(x1 As Variant, x2 As Variant) As Variant
       Dim aClass As Object

       On Error Goto Handle_Error
       aClass = CreateObject("mycomponent.myclass.1_0")
       ' (call some methods on aClass)
       Exit Function
Handle_Error:
   foo = Err.Description
End Function
```

Visual Basic New Operator

This method uses the Visual Basic `New` operator on a variable explicitly dimensioned as the class to be created.

1. Make sure that you reference the type library containing the class in the current Visual Basic project.
   a. Open the Visual Basic editor.
   b. Click **Project > References > Available References**.
   c. Select the necessary type library.
2. Dimension the class instance.
3. Use `New` to instantiate the class with a particular name.

The following sample function, `foo`, shows how to use the `New` operator to create a class instance.

```
Function foo(x1 As Variant, x2 As Variant) As Variant
   Dim aClass As mycomponent.myclass

   On Error Goto Handle_Error
   Set aClass = New mycomponent.myclass
   ' (call some methods on aClass)
   Exit Function
Handle_Error:
   foo = Err.Description
End Function
```

In this example, the class instance could be dimensioned as simply `myclass`. The full declaration in the form `<component-name>.<class-name>` guards against name collisions that could occur if other libraries in the current project contain types named `myclass`.

Advantages of Each Technique

Both techniques (using `CreateObject` and using `New`) are equivalent in the way they function, but each has different advantages. The first technique does not require a reference to the type library in the Visual Basic project, while the second results in faster code execution. The second technique has

the added advantage of enabling **Auto-List-Members** and **Auto-Quick-Info** in the Visual Basic editor to help you work with your classes.

Declaring a Reusable Class Instance

In the previous examples, the class instance used to call the method is a local variable within a procedure. Thus a new class instance is created and destroyed for each call to the method. As an alternative, you can declare a single module-scoped class instance that is reused by all function calls. The next example shows this technique.

```
Dim aClass As mycomponent.myclass

Function foo(x1 As Variant, x2 As Variant) As Variant
   On Error Goto Handle_Error
   If aClass Is Nothing Then
      Set aClass = New mycomponent.myclass
   End If
   ' (call some methods on aClass)
   Exit Function
Handle_Error:
   foo = Err.Description
End Function
```

Calling the Methods of a Class Instance

After you create a class instance, you can call the class methods to access the encapsulated MATLAB functions. MATLAB Builder for COM uses a standard technique to map the original MATLAB function syntax to the method's argument list. This standard mapping technique is as follows:

- `nargout`

  When a method has output arguments, the first argument is always `nargout`, which is of type `Long`. This input parameter passes the normal MATLAB `nargout` parameter to the encapsulated function and specifies how many outputs are requested. Methods that do not have output arguments do not pass a `nargout` argument.

- Output parameters

  Following `nargout` are the output parameters listed in the same order as they appear on the left side of the original MATLAB function.

- Input parameters

  Next come the input parameters listed in the same order as they appear on the right side of the original MATLAB function.

For example, the most generic MATLAB function is

```
function [Y1, Y2, ..., varargout] = foo(X1, X2, ..., varargin)
```

This function maps directly to the following Visual Basic signature:

```
Sub foo(nargout As Long, _
        Y1 As Variant, _
        Y2 As Variant, _
        .
        .
        varargout As Variant, _
        X1 As Variant, _
        X2 As Varaint, _
        .
        .
        varargin As Variant)
```

See "Calling Conventions" on page 6-22 for more details and examples of the standard mapping from MATLAB functions to COM class method calls.

Variant

All input and output arguments are typed as `Variant`, the default Visual Basic data type. The `Variant` type can hold any of the basic Visual Basic types, arrays of any type, and object references. See "Data Conversion Rules" on page 6-8 for details about the conversion of any basic type to and from MATLAB data types.

In general, you can supply any Visual Basic type as an argument to a class method, with the exception of Visual Basic User Defined Types (UDTs).

When you pass a simple `Variant` type as an output parameter, the called method allocates the received data and frees the original contents of the `Variant`. In this case it is sufficient to dimension each output argument as a single `Variant`. When an object type (like an Excel `Range`) is passed as an output parameter, the object reference is passed in both directions, and the object's `Value` property receives the data.

Examples of Passing Input and Output

The following examples show how to pass input and output parameters to COM Builder component class methods in Visual Basic.

The first example is a function, `foo`, that takes two arguments and returns one output argument. The `foo` function dispatches a call to a class method that corresponds to a MATLAB function of the form `function y = foo(x1,x2)`.

```
Function foo(x1 As Variant, x2 As Variant) As Variant
   Dim aClass As Object
   Dim y As Variant

   On Error Goto Handle_Error
   aClass = CreateObject("mycomponent.myclass.1_0")
   Call aClass.foo(1,y,x1,x2)
   foo = y
   Exit Function
Handle_Error:
```

```
   foo = Err.Description
End Function
```

The second example rewrites the `foo` function as a subroutine.

```
Sub foo(Xout As Variant, X1 As Variant, X2 As Variant)
   Dim aClass As Object

   On Error Goto Handle_Error
   aClass = CreateObject("mycomponent.myclass.1_0")
   Call aClass.foo(1,Xout,X1,X2)
   Exit Sub
Handle_Error:
   MsgBox(Err.Description)
End Sub
```

Calling a COM Object in a C++ Program

The following steps show you how to create a COM object with MATLAB Builder for COM and call that same object in a C++ program.

Note You must choose a Microsoft compiler to compile and use any COM object.

Using COM Builder to Create the Object

Build the COM object as follows:

1. Start MATLAB.

2. Execute the following command in MATLAB:

   ```
   mbuild -setup
   ```

   Be sure to choose a Microsoft compiler.

3. Open the MATLAB Editor and create a file named `adddoubles.m` with the following M-code:

   ```
   function z=adddoubles(x,y)
   z=x+y;
   ```

4. In the MATLAB Command Window, issue the following command:

   ```
   comtool
   ```

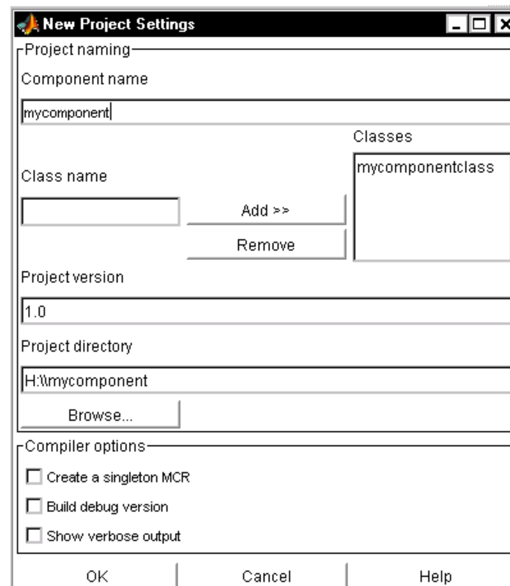
   The MATLAB Builder window opens.

5. Create a COM project as follows:

   a. From the menu bar in MATLAB Builder, click **File > New Project**.

   b. Type `mycomponent` as the component name.

      COM Builder automatically fills in the class name as `mycomponentclass` and adds the class to the **Classes** box, as shown:



c. Click **OK** to save the project settings.

d. If you are prompted to confirm whether you want to create a directory named `mycomponent` click **Yes**.

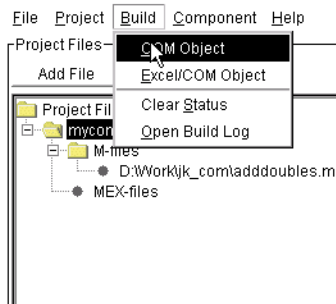
6. Select the folder named `mycomponentclass` in the left panel of the MATLAB Builder window and click **Add File**.

7. In the **Add file to project** dialog box, click `adddoubles.m` and click **Open**.

8. Open the `mycompponentclass` and `M-files` folders in the MATLAB Builder window.

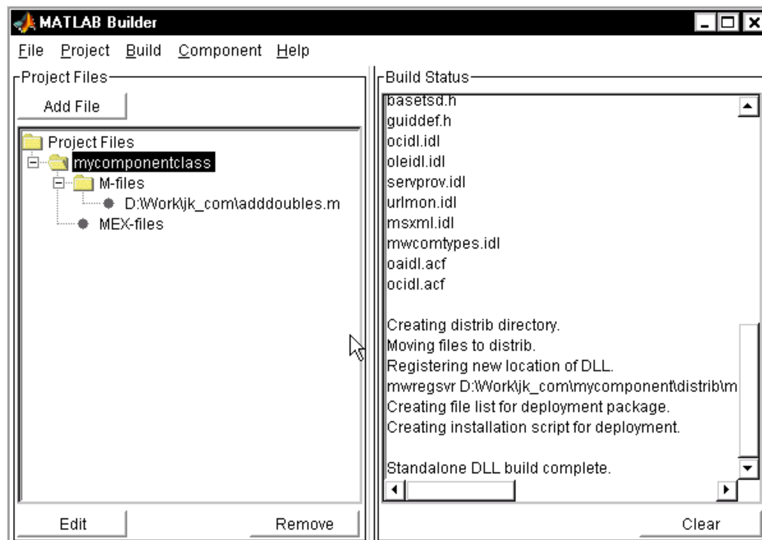
   You can see that the M-file has been added to the project in the `mycomponentclass` folder. This means that the MATLAB function, `adddoubles`, will be a method in `mycomponentclass`.

9. From the MATLAB Builder menu bar click **Build > COM Object**, as shown:



COM Builder generates a self-registering COM object that you can use in your C++ code.

The **Build Status** pane in the MATLAB Builder window displays the output of the build process, as shown:



Note Scroll horizontally and vertically to see all of the output from the build process.

Using the Component in a C++ Program

Use the COM object you have created as follows:

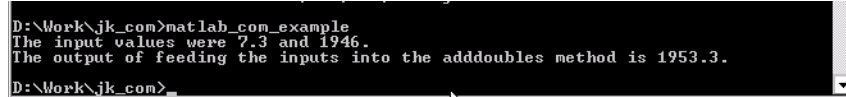
1 Create a C++ program in a file named `matlab_com_example.cpp` with the following code:

```
#include <iostream>
using namespace std;

// include the following files generated by MATLAB Builder for COM
#include "mycomponent\src\mycomponent_idl.h"
#include "mycomponent\src\mycomponent_idl_i.c"

int main() {
// Initialize argument variables
            VARIANT x, y, out1;
//Initialize the COM library
            HRESULT hr = CoInitialize(NULL);
//Create an instance of the COM object you created
            Imycomponentclass *pImycomponentclass;
            hr=CoCreateInstance
              (CLSID_mycomponentclass, NULL, CLSCTX_INPROC_SERVER, IID_Imycomponentclass,
                   (void **)&pImycomponentclass);
// Set the input arguments to the COM method
            x.vt=VT_R8;
            y.vt=VT_R8;
            x.dblVal=7.3;
            y.dblVal=1946.0;
// Access the method with arguments and receive the output out1
            hr=(pImycomponentclass -> adddoubles(1,&out1,x,y));
// Print the output
            cout << "The input values were " << x.dblVal << " and "
<< y.dblVal << ".\n";
            cout << "The output of feeding the inputs into the adddoubles method is " <<
out1.dblVal << ".\n";
// Uninitialize COM
            CoUninitialize();
            return 0;
}
```

2 In the MATLAB Command Window, compile the program as follows:

```
mbuild matlab_com_example.cpp
```

When you run the executable, the program displays two numbers and their sum, as returned by the COM object's `adddoubles`, as shown:



To distribute this program to users who do not have MATLAB, see "Package Files Window" on page 2-8.

Add Events to MATLAB Builder for COM Objects

MATLAB Builder for COM supports events, or callbacks, through a MATLAB language pragma. A *pragma* is a directive to COM Builder, beyond what is conveyed in the MATLAB language itself. The pragma for adding events is `#event`.

MATLAB interprets the `%#event` statement as a comment. But when COM Builder encapsulates a function, the `#event` pragma tells COM Builder that the function requires an *outgoing interface* and an *event handler*.

To use the `#event` pragma:

1. Write the code for a MATLAB function stub that serves as the prototype for the event. This function stub is the *event function*.

2. Build the COM component as usual. Make sure that you specify the event function you wrote in MATLAB as a method in the component class.

3. In your application, add the code to implement the event handler (the event handler belongs to the COM object created by COM Builder). The code for the event handler should implement the event function, or function stub, that you wrote in MATLAB.

When an encapsulated MATLAB function (now a method in a COM object in your application) calls the event function, the call is dispatched to the event handler in the application.

Some examples of how you might use callbacks in your code are

- To give the application periodic feedback during a long-running calculation by an encapsulated MATLAB function. For example, if you have a task that requires `n` iterations, you might signal an event to increment a progress bar in the user interface on each iteration.
- ·To signal a warning during a calculation but continue execution of the task.
- To return intermediate results of a calculation to the user and continue execution of the task.

Using a Callback with a Visual Basic Event

The example in this topic shows how to use a callback in conjunction with a Visual Basic `ProgressBar` control.

The MATLAB function `iterate` runs through `n` iterations and fires an event every `inc` iterations. When the function finishes, it returns a single output. To simulate actually doing something, the sample code includes a `pause` statement in the main loop so that the function waits for 1 second in each iteration.

The sample includes MATLAB functions `iterate.m` and `progress.m`.

iterate.m

```
function [x] = iterate(n,inc)
    %initialize x
    x = 0;
    % Run n iterations, callback every inc time
    k = 0;
    for i=1:n
        k = k + 1;
        if k == inc
            progress(i);
            k = 0;
        end;
        % Do some work on x...
        x = x + 1;
        % Pause for 1 second to simulate doing
        % something
        pause(1);
    end;
```

progess.m

```
function progress(i)
    %#event
    i
```

The `iterate` function runs through `n` iterations and calls the `progress` function every `inc` iterations, passing the current iteration number as an

argument. When this function is executed in MATLAB, the value of `i` is displayed each time the `progress` function gets called.

Suppose you create a COM Builder component that has these two functions included as class methods. For this example the component has a single class named `myclass`. The resulting COM class has a method `iterate` and an event `progress`.

To receive the event calls, implement a "listener" in the application. The Visual Basic syntax for the event handler for this example is

```
Sub aClass_progress(ByVal i As Variant)
```

where `aClass` is the variable name used for your class instance. The `ByVal` qualifier is used on all input parameters of an event function. To enable the listening process, dimension the `aClass` variable with the `WithEvents` keyword.

This example uses a simple Visual Basic form with three `TextBox` controls, one `CommandButton` control, and one `ProgressBar` control. The first text box, `Text1`, inputs the number of iterations, stored in the form variable `N`. The second text box, `Text2`, inputs the callback increment, stored in the variable `Inc`. The third text box, `Text3`, displays the output of the function when it finishes executing. The command button, `Command1`, executes the `iterate` method on your class when pressed. The progress bar control, `ProgressBar1`, updates itself in response to the progress event.

```
'Form Variables
Private WithEvents aClass As myclass    'Class instance
Private N As Long                       'Number of iterations
Private Inc As Long                     'Callback increment
Private Sub Form_Load()
'When form is loaded, create new myclass instance
    Set aClass = New myclass
    'Initialize variables
    N = 2
    Inc = 1
End Sub
Private Sub Text1_Change()
'Update value of N from Text1 text whenever it changes
```

```
    On Error Resume Next
    N = CLng(Text1.Text)
    If Err <> 0 Then N = 2
    If N < 2 Then N = 2
End Sub
Private Sub Text2_Change()
'Update value of Inc from Text2 text whenever it changes
    On Error Resume Next
    Inc = CLng(Text2.Text)
    If Err <> 0 Then Inc = 1
    If Inc <= 0 Then Inc = 1
End Sub
Private Sub Command1_Click()
'Execute function whenever Execute button is clicked
    Dim x As Variant
    On Error GoTo Handle_Error
    'Initialize ProgressBar
    ProgressBar1.Min = 1
    ProgressBar1.Max = N
    Text3.Text = ""
    'Iterate N times and call back at Inc intervals
    Call aClass.iterate(1, x, CDbl(N), CDbl(Inc))
    Text3.Text = Format(x)
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
Private Sub aClass_progress(ByVal i As Variant)
'Event handler. Called each time the iterate function
'calls the progress function. Progress bar is updated
'with the value passed in, causing the control to advance.
    ProgressBar1.Value = i
End Sub
```

Passing Arguments

When it encapsulates MATLAB functions, MATLAB Builder for COM adds the MATLAB function arguments to the argument list of the class methods it creates. Thus, if a MATLAB function uses `varargin` and/or `varargout`, COM Builder adds these arguments to the argument list of the class method. They are added at the end of the argument list for input and output arguments.

You can pass multiple arguments as a `varargin` array by creating a `Variant` array, assigning each element of the array to the respective input argument.

See "Producing a COM Class" on page 6-22 for more information about mapping of input and output arguments.

Creating and Using a varargin Array in Visual Basic Programs

The following example creates a `varargin` array to call a method encapsulating a MATLAB function of the form `y = foo(varargin)`.

The `MWUtil` class included in the `MWComUtil` utility library provides the `MWPack` helper function to create `varargin` parameters.

```
Function foo(x1 As Variant, x2 As Variant, x3 As Varaint, _
             x4 As Variant, x5 As Variant) As Variant
   Dim aClass As Object
   Dim v(1 To 5) As Variant
   Dim y As Variant

   On Error Goto Handle_Error
   v(1) = x1
   v(2) = x2
   v(3) = x3
   v(4) = x4
   v(5) = x5
   aClass = CreateObject("mycomponent.myclass.1_0")
   Call aClass.foo(1,y,v)
   foo = y
   Exit Function
Handle_Error:
```

```
   foo = Err.Description
End Function
```

Creating and using varargout in Visual Basic programs

The next example processes a `varargout` argument as three separate arguments. This function uses the `MWUnpack` function in the utility library.

The MATLAB function used is `varargout = foo(x1,x2)`.

```
Sub foo(Xout1 As Variant, Xout2 As Variant, Xout3 As Variant, _
        Xin1 As Variant, Xin2 As Variant)
   Dim aClass As Object
   Dim aUtil As Object
   Dim v As Variant

   On Error Goto Handle_Error
   aUtil = CreateObject("MWComUtil.MWUtil")
   aClass = CreateObject("mycomponent.myclass.1_0")
   Call aClass.foo(3,v,Xin1,Xin2)
   Call aUtil.MWUnpack(v,0,True,Xout1,Xout2,Xout3)
   Exit Sub
Handle_Error:
   MsgBox(Err.Description)
End Sub
```

Using Flags to Control Array Formatting and Data Conversion

Generally, you should write your application code so that it matches the arguments (input and output) of the MATLAB functions that are encapsulated in the COM objects that you are using. The mapping of arguments from MATLAB to Visual Basic is fully described in MATLAB to COM `VARIANT` Conversion Rules on page 6-11 and COM `VARIANT` to MATLAB Conversion Rules on page 6-15.

In some cases it is not possible to match the two kinds of arguments exactly; for example, when existing MATLAB code is used in conjunction with a third party product such as Microsoft Excel. For these and other cases, COM Builder supports formatting and conversion flags that control how array data is formatted in both directions (input and output).

Overview

When it creates a component, COM Builder includes a component property named `MWFlags`. The `MWFlags` property is readable and writable.

The `MWFlags` property consists of two sets of constants: *array formatting flags* and *data conversion flags*. Array formatting flags affect the transformation of arrays, whereas data conversion flags deal with type conversions of individual array elements.

Array Formatting Flags

The following tables provide a quick overview of how to use array formatting flags to specify conversions for input and output arguments.

<table>
<tr><th>Name of Flag</th><th>Possible Values of Flag</th><th>Results of Conversion</th></tr>
<tr><td rowspan="3"><code>InputArrayFormat</code></td><td><code>mwArrayFormatMatrix</code> (default)</td><td>MATLAB matrix from general <code>Variant</code> data.</td></tr>
<tr><td><code>mwArrayFormatCell</code></td><td>MATLAB cell array from general <code>Variant</code> data.</td></tr>
<tr><td colspan="2">Array data from an Excel <code>range</code> is coded in Visual Basic as an array of <code>Variant</code>. Since MATLAB functions typically have matrix arguments, using the default setting makes sense when you are dealing with data from Excel.</td></tr>
<tr><td rowspan="8"><code>OutputArrayFormat</code></td><td><code>mwArrayFormatAsIs</code></td><td>Array of <code>Variant</code></td></tr>
<tr><td colspan="2">Converts arrays according to the default conversion rules listed in MATLAB to COM <code>VARIANT</code> Conversion Rules on page 6-11.</td></tr>
<tr><td><code>mwArrayFormatMatrix</code></td><td>A <code>Variant</code> containing an array of a basic type.</td></tr>
<tr><td><code>mwArrayFormatCell</code></td><td>MATLAB cell array from general <code>Variant</code> data.</td></tr>
<tr><td><code>AutoResizeOutput</code></td><td>When this flag is set, the target range automatically resizes to fit the resulting array. If this flag is not set, the target range must be at least as large as the output array or the data is truncated.</td></tr>
<tr><td colspan="2">Use this flag for Excel <code>Range</code> objects passed directly as output parameters.</td></tr>
<tr><td><code>TransposeOutput</code></td><td>Transposes all array output.</td></tr>
<tr><td colspan="2">Use this flag when dealing with an encapsulated MATLAB function whose output is a one-dimensional array. By default, MATLAB handles one-dimensional arrays as 1-by-n matrices (that is, as row vectors). Change this default with the <code>TransposeOutput</code> flag if you prefer column output</td></tr>
</table>

Using Array Formatting Flags

To use the following example make sure that you reference the `MWComUtil` library in the current project:

1 Click **Tools > References**.

2 Click **MWComUtil 7.1 Type Library**.

Consider the following Visual Basic function definition for `foo`.

```
Sub foo( )
   Dim aClass As mycomponent.myclass
   Dim var1(1 To 2, 1 To 2), var2 As Variant
   Dim x(1 To 2, 1 To 2) As Double
   Dim y1,y2 As Variant

   On Error Goto Handle_Error
   var1(1,1) = 11#
   var1(1,2) = 12#
   var1(2,1) = 21#
   var1(2,2) = 22#
   x(1,1) = 11
   x(1,2) = 12
   x(2,1) = 21
   x(2,2) = 22
   var2 = x
   Set aClass = New mycomponent.myclass
   Call aClass.foo(1,y1,var1)
   Call aClass.foo(1,y2,var2)
   Exit Sub
Handle_Error:
   MsgBox(Err.Description)
End Sub
```

The example has two `Variant` variables, `var1` and `var2`. These two variables contain the same numerical data, but internally they are structured differently; one is a 2-by-2 array of `variant` and the other is a 1-by-1 array of `variant`. The variables are described in the following table

	var1	var2
Numerical data	11 12 21 22	11 12 21 22
Internal structure in Visual Basic	2-by-2 array of `Variant`. Each `variant` is a 1-by-1 array of `Double`.	1-by-1 `Variant`, which contains a 2-by-2 array of `Double`
Result of conversion by COM Builder according to the default data conversion rules	2-by-2 cell array. Each element is a 1-by-1 array of `double`.	2-by-2 matrix. Each element is a `Double`.

The `InputArrayFormat` flag controls how the arrays are handled. In this example, the value for the `InputArrayFormat` flag is the default, which is `mwArrayFormatMatrix`. The default causes an array to be converted to a matrix. See the table for the result of the conversion of `var2`.

To specify a cell array (instead of a matrix) as input to the function call, set the `InputArrayFormat` flag to `mwArrayFormatCell` instead of the default. Do this in this example by adding the following line after creating the class and before the method call.

```
aClass .MWFlags.ArrayFormatFlags.InputArrayFormat =
mwArrayFormatCell
```

Setting the flag to `mwArrayFormatCell` causes all array input to the encapsulated MATLAB function to be converted to cell arrays.

Modifying Output Format. Similarly, you can manipulate the format of output arguments using the `OutputArrayFormat` flag. You can also modify array output with the `AutoResizeOutput` and `TransposeOutput` flags.

Using Data Conversion Flags

Two data conversion flags, `CoerceNumericToType` and `InputDateFormat`, govern how numeric and date types are converted from Visual Basic to MATLAB.

To use the following example make sure that you reference the `MWComUtil` library in the current project:

1 Click **Tools > References**.

2 Click **MWComUtil 7.1 Type Library**.

This example converts `var1` of type `Variant/Integer` to an `int16` and `var2` of type `Variant/Double` to a `double`.

```
Sub foo( )
   Dim aClass As mycomponent.myclass
   Dim var1, var2 As Variant
   Dim y As Variant

   On Error Goto Handle_Error
   var1 = 1
   var2 = 2#
   Set aClass = New mycomponent.myclass
   Call aClass.foo(1,y,var1,var2)
   Exit Sub
Handle_Error:
   MsgBox(Err.Description)
End Sub
```

If the original MATLAB function expects `doubles` for both arguments, this code might cause an error. One solution is to assign a `double` to `var1`, but this may not be possible or desirable. As an alternative, you can set the `CoerceNumericToType` flag to `mwTypeDouble`, causing the data converter to convert all numeric input to `double`. To do this, place the following line after creating the class and before calling the methods.

```
aClass .MWFlags.DataConversionFlags.CoerceNumericToType =
mwTypeDouble
```

The next example shows how to use the `InputDateFormat` flag, which controls how the Visual Basic `Date` type is converted. The example sends the current date and time as an input argument and converts it to a string.

```
Sub foo( )
   Dim aClass As mycomponent.myclass
```

```
   Dim today As Date
   Dim y As Variant

   On Error Goto Handle_Error
   today = Now
   Set aClass = New mycomponent.myclass
   aClass. MWFlags.DataConversionFlags.InputDateFormat =
mwDateFormatString
   Call aClass.foo(1,y,today)
   Exit Sub
Handle_Error:
   MsgBox(Err.Description)
End Sub
```

The next example uses an `MWArg` object to modify the conversion flags for one argument in a method call. In this case the first output argument (`y1`) is coerced to a `Date`, and the second output argument (`y2`) uses the current default conversion flags supplied by `aClass`.

```
Sub foo(y1 As Variant, y2 As Variant)
    Dim aClass As mycomponent.myclass
    Dim ytemp As MWArg
 Dim today As Date

 On Error Goto Handle_Error
 today = Now
 Set aClass = New mycomponent.myclass
 Set ytemp = New MWArg
 ytemp.MWFlags.DataConversionFlags.OutputAsDate = True
 Call aClass.foo(2, ytemp, y2, today)
 y1 = ytemp
 Exit Sub
Handle_Error:
 MsgBox(Err.Description)
End Sub
```

Special Flags for Some Visual Basic Types

In general, you use the `MWFlags` class property to change specified behaviors of the conversion from Visual Basic `Variant` types to MATLAB types, and vice versa. There are some exceptions — some types generated by COM Builder have their own `MWFlags` property. When you use these particular types, the method call behaves according to the settings of the type and not of the class containing the method being called. The exceptions are for the following types generated by COM Builder:

- `MWStruct`
- `MWField`
- `MWComplex`
- `MWSparse`
- `MWArg`

Note The `MWArg` class is supplied specifically for the case when a particular argument needs different settings from the default class properties.

Using MATLAB Global Variables

Global variables are variables that are declared in MATLAB with the `global` keyword. COM Builder automatically converts all global variables shared by the M-files that make up a class to properties on that class. Class properties allow an object to retain an internal state between method calls.

Properties are particularly useful when you have a large array containing values that do not change often, but you need to operate on it frequently. In this case, you can set the array once as a class property and operate on it repeatedly without incurring the overhead of passing (and converting) the data for passing to each method every time it is called.

Using MATLAB Global Variables in Visual Basic

The following example shows how to use a class property in a matrix factorization class. The example develops a class that performs Cholesky, LU, and QR factorizations on the same matrix. It stores the input matrix (coded as `A` in MATLAB) as a class property so that it does not need to be passed to the factorization routines.

Consider these three M-files.

`Cholesky.m`

```
function [L] = Cholesky()
    global A;
    if (isempty(A))
        L = [];
        return;
    end
    L = chol(A);
```

`LUDecomp.m`

```
function [L,U] = LUDecomp()
    global A;
    if (isempty(A))
        L = [];
        U = [];
```

```
        return;
    end
    [L,U] = lu(A);
```

QRDecomp.m

```
function [Q,R] = QRDecomp()
    global A;
    if (isempty(A))
        Q = [];
        R = [];
        return;
    end
    [Q,R] = qr(A);
```

These three files share a common global variable `A`. Each function performs a matrix factorization on `A` and returns the results.

To build the class:

1 Create a new COM Builder project named `mymatrix` with a version of 1.0.

2 Add a single class called `myfactor` to the component.

3 Add the above three M-files to the class.

4 Build the component.

To test your application, make sure that you reference the library generated by COM Builder in the current Visual Basic project:

1 Click **Project > References** in the Visual Basic main menu.

2 Click **mymatrix 1.0 Type Library**.

Use the following Visual Basic subroutine to test the `myfactor` class.

Sub TestFactor

```
Sub TestFactor()
    Dim x(1 To 2, 1 To 2) As Double
```

```
        Dim C As Variant, L As Variant, U As Variant, _
        Q As Variant, R As Variant
        Dim factor As myfactor

        On Error GoTo Handle_Error
        Set factor = New myfactor
        x(1, 1) = 2#
        x(1, 2) = -1#
        x(2, 1) = -1#
        x(2, 2) = 2#
        factor.A = x
        Call factor.cholesky(1, C)
        Call factor.ludecomp(2, L, U)
        Call factor.qrdecomp(2, Q, R)
        Exit Sub
    Handle_Error:
        MsgBox (Err.Description)
    End Sub
```

Run the subroutine, which does the following:

1. Creates an instance of the `myfactor` class.
2. Assigns a double matrix to the property `A`.
3. Calls the three factorization methods.

Obtaining Registry Information

When programming with COM components you might need details about a component. You can use componentinfo, which is a MATLAB function, to query the system registry for details about any installed MATLAB Builder for COM component.

This example queries the registry for a component named mycomponent and a version of 1.0. This component has four methods: mysum, randvectors, getdates, and myprimes, two properties: m and n, and one event: myevent.

```
Info = componentinfo('mycomponent', 1, 0)

Info =

       Name: 'mycomponent'
      TypeLib: 'mycomponent 1.0 Type Library'
        LIBID: '{3A14AB34-44BE-11D5-B155-00D0B7BA7544}'
    MajorRev: 1
    MinorRev: 0
    FileName: 'D:\Work\ mycomponent\distrib\mycomponent_1_0.dll'
    Interfaces: [1x1 struct]
    CoClasses: [1x1 struct]

Info.Interfaces

ans =

      Name: 'Imyclass'
       IID: '{3A14AB36-44BE-11D5-B155-00D0B7BA7544}'

Info.CoClasses

ans =

          Name: 'myclass'
         CLSID: '{3A14AB35-44BE-11D5-B155-00D0B7BA7544}'
         ProgID: 'mycomponent.myclass.1_0'
   VerIndProgID: 'mycomponent.myclass'
InprocServer32:'D:\Work\mycomponent\distrib\mycomponent_1_0.dll'
```

```
       Methods: [1x4 struct]
    Properties: {'m', 'n'}
        Events: [1x1 struct]

Info.CoClasses.Events.M

ans =

function myevent(x, y)

Info.CoClasses.Methods

ans =

1x4 struct array with fields:
    IDL
    M
    C
    VB

Info.CoClasses.Methods.M

ans =

function [y] = mysum(varargin)

ans =

function [varargout] = randvectors()

ans =

function [x] = getdates(n, inc)

ans =

function [p] = myprimes(n)
```

The returned structure contains fields corresponding to the most important information from the registry and type library for the component.

Handling Errors During a Method Call

If your application generates an error while creating a class instance or during a class method call, the current procedure creates an exception.

Visual Basic provides an exception handling capability through the On Error Goto <label> statement, in which the program execution jumps to <label> when an error occurs. (<label> must be located in the same procedure as the On Error Goto statement). All errors in Visual Basic are handled this way, including errors within the MATLAB code that you have encapsulated into a COM object. An exception creates a Visual Basic ErrObject object in the current context in a variable called Err.

See the Visual Basic documentation for a detailed discussion on Visual Basic error handling.

Usage Examples

Magic Square Example (p. 4-2)	Demonstrates the creation of a COM component from a simple MATLAB M-file.
Spectral Analysis Example (p. 4-11)	Shows the creation of a comprehensive Excel add-in.
Univariate Interpolation (p. 4-27)	Uses Akima's Univariate Interpolation example available on the MathWorks Web site.
Matrix Calculator (p. 4-39)	Creates a matrix calculator and shows how to compile the MATLAB functions into a COM component.
Curve Fitting (p. 4-52)	Demonstrates the optimal fitting of a nonlinear function to a set of data.
Bouncing Ball Simulation (p. 4-63)	An adaptation of the `ballode` demo provided with core MATLAB.

Magic Square Example

This example uses a simple M-file that takes a single input and creates a magic square of that size. It then builds a MATLAB Builder for COM component using this M-file as a class method. Finally, the example shows the integration of this component into a stand-alone Visual Basic application. The application accepts the magic square size as input and displays the matrix in a ListView control box.

Note ListView is a Windows Form control that displays a list of items with icons. You can use a list view to create a user interface like the right pane of Windows Explorer. See the MSDN Libray for more information about Windows Form controls.

Creating the M-File

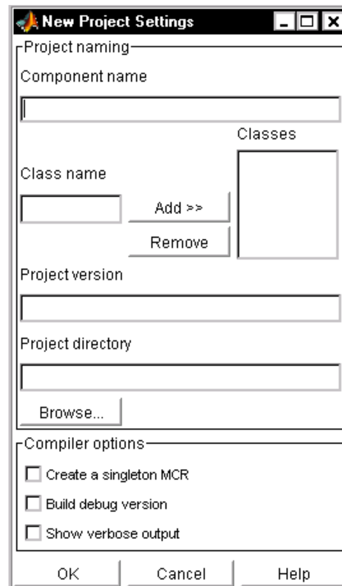
To get started, create the M-file `mymagic.m` containing the following code:

```
function y = mymagic(x)
y = magic(x);
```

Creating the Project

Enter the command `comtool` to display the MATLAB Builder window.

Click **File > New Project** to open the New Project Settings dialog box, as shown.



Empty New Project Settings Dialog Box

In the New Project Settings dialog box, enter the settings for this example:

1 In the **Component name** text block, type the component name `magicdemo` and press **Enter**.

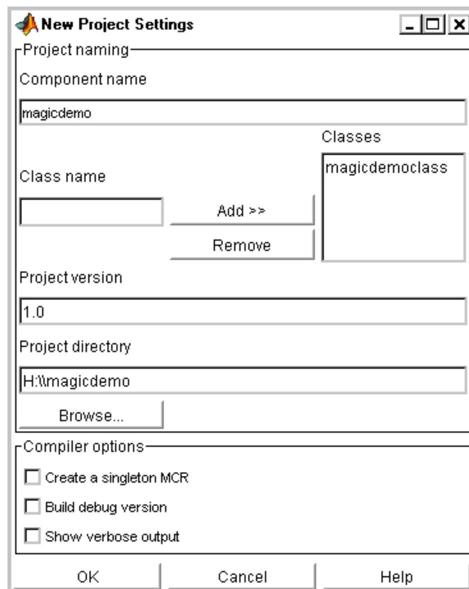
   COM Builder automatically creates `magicdemoclass` as the class name based on the component name and lists the class in the **Classes** field.

2 The version has a default of 1.0. Leave this number unchanged.

3 The **Project directory** field contains a default combination of the directory where COM Builder was started and the component name, `magicdemo`. You can change this to any directory that you choose. If the directory you choose does not exist, you will be asked to create it.

4 Leave all compiler options unselected.

The New Project Settings dialog box now looks like the following figure:



New Project Settings with Entries

- Click **OK** to create the `magicdemo` project.

Summary of Project Settings

Component name: `magicdemo`

Class name: `magicdemoclass`

Project version: `1.0`

Project directory: (*accept default or choose another directory*)

Compiler options: (*leave all unselected*)

Building the Project

- In the MATLAB Builder window, select the `magicdemoclass` folder..
- Click **Add File**.

- Select the file `mymagic.m` from the directory where you saved it and click **Open**.
- Click **Build > COM Object**.

Creating the Visual Basic Project

Note This procedure assumes that you are using Visual Basic 6.0.

1. Start Visual Basic.
2. In the New Project dialog box, select **Standard EXE** as the project type and click **Open**. This creates a new Visual Basic project with a blank form.
3. From the main menu, click **Project > References** to display the Project References dialog box.
4. Select **magicdemo 1.0 Type Library** from the list of available components.
5. Returning to the Visual Basic main menu, click **Project > Components** to display the Components dialog box.
6. Click **Microsoft Windows Common Controls 6.0**. You will use the `ListView` control from this component library.

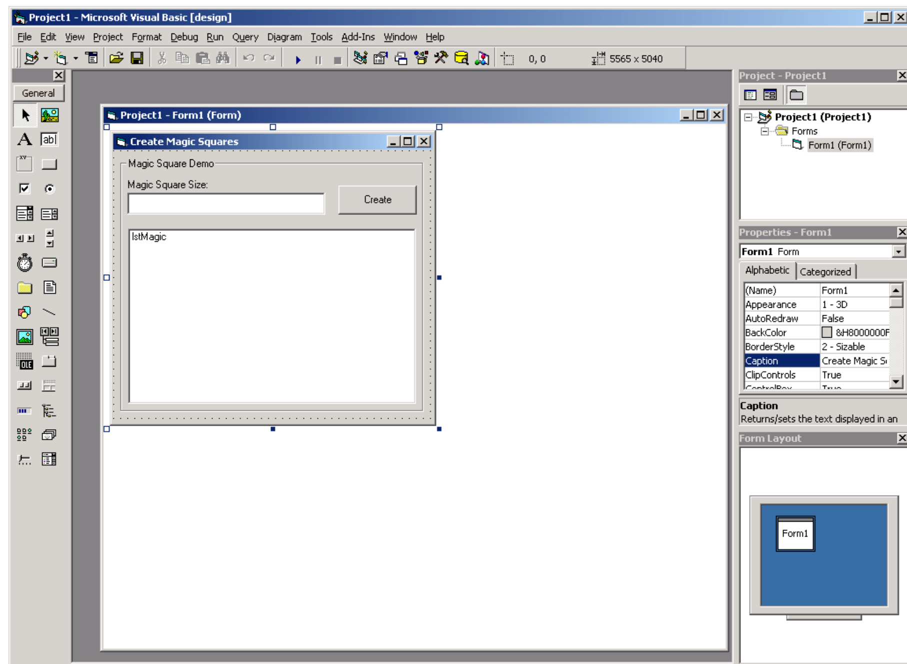
Creating the User Interface

After you create the project, add a series of controls to the blank form. The required settings are summarized in the following table.

Control Type	Control Name	Properties	Purpose
`Frame`	`Frame1`	Caption = `Magic Squares Demo`	Groups controls
`Label`	`Label1`	Caption = `Magic Square Size`	Labels the magic square edit box.

Control Type	Control Name	Properties	Purpose
TextBox	edtSize		Accepts input of magic square size.
CommandButton	btnCreate	Caption = Create	When pressed, creates a new magic square with current size.
ListView	lstMagic	GridLines = True LabelEdit = lvwManual View = lvwReport	Displays the magic square.

The following figure shows the controls layout on the form you created:



When the form and controls are complete, add the code below to the form. This code references the control and variable names listed above. If you have

given different names for any of the controls or any variable, change this code to reflect those differences.

```
Private Size As Double 'Holds current matrix size
Private theMagic As magicdemo.magic 'magic object instance

Private Sub Form_Load()
'This function is called when the form is loaded.
'Creates a new magic class instance.
    On Error GoTo Handle_Error
    Set theMagic = New magicdemo.magic
    Size = 0
Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub

Private Sub btnCreate_Click()
'This function is called when the Create button is pressed.
'Calls the mymagic method, and displays the magic square.
    Dim y As Variant
    If Size <= 0 Or theMagic Is Nothing Then Exit Sub
    On Error GoTo Handle_Error
    Call theMagic.mymagic(1, y, Size)
    Call ShowMatrix(y)
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub

Private Sub edtSize_Change()
'This function is called when ever the contents of the
'Text box change. Sets the current value of Size.
    On Error Resume Next
    Size = CDbl(edtSize.Text)
    If Err <> 0 Then
        Size = 0
    End If
End Sub
```

```
Private Sub ShowMatrix(y As Variant)
'This function populates the ListView with the contents of
'y. y is assumed to contain a 2D array.
    Dim n As Long
    Dim i As Long
    Dim j As Long
    Dim nLen As Long
    Dim Item As ListItem

    On Error GoTo Handle_Error
    'Get array size
    If IsArray(y) Then
        n = UBound(y, 1)
    Else
        n = 1
    End If
    'Set up Column headers
    nLen = lstMagic.Width / 5
    Call lstMagic.ListItems.Clear
    Call lstMagic.ColumnHeaders.Clear
    Call lstMagic.ColumnHeaders.Add(, , "", nLen, lvwColumnLeft)
    For i = 1 To n
        Call lstMagic.ColumnHeaders.Add(, , _
        "Column " & Format(i), nLen, lvwColumnLeft)
    Next
    'Add array contents
    If IsArray(y) Then
        For i = 1 To n
            Set Item = lstMagic.ListItems.Add(, , "Row " & Format(i))
            For j = 1 To n
                Call Item.ListSubItems.Add(, , Format(y(i, j)))
            Next
        Next
    Else
        Set Item = lstMagic.ListItems.Add(, , "Row 1")
        Call Item.ListSubItems.Add(, , Format(y))
    End If
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
```

```
End Sub
```

Creating the Executable

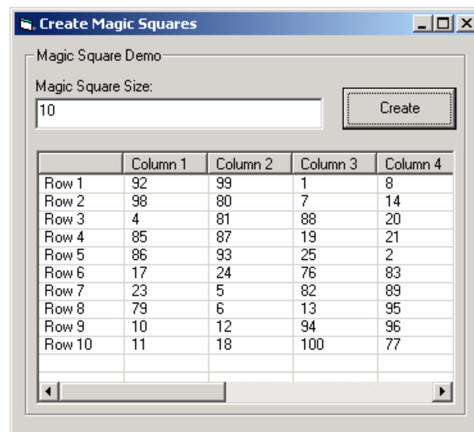
After the code is complete, create the stand-alone executable `magic.exe`:

1. Save the project by clicking **File > Save Project** from the main menu. Accept the default name for the main form and enter `magic.vbp` for the project name.

2. Return to the **File** menu. Click **File > Make magic.exe** to create the finished product.

Testing the Application

You can run the `magic.exe` executable as you would any other program. When the main dialog box starts, enter a positive number in the input box and click the **Create** button. A magic square of the input size appears as shown:



The `ListView` control automatically implements scrolling if the magic square is larger than 4-by-4.

Packaging the Component

As a final step, package the `magicdemo` component and all supporting libraries into a self-extracting executable. Then anyone can install the package onto

another computer, in particular a computer without MATLAB installed, and use the magicdemo application.

To package the component, follow these steps:

1 Return to the MATLAB Builder window. If necessary, type comtool in the Command window and open the magicdemo project.

2 Click **Component > Package Component**. This command creates the magicdemo.exe self-extracting executable.

To install the component onto another computer, copy the magicdemo.exe package and to that machine, run magicdemo.exe from a command prompt, and follow the instructions.

Spectral Analysis Example

This example shows how to create a comprehensive Excel add-in to perform spectral analysis. It requires knowledge of Visual Basic forms and controls, as well as Excel workbook events. See the Visual Basic documentation included with Microsoft Excel for a complete discussion of these topics.

The example creates an Excel add-in that performs a fast Fourier transform (FFT) on an input data set located in a designated worksheet range. The function returns the FFT results, an array of frequency points, and the power spectral density of the input data. It places these results into ranges you indicate in the current worksheet. You can also optionally plot the power spectral density. You develop the function so that you can invoke it from the Excel **Tools** menu and can select input and output ranges through a graphical user interface (GUI).

To create this add-in requires four basic steps:

1. Build a stand-alone COM component from MATLAB code.
2. Implement the necessary Visual Basic Application (VBA) code to collect input and dispatch the calls to your component.
3. Create the GUI.
4. Create an Excel add-in and package all necessary components for application deployment.

Building the Component

Your component will have one class with two methods, `computefft` and `plotfft`.

- The `computefft` method 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.
- The `plotfft` method performs the same operations as `computefft`, but also plots the input data and the power spectral density in a MATLAB figure window.

The MATLAB code for these two methods resides in two M-files, `computefft.m` and `plotfft.m`, as shown:

```
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 then zero');
        return;
    end
    fftdata = fft(data);
    freq = (0:length(fftdata)-1)/(length(fftdata)*interval);
    powerspect = abs(fftdata)/(sqrt(length(fftdata)));


plotfft.m:

function [fftdata, freq, powerspect] = plotfft(data, interval)
    [fftdata, freq, powerspect] = computefft(data, interval);
    len = length(fftdata);
    if (len <= 0)
        return;
    end
    t = 0:interval:(len-1)*interval;
    subplot(2,1,1), plot(t, data)
    xlabel('Time'), grid on
    title('Time domain signal')
    subplot(2,1,2), plot(freq(1:len/2), powerspect(1:len/2))
    xlabel('Frequency (Hz)'), grid on
    title('Power spectral density')
```

To build the component, follow these steps:

1 Start `comtool`.

2 Create a new project with these settings:

   - **Component name**: `Fourier`
   - **Class name**: `Fourier`
   - **Project version**: `1.0`

   See "Project Settings Window" on page 2-6 for a description of new project settings.

3 Add the `computefft.m` and `plotfft.m` M-files to the project.

4 Save the project.

5 Click **Build > COM Object** to create the component.

Integrating the Component with VBA

The next step is to implement the necessary VBA code to integrate the component into Excel.

Follow these steps to open Excel and select the libraries you need to develop the add-in:

1 Start Excel.

2 From the Excel main menu, click **Tools > Macro > Visual Basic Editor**.

3 When the Visual Basic Editor starts, click **Tools > References**to display the Project References dialog.

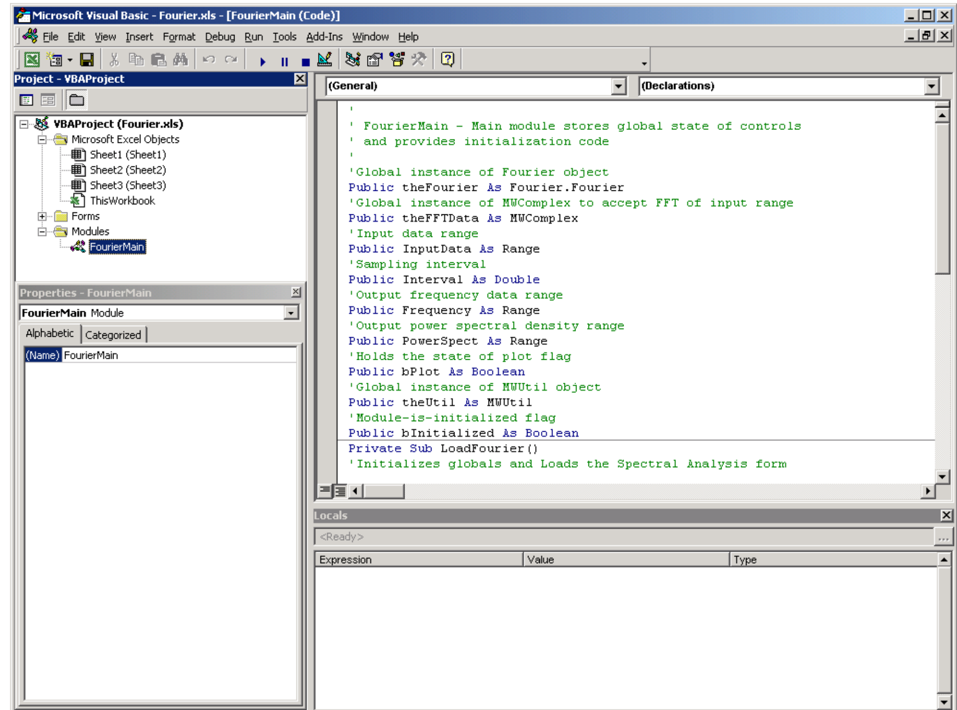
4 Select **Fourier 1.0 Type Library** and **MWComUtil 7.1 Type Library** .

Creating the Main VBA Code Module

The add-in requires some initialization code and some global variables to hold the application's state between function invocations. To achieve this, implement a Visual Basic code module to manage these tasks, as follows:

1 Right-click **VBAProject** in the project window and click **Insert > Module**.

2 A new module appears under **Modules** in the **VBA Project**. In the module's property page, set the **Name** property to `FourierMain`, as shown:



3 Enter the following code in the `FourierMain` module:

```
' FourierMain - Main module stores global state of controls
' and provides initialization code
'
'Global instance of Fourier object
Public theFourier As Fourier.Fourier
'Global instance of MWComplex to accept FFT
Public theFFTData As MWComplex
'Input data range
Public InputData As Range
'Sampling interval
Public Interval As Double
'Output frequency data range
```

```
Public Frequency As Range
'Output power spectral density range
Public PowerSpect As Range
'Holds the state of plot flag
Public bPlot As Boolean
'Global instance of MWUtil object
Public theUtil as MWUtil object
'Module-is-initialized flag
Public bInitialized As Boolean
Private Sub LoadFourier()
'Initializes globals and Loads the Spectral Analysis form
    Dim MainForm As frmFourier
    On Error GoTo Handle_Error
    Call InitApp
    Set MainForm = New frmFourier
    Call MainForm.Show
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub

Private Sub InitApp()
'Initializes classes and libraries. Executes once
'for a given session of Excel
    If bInitialized Then Exit Sub
    On Error GoTo Handle_Error
    If theFourier Is Nothing Then
        Set theFourier = New Fourier.Fourier
    End If
    If theFFTData Is Nothing Then
        Set theFFTData = New MWComplex
    End If
    bInitialized = True
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
```

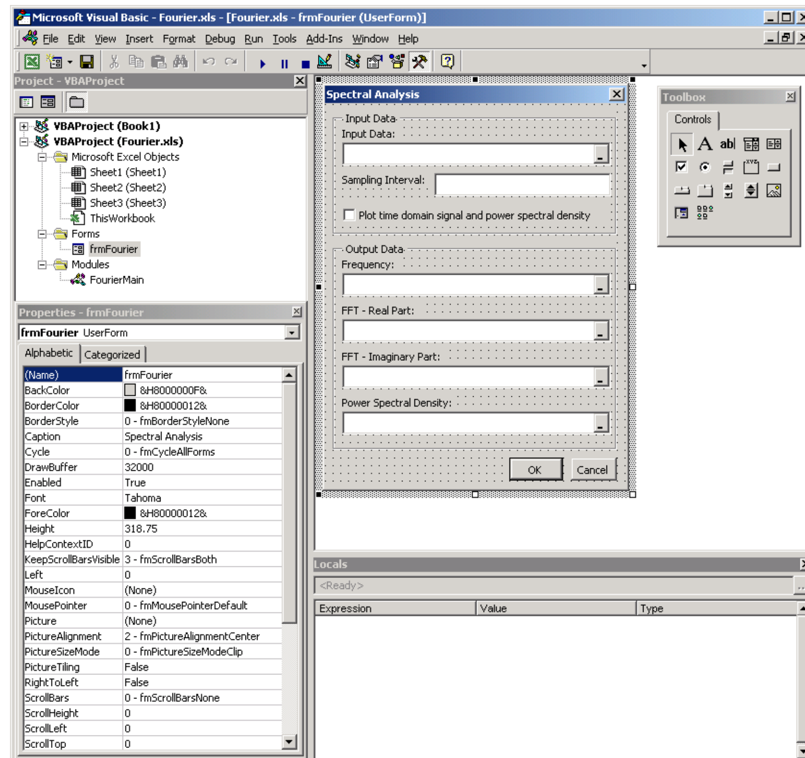
Creating the Visual Basic Form

The next step is to develop a user interface for your add-in using the Visual Basic editor. Follow these steps to create a new user form and populate it with the necessary controls:

1 Right-click **VBAProject** item in the project window and click **Insert > UserForm**.

A new form appears under **Forms** in the VBA Project.

2 In the form's property page, set the name property to `frmFourier` and the `Caption` property to `Spectral Analysis`, as shown:

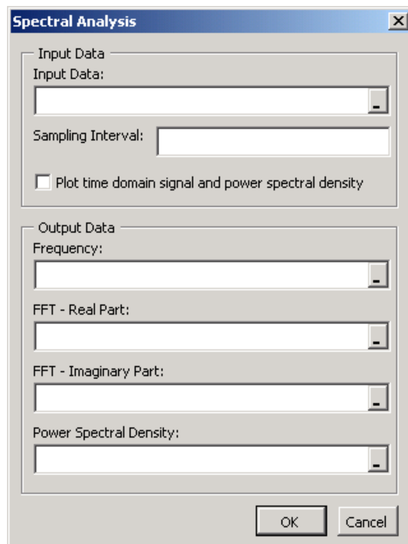


3 Add a series of controls to the blank form to complete the dialog, as summarized in the following table.

Control Type	Control Name	Properties	Purpose
Frame	Frame1	Caption = Input Data	Groups all input controls.
Label	Label1	Caption = Input Data:	Labels the RefEdit for input data.
RefEdit	refedtInput		Selects range for input data.
Label	Label2	Caption = Sampling Interval	Labels the text box for sampling interval.
TextBox	edtSample		Specifies the sampling interval.
CheckBox	chkPlot	Caption = Plot time domain Signal and Power Spectral Density	Plots input data and power spectral density.
Frame	Frame2	Caption = Output Data	Groups all output controls.
Label	Label3	Caption = Frequency:	Labels the RefEdit for frequency output.
RefEdit	refedtFreq		Selects output range for frequency points.
Label	Label4	Caption = FFT - Real Part:	Labels the RefEdit for real part of FFT.
RefEdit	refedtReal		Selects output range for real part of FFT of input data.
Label	Label5	Caption = FFT - Imaginary Part:	Labels the RefEdit for imaginary part of FFT.
RefEdit	refedtImag		Selects output range for imaginary part of FFT of input data.

Control Type	Control Name	Properties	Purpose
`Label`	`Label6`	`Caption = Power Spectral Density`	Labels the `RefEdit` for power spectral density.
`RefEdit`	`refedtPowSpect`		Selects output range for power spectral density of input data.
`CommandButton`	`btnOK`	Caption = `OK` Default = `True`	Executes the function and dismisses the dialog
`CommandButton`	`btnCancel`	Caption = `Cancel` Cancel = `True`	Dismisses the dialog box without executing the function.

The following figure shows the resulting layout.



When the form and controls are complete, right-click anywhere in the form and click **View Code**. The following code listing shows the code to implement. Note that this code references the control and variable names

listed above. If you have given different names for any of the controls or any global variable, change this code to reflect those differences.

```
'
'frmFourier Event handlers
'
Private Sub UserForm_Activate()
'UserForm Activate event handler. This function gets called before
'showing the form, and initializes all controls with values stored
'in global variables.
    On Error GoTo Handle_Error
    If theFourier Is Nothing Or theFFTData Is Nothing Then Exit Sub
    'Initialize controls with current state
    If Not InputData Is Nothing Then
        refedtInput.Text = InputData.Address
    End If
    edtSample.Text = Format(Interval)
    If Not Frequency Is Nothing Then
        refedtFreq.Text = Frequency.Address
    End If
    If Not IsEmpty (theFFTData.Real) Then
    If IsObject(theFFTData.Real) And TypeOf theFFTData.Real Is Range Then
            refedtReal.Text = theFFTData.Real.Address
        End If
    End If
    If Not IsEmpty (theFFTData.Imag) Then
    If IsObject(theFFTData.Imag) And TypeOf theFFTData.Imag Is Range Then
            refedtImag.Text = theFFTData.Imag.Address
        End If
    End If
    If Not PowerSpect Is Nothing Then
        refedtPowSpect.Text = PowerSpect.Address
    End If
    chkPlot.Value = bPlot
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub

Private Sub btnCancel_Click()
```

```
'Cancel button click event handler. Exits form without computing fft
'or updating variables.
    Unload Me
End Sub
Private Sub btnOK_Click()
'OK button click event handler. Updates state of all variables from controls
'and executes the computefft or plotfft method.
    Dim R As Range

    If theFourier Is Nothing Or theFFTData Is Nothing Then GoTo Exit_Form
    On Error Resume Next
    'Process inputs
    Set R = Range(refedtInput.Text)
    If Err <> 0 Then
        MsgBox ("Invalid range entered for Input Data")
        Exit Sub
    End If
    Set InputData = R
    Interval = CDbl(edtSample.Text)
    If Err <> 0 Or Interval <= 0 Then
        MsgBox ("Sampling interval must be greater than zero")
        Exit Sub
    End If
    'Process Outputs
    Set R = Range(refedtFreq.Text)
    If Err = 0 Then
        Set Frequency = R
    End If
    Set R = Range(refedtReal.Text)
    If Err = 0 Then
        theFFTData.Real = R
    End If
    Set R = Range(refedtImag.Text)
    If Err = 0 Then
        theFFTData.Imag = R
    End If
    Set R = Range(refedtPowSpect.Text)
    If Err = 0 Then
        Set PowerSpect = R
    End If
```

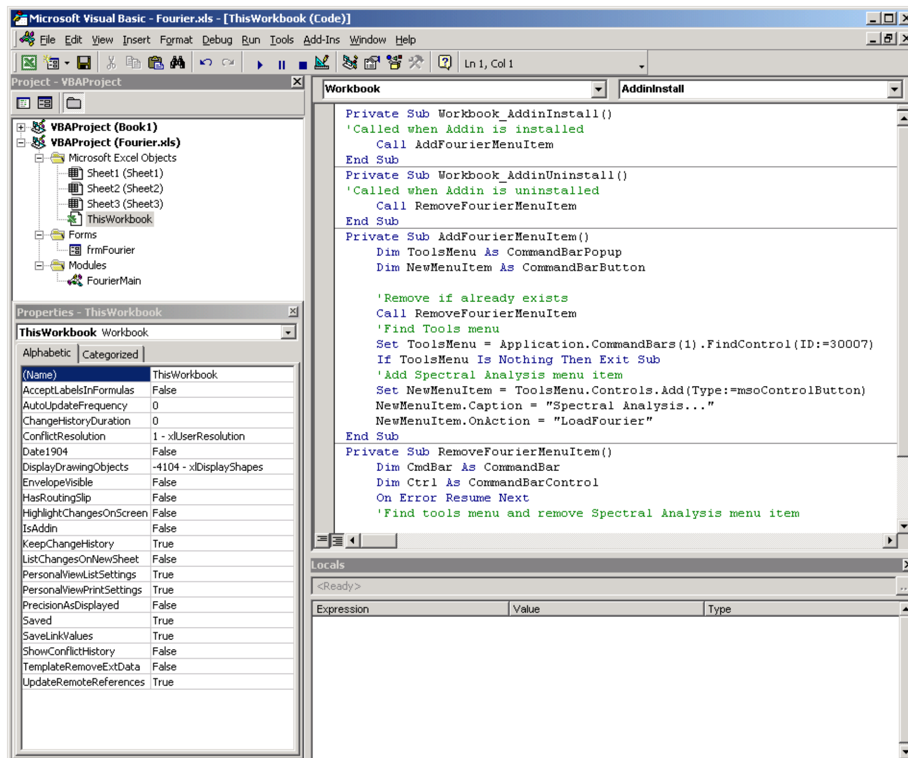
```
    bPlot = chkPlot.Value
    'Compute the fft and optionally plot power spectral density
    If bPlot Then
        Call theFourier.plotfft(3, theFFTData, Frequency, PowerSpect,_
InputData, Interval)
    Else
        Call theFourier.computefft(3, theFFTData, Frequency, PowerSpect,_
InputData, Interval)
    End If
    GoTo Exit_Form
Handle_Error:
    MsgBox (Err.Description)
Exit_Form:
    Unload Me
End Sub
```

Adding The Spectral Analysis Menu Item to Excel

The last step in the integration process is to add a menu item to Excel so that you can invoke the tool from Excel's **Tools** menu. To do this you add event handlers for the workbook's `AddinInstall` and `AddinUninstall` events; these are events that install and uninstall menu items. The menu item calls the `LoadFourier` function in the `FourierMain` module.

Follow these steps to implement the menu item:

1 Right-click **ThisWorkbook** in the Visual Basic project window and click **View Code**.



2 Add the following code to the **ThisWorkbook** object.

```
Private Sub Workbook_AddinInstall()
'Called when Addin is installed
    Call AddFourierMenuItem
End Sub

Private Sub Workbook_AddinUninstall()
'Called when Addin is uninstalled
    Call RemoveFourierMenuItem
End Sub

Private Sub AddFourierMenuItem()
    Dim ToolsMenu As CommandBarPopup
    Dim NewMenuItem As CommandBarButton
```

```
    'Remove if already exists
    Call RemoveFourierMenuItem
    'Find Tools menu
    Set ToolsMenu = Application.CommandBars(1).FindControl(ID:=30007)
    If ToolsMenu Is Nothing Then Exit Sub
    'Add Spectral Analysis menu item
    Set NewMenuItem = ToolsMenu.Controls.Add(Type:=msoControlButton)
    NewMenuItem.Caption = "Spectral Analysis..."
    NewMenuItem.OnAction = "LoadFourier"
End Sub

Private Sub RemoveFourierMenuItem()
Dim CmdBar As CommandBar
Dim Ctrl As CommandBarControl
On Error Resume Next
'Find tools menu and remove Spectral Analysis menu item
Set CmdBar = Application.CommandBars(1)
Set Ctrl = CmdBar.FindControl(ID:=30007)
Call Ctrl.Controls("Spectral Analysis...").Delete
End Sub
```

Saving the Add-in

Name the add-in `Spectral Analysis` and follow these steps to save it:

1. From the main menu in Excel, click **File > Properties**.

   The **Workbook Properties** dialog box appears.

2. Click the **Summary** tab and enter `Spectral Analysis` as the workbook title.

3. Click **OK** to save the edits.

4. Click **File > Save As** from the Excel main menu.

5. Click **Microsoft Excel Add-In (*.xla)** as the file type.

6. Enter `Fourier.xla` as the filename.

7 Click **Save** to save the add-in.

Testing The Add-in

Before distributing the add-in, test it with a sample problem. Spectral analysis is commonly used to find the frequency components of a signal buried in a noisy time domain signal. In this example you will create a data representation of a signal containing two distinct components and add to it a random component. This data along with the output will be stored in columns of an Excel worksheet, and you will plot the time-domain signal along with the power spectral density.

Follow the steps outlined below to create the test problem:

1. Start a new session of Excel with a blank workbook.
2. Click **Tools > Add-Ins** from the main menu.
3. When the **Add-Ins** dialog box comes up, click **Browse**.
4. Browse to the `Fourier.xla` file and click **OK**.
5. The `Spectral Analysis` add-in appears in the available **Add-Ins** list and is selected.
6. Click **OK** to load the add-in.

This add-in installs a menu item under the Excel **Tools** menu. You can display the Spectral Analysis GUI by clicking **Tools > Spectral Analysis**.

Before invoking the add-in, create some data, in this case a signal with components at 15 and 40 Hz. Sample the signal for 10 seconds at a sampling rate of 0.01 second. Put the time points into column A and the signal points into column B.

Creating the Data

Follow these steps to create the data:

1. Enter `0` for cell A1 in the current worksheet.
2. Click on cell A2 and type the formula `= A1 + 0.01`.

3 Drag the formula in cell A2 down the column to cell A1001.

   This procedure fills the range A1:A1001 with the interval 0 to 10 incremented by 0.01.

4 Click on cell B1 and type the formula `SIN(2*PI()*15*A1) + SIN(2*PI()*40*A1) + RAND()`.

5 Repeat the drag procedure to copy this formula to all cells in the range B1:B1001.

Running the Test

Using the column of data (column B), test the add-in as follows:

1 Click **Tools > Spectral Analysis** from the main menu.

2 Click **Input Data**.

3 Click the B1:B1001 range from the worksheet, or type this address into **Input Data**.

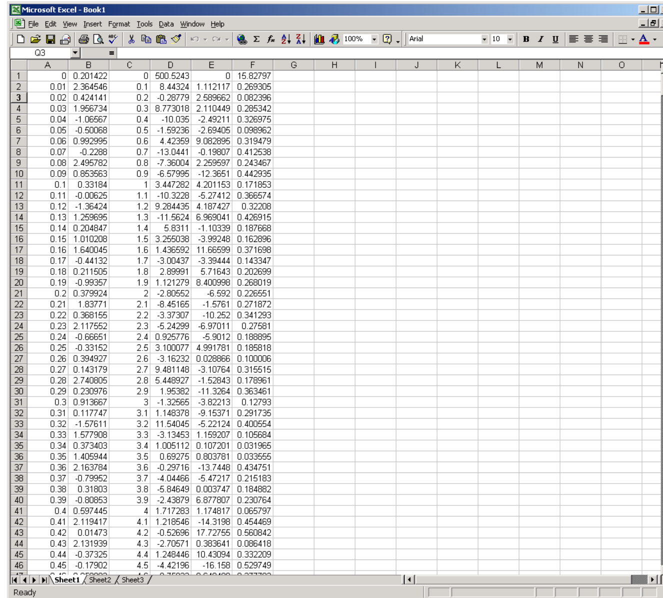
4 Click **Sampling Interval** box and type `0.01`.

5 Click **Plot time domain signal and power spectral density**.

6 Enter `C1:C1001` for frequency output. Similarly, enter `D1:D1001`, `E1:E1001`, and `F1:F1001` for the FFT real and imaginary parts, and spectral density.

7 Click **OK** to run the analysis.

   The following figure shows the output.



	A	B	C	D	E	F
1	0	0.201422	0	500.5243	0	15.82797
2	0.01	2.364546	0.1	8.44324	1.112117	0.269305
3	0.02	0.424141	0.2	-0.28779	2.589662	0.082396
4	0.03	1.956734	0.3	8.773018	2.110449	0.285342
5	0.04	-1.06567	0.4	-10.035	-2.49211	0.326975
6	0.05	-0.50068	0.5	-1.59236	-2.69405	0.098962
7	0.06	0.992995	0.6	4.42359	9.082895	0.319479
8	0.07	-0.2288	0.7	-13.0441	-0.19807	0.412538
9	0.08	2.495782	0.8	-7.36004	2.259597	0.243467
10	0.09	0.853563	0.9	-6.57995	-12.3651	0.442935
11	0.1	0.33184	1	3.447282	4.201153	0.171853
12	0.11	-0.00625	1.1	-10.3228	-5.27412	0.366574
13	0.12	-1.36424	1.2	9.284435	4.187427	0.32208
14	0.13	1.259695	1.3	-11.5624	6.969041	0.426915
15	0.14	0.204847	1.4	5.8311	-1.10339	0.187668
16	0.15	1.010208	1.5	3.255038	-3.99248	0.162896
17	0.16	1.640045	1.6	1.436592	11.66599	0.371698
18	0.17	-0.44132	1.7	-3.00437	-3.39444	0.143347
19	0.18	0.211505	1.8	2.89991	5.71643	0.202699
20	0.19	-0.99357	1.9	1.121279	8.400998	0.268019
21	0.2	0.379924	2	-2.80552	-6.592	0.226551
22	0.21	1.83771	2.1	-8.45165	-1.5761	0.271872
23	0.22	0.368155	2.2	-3.37307	-10.252	0.341293
24	0.23	2.117552	2.3	-5.24299	-6.97011	0.27581
25	0.24	-0.66651	2.4	0.925776	-5.9012	0.188895
26	0.25	-0.33152	2.5	3.100077	4.991781	0.185818
27	0.26	0.394927	2.6	-3.16232	0.028866	0.100006
28	0.27	0.143179	2.7	9.481148	-3.10764	0.315515
29	0.28	2.740805	2.8	5.448927	-1.52843	0.178961
30	0.29	0.230976	2.9	1.95382	-11.3264	0.363461
31	0.3	0.913667	3	-1.32565	-3.82213	0.12793
32	0.31	0.117747	3.1	1.148378	-9.15371	0.291735
33	0.32	-1.57611	3.2	11.54045	-5.22124	0.400554
34	0.33	1.577908	3.3	-3.13453	1.159207	0.105684
35	0.34	0.373403	3.4	1.005112	0.107201	0.031965
36	0.35	1.405944	3.5	0.69275	0.803781	0.033555
37	0.36	2.163784	3.6	-0.29716	-13.7448	0.434751
38	0.37	-0.79952	3.7	-4.04466	-5.47217	0.215183
39	0.38	0.31803	3.8	-5.84649	0.003747	0.184882
40	0.39	-0.80853	3.9	-2.43879	6.877807	0.230764
41	0.4	0.597445	4	1.717283	1.174817	0.065797
42	0.41	2.119417	4.1	1.218546	-14.3198	0.454469
43	0.42	0.01473	4.2	-0.52696	17.72755	0.560842
44	0.43	2.131939	4.3	-2.70571	0.383641	0.086418
45	0.44	-0.37325	4.4	1.248446	10.43094	0.332209
46	0.45	-0.17902	4.5	-4.42196	-16.158	0.529749

The power spectral density reveals the two signals at 15 and 40 Hz.

Package the Component

As a final step, package the COM component and all supporting libraries into a self-extracting executable. The package can then be installed on other computers that need to use the Spectral Analysis component. You will also need to copy the `Fourier.xla` file to any machine that will use this component from inside Excel.

To package the component, follow these steps:

1 Return to the MATLAB Builder window. If necessary, issue the `comtool` command and reload the Fourier project.

2 Click **Component > Package Component**.

This command creates the `Fourier.exe` self-extracting executable. To install this component on another computer, copy the `Fourier.exe` package to that machine, run it from a command prompt, and follow the instructions.

Univariate Interpolation

This example is created using the Akima's Univariate Interpolation example posted by N. Shyamsundar on the MathWorks Web site. You can download the original M-file from `http://www.mathworks.com/matlabcentral/`.

This example shows you how to create the COM component using MATLAB Builder for COM and how to use this COM component in external Microsoft Visual Basic Code independent of MATLAB. It assumes that you have downloaded the M-file to the *`matlab/work`* directory.

Building the Component

Build the component as follows:

1. From the MATLAB command prompt change directories to *`matlab/work`*.

2. Enter the command `comtool`.

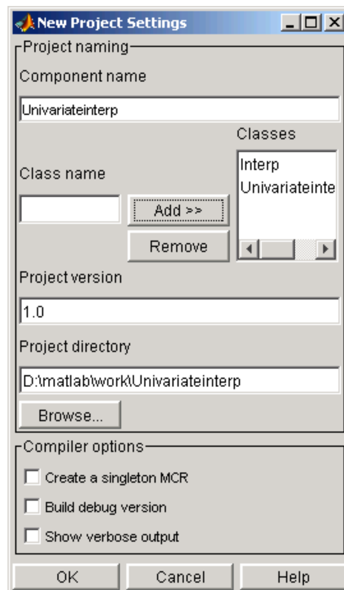
3. In the MATLAB Builder window, click **File > New Project**.

   This opens the New Project Settings dialog box.

Enter the following settings:

- In the **Component name** field enter the component name `UnivariateInterp`. Press the **Tab** key to move to the **Class name** field.
- Enter the **Class name** `Interp`.
- Click **Add**. This adds the class name `Interp` to the list of classes in the **Classes** field.
- The version has a default of 1.0. Leave this number unchanged.
- The **Project directory** field contains a default of a combination of the directory where MATLAB Builder for COM was started and the **Component name**, `Univariateinterp`. You can change this to any directory that you choose. If the directory you choose does not exist, you will be asked to create it.
- Leave all compiler options unselected.

The New Project Settings dialog box now looks as shown below



Project Settings for Univariate Interpolation Project

- Click **OK** to create the `Univariateinterp` project.

Summary of Project Settings

Component name: `Univariateinterp`

Class name: `Univariateinterp`

Project version: `1.0`

Project directory: (*accept default or choose another directory*)

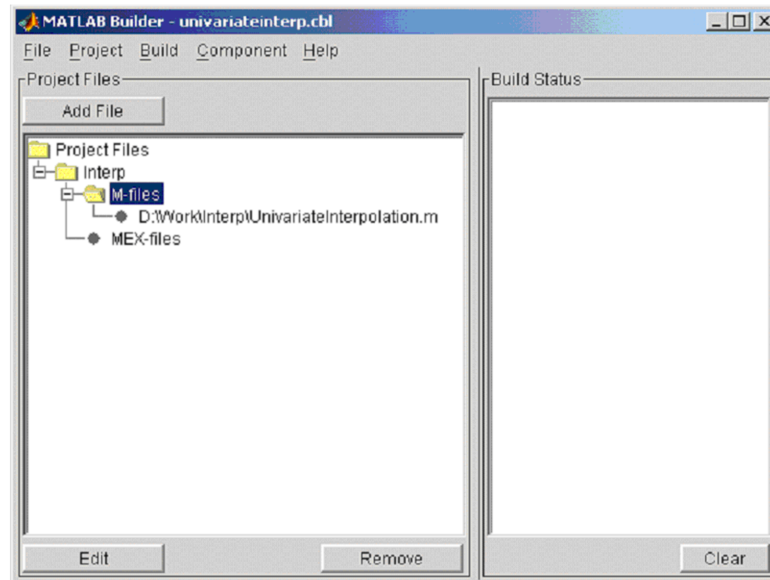
Compiler options: (*leave unselected*)

Building the Project

Follow these steps to create the component.

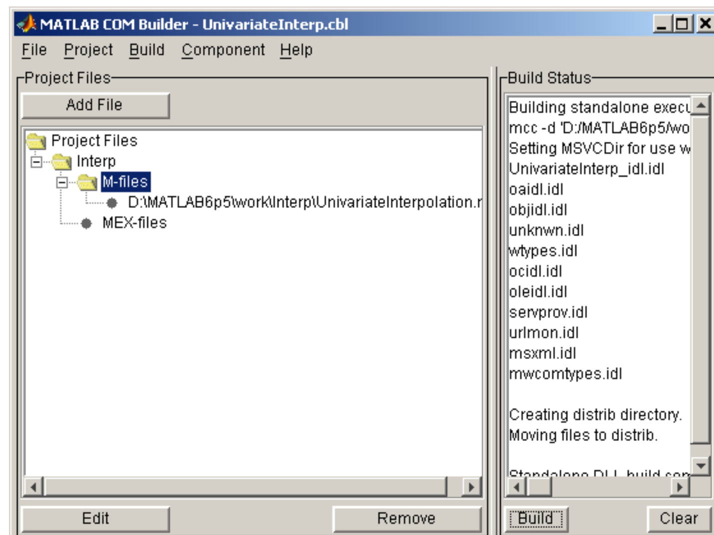
1 In the MATLAB Builder window, click **Add File**.

2 Add `UnivariateInterpolation.m` from the *`matlab`*`\`*`work`*`/Interp` directory
.



3 Click **COM Object > Build** to create the component.

The component is created and placed in the `distrib` directory within the `Class` directory.



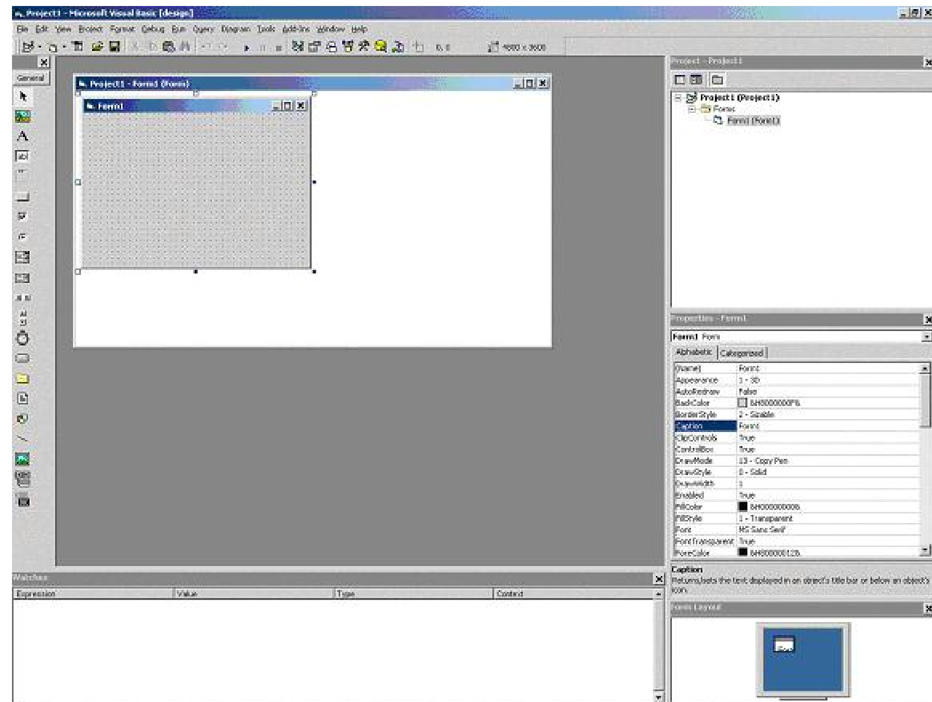
MATLAB Builder for COM GUI After Compilation

Using the Component in Visual Basic

You can call the component from any application that supports COM.

Follow these steps to create a Visual Basic project and add references to the necessary libraries.

1. Start Visual Basic.
2. Create a new Standard EXE project. The design form appears as shown:



Note You might have different components on the left side of the window depending upon the components you have selected for viewing.

---

3 Click **Project > References**.

4 Check the following libraries:

```
UnivariateInterp 1.0 Type Library

MWComUtil 7.1 Type Library
```

Note If you do not see these libraries, you may not have registered the libraries using `mwregsvr`. Refer to "Component Registration" on page 6-4 for information on this process.

Creating the Visual Basic Form

The next step creates a front end or a Visual Basic form for the application. You receive data from the user through this form.

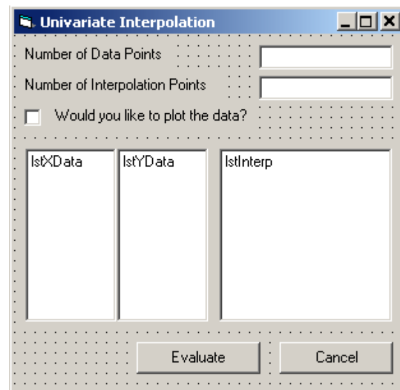
Follow these steps to create a new user form and populate it with the necessary controls.

1. Click **Projects > Component**. Alternatively, press **Ctrl+T**.

   Check that **Microsoft Windows Common Controls 6.0** is selected.

   You will use the `ListView` control from this component library.

2. Add a series of controls to the blank form to create an interface as shown in the next figure.



   The following table summarizes the list of components added and the properties modified.

Control Type	Control Name	Properties	Purpose
Form	frmInterp	Caption = Univariate Interpolation	Container for all components.
Label	lblDataCount	Caption = Number of Data Points	Labels the text box txtNumDataPts.
TextBox	txtNumDataPts	Text =	Number of original data points.
Label	lblInterp	Caption = Number of Interpolation Points	Labels the text box txtInterp.
TextBox	txtInterp	Text =	Number of points over which to interpolate.
Label	lblPlot	Caption = Would you like to plot the data?	Labels the check box chkPlot.
CheckBox	chkPlot		When selected, a message is sent to the COM component to plot the data.
ListView	lstXData	Name = lstXData GridLines = True LabelEdit = lvwAutomatic View = lvwReport	X-data values. Set the view type to lvwReport to allow the user to add data to the list view.
ListView	lstYData	Name = lstYData GridLines = True LabelEdit = lvwAutomatic View = lvwReport	Y-data values. Set the view type to lvwReport to allow the user to add data to the list view.
ListView	lstInterp	Name = lstInterp GridLines = True LabelEdit = lvwAutomatic View = lvwReport	Interpolation points.

Control Type	Control Name	Properties	Purpose
`CommandButton`	`cmdEvaluate`	Caption = `Evaluate` Default = `True`	Executes the function.
`CommandButton`	`cmdCancel`	Caption = `Cancel` Cancel = `True`	Dismisses dialog box without executing function.

3 When the design is complete, save the project by clicking **File > Save**.

4 When prompted for the project name, type `Interp.vbp`, and for the form, type `frmInterp.frm`.

5 To write the underlying code, right-click **frmInterp** in the Project window and click **View Code**.

The following code listing shows the code to implement. Note that this code references the control and variable names listed above. If you have given a different name to any of the controls or any global variable, change this code to reflect the differences.

```
Private theInterp As UnivariateInterp.Interp 'Variable to hold the COM object

Private Sub cmdCancel_Click()
    ' Unload the form if the user hits the cancel button.
    Unload Me
End Sub

Private Sub Form_Initialize()
    On Error GoTo Handle_Error
    ' Create the COM object
    ' If there is an error, handle it accordingly.
    Set theInterp = New UnivariateInterp.Interp
    ' Set the flags such that the input is always passed as double data.
    theInterp.MWFlags.DataConversionFlags.CoerceNumericToType = mwTypeDouble
    Exit Sub
Handle_Error:
    ' Error handling code
    MsgBox ("Error " & Err.Description)
```

```
End Sub

Private Sub Form_Load()
    ' Set the run time properties of the components
    Dim Len1 As Long ' Variable to hold length parameter of the list box
    Dim Len2 As Long ' Variable to hold the length parameter of the list box
    Len2 = lstInterp.Width / 2
    Len1 = (lstInterp.Width - Len2) - 150
    ' Add the column headers to the list boxes
    Call lstXData.ColumnHeaders.Add(, , "XData", Len2)
    Call lstYData.ColumnHeaders.Add(, , "YData", Len2)
    Call lstInterp.ColumnHeaders.Add(, , "Interp Data", Len1)
    Call lstInterp.ColumnHeaders.Add(, , "Interp YData", Len2)

    ' Enable the grid lines
    lstXData.GridLines = True
    lstYData.GridLines = True
    lstInterp.GridLines = True
    lstInterp.FullRowSelect = True

    ' Set the Tab indices for each of the components
    txtNumDataPts.TabIndex = 1
    txtInterp.TabIndex = 2
    lstXData.TabIndex = 3
    lstYData.TabIndex = 4
    lstInterp.TabIndex = 5
    cmdEvaluate.TabIndex = 6
    cmdCancel.TabIndex = 7
End Sub

Private Sub txtInterp_Change()
    ' If user changes number of interpolation points, set the interpolation
    ' point listbox to accomodate the new number of points.
    Dim loopCount As Integer ' loop count
    Dim numData As Integer
    On Error GoTo Handle_Error
    ' First clear the listbox
    Call lstInterp.ListItems.Clear
    ' Create space for the requested number of interpolation points
    If Not (txtInterp.Text = "") Then
```

```
        numData = CDbl(txtInterp.Text)
        For loopCount = 1 To numData
            Call lstInterp.ListItems.Add(loopCount, , "")
        Next
    End If
    Exit Sub
Handle_Error:
    ' Reset the list to 0 elements and also the text box to an empty string.
    MsgBox ("Invalid value for number of Data points")
    lstInterp.ListItems.Clear
    txtInterp.Text = ""
End Sub

Private Sub txtNumDataPts_Change()
    ' If the user changes the number of data points, set the XData and YData
    ' listboxes to accomodate the new number of points.
    Dim loopCount As Integer ' loop count
    Dim numData As Integer
    On Error GoTo Handle_Error
    ' First clear both the listbox (XData and YData)
    Call lstXData.ListItems.Clear
    Call lstYData.ListItems.Clear
    ' Create space for the requested number of data points (XData and YData).
    If Not (txtNumDataPts.Text = "") Then
        numData = CDbl(txtNumDataPts.Text)
        For loopCount = 1 To numData
            Call lstXData.ListItems.Add(loopCount, , "")
            Call lstYData.ListItems.Add(loopCount, , "")
        Next
    End If
    Exit Sub
Handle_Error:
    ' Reset the list to 0 elements and also the text box to an empty string.
    MsgBox ("Error: " & Err.des)
    Call lstXData.ListItems.Clear
    Call lstYData.ListItems.Clear
    txtNumDataPts.Text = ""
End Sub

Private Sub cmdEvaluate_Click()
```

```
' Dim R As Range
Dim XDataInterp As Variant ' Result variable object
Dim loopCount As Integer ' A variable used for loop count
Dim item As ListItem ' Temporary variable to store data in list box
Dim XData() As Double ' X value of data points, passed to COM object
Dim YData() As Double ' Y value of data points, passed to the COM object
Dim XInterp() As Double ' X value of interpolation points, passed to COM
                        ' object
Dim Yi As Variant ' Y value of interpolation points, obtained from COM
                  ' object as ouput value

' Set dimensions of the input and ouput data based on user inputs (number
' of data points and number of interpolation points).
ReDim XData(1 To lstXData.ListItems.Count)
ReDim YData(1 To lstYData.ListItems.Count)
ReDim XInterp(1 To lstInterp.ListItems.Count)
ReDim Yi(1 To lstInterp.ListItems.Count)

' Collect the Data and set the XData, YData, XInterp matrices accordingly
For loopCount = 1 To lstXData.ListItems.Count
    XData(loopCount) = CDbl(lstXData.ListItems.item(loopCount))
    YData(loopCount) = CDbl(lstYData.ListItems.item(loopCount))
Next
For loopCount = 1 To lstInterp.ListItems.Count
    XInterp(loopCount) = CDbl(lstInterp.ListItems.item(loopCount))
    Yi(loopCount) = -1
Next

' Check if the object was created properly.
' If not, go to the error handling routine.

If theInterp Is Nothing Then GoTo Exit_Form

' If there is an error, continue with the code.
On Error GoTo Handle_Error

'Compute Curve Fitting Data
Call theInterp.UnivariateInterpolation(1,Yi,XData,YData,XInterp,_
                                       chkPlot.Value)
```

```
    'Call lstInterp.ListItems.Clear
    For loopCount = LBound(Yi, 2) To UBound(Yi, 2)
        Set item = lstInterp.ListItems(loopCount)
        Call item.ListSubItems.Add(, , Format(Yi(1, loopCount), "##.###"))
    Next
    Call lstInterp.Refresh
    GoTo Exit_Form
Handle_Error:
    ' Error handling routine
    MsgBox ("Error: " & Err.Description)
Exit_Form:
End Sub
```

Matrix Calculator

This example shows how to encapsulate MATLAB utilities that perform basic matrix arithmetic. It includes M-code that performs matrix addition, subtraction, multiplication, division and lef t division and a function to evaluate the eigenvalues for a matrix. The example shows how to create the COM component using MATLAB Builder for COM and how to use the COM component in a Visual Basic application independent of MATLAB.

Note This example assumes that you have downloaded the M-code from `http://www.mathworks.com/matlabcentral/` to the *`matlab`*`/`*`work`* directory. To get the download, search the `File Exchange` at `matlabcentral` for `MatrixArith`.

Building the Component

To build the component:

1. From the MATLAB command prompt change directories to *`matlab`*`/`*`work`*`/MatrixArith`.

2. Enter the command `comtool` to open the MATLAB Builder window.

3. Click **File > New Project**.

   The New Project Settings dialog box“Project Settings Window” on page 2-6 opens.

4. Enter the following settings:

   a. In the **Component name** field enter the component name `matrixMath`.

   b. Press the **Tab** key to move to the **Class name** field. This automatically fills in the **Classes** field with the name `matrixMathclass`. You can use this value if you want or change it.

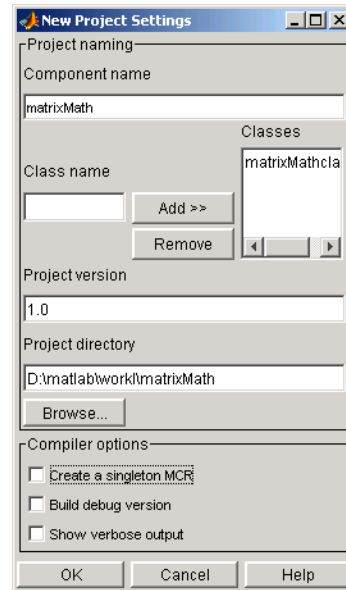
   c. The version has a default of 1.0. Leave this number unchanged.

   d. The **Project directory** field contains a default of a combination of the directory where COM Builder was started, `<matlab>\work` , and the **Component name**, `matrixMath`. You can change this to any directory

that you choose. If the directory you choose does not exist, you will be asked to create it.

e Leave all compiler options unselected.

The New Project Settings dialog box now appears as shown below.



5 Click **OK** to create the `matrixMath` project.

Summary of Project Settings

Component name: `matrixMath`

Class name: `matrixMath`

Project version: `1.0`

Project directory: (*accept default or choose another directory*)

Compiler options: (*leave all unselected*)

Building the Project

In the MATLAB Builder window click **Add File**.

1 Add the following files one at a time:

   - `addMatrices.m`
   - `divideMatrices.m`
   - `eigenValue.m`
   - `leftDivideMatrices.m`
   - `multiplyMatrices.m`
   - `subtractMatrices.m`

   from the directory *`matlab`*`/`*`work`*`/matrixMath`.

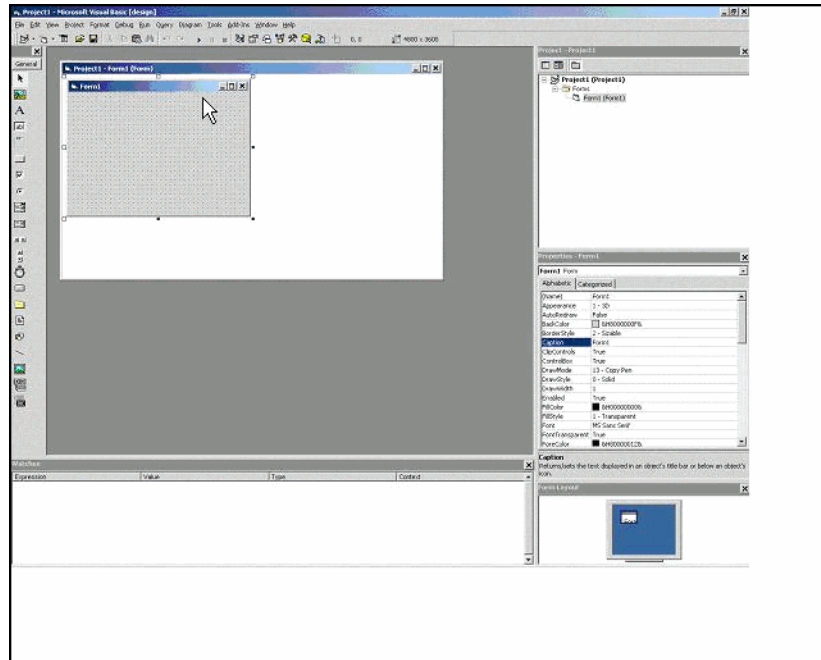
2 Click **COM Object > Build** to create the component.

Using the Component in Visual Basic

You can call the component from any application that supports COM. Follow these steps to create a Visual Basic project and add references to the necessary libraries.

1 Start Visual Basic.

2 Create a new Standard EXE project. This displays the design form shown below.



Note You might have different components on the left side of the window depending upon the components you have selected for viewing.

---

3 Click **Project > References**.

4 Check the following libraries:

```
MatrixMath 1.0 Type Library
```

```
MWComUtil 7.1 Type Library
```

Note If you do not see these libraries, you may not have registered the libraries using `mwregsvr`. Refer to “Component Registration” on page 6-4 for information on this.

Creating the Visual Basic Form

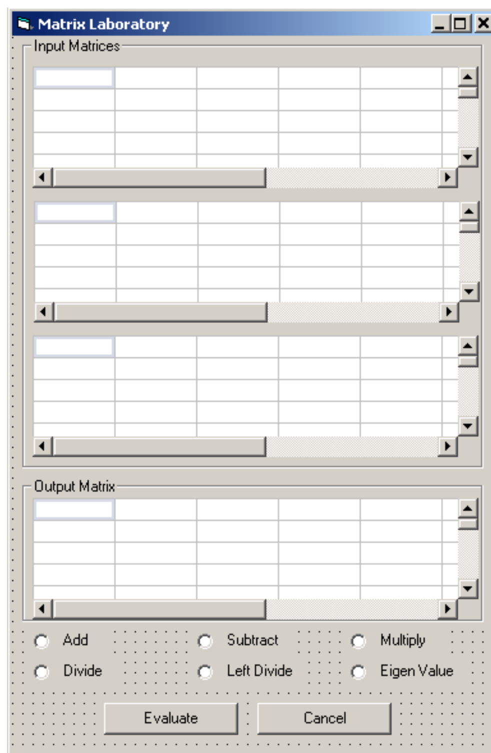
The next step creates a front end or a Visual Basic form for the application. End users enter data in this form.

Follow these steps to create a new user form and populate it with the necessary controls:

1 Click **Projects > Component**. Alternatively, press **Ctrl+T**.

2 Make sure that **Microsoft Windows Common Controls 6.0** are selected. You will use the `Spreadsheet` control from this component library.

3 Add a series of controls to the blank form to create an interface as shown in the next figure.



4 One of the main components used in the Visual Basic form is a Spreadsheet component. For each Spreadsheet component, set properties as follows:

Property	Original Value	New Value
DisplayColumnHeaders	True	False
DisplayHorizontalScrollBar	True	False
DisplayRowHeaders	True	False
DisplayTitleBar	True	False
DisplayToolBar	True	False
DisplayVerticalScrollBar	True	False
MaximumWidth	80%	100%
ViewableRange	1:65536	A1:E5

A consolidated list of components added to the form and the properties modified is as follows:

Control Type	Control Name	Properties	Purpose
Form	frmMatrixMath	Caption = Matrix Laboratory	Container for all components.
Frame	frmInput	Caption = Input Data Points	Groups all input controls .
Frame	frmOutput	Caption = Output Coefficients	Groups all output controls.
Spreadsheet	sheetMat1	Refer to previous table.	Accepts input matrix 1 from user
Spreadsheet	sheetMat2	Refer to previous table.	Accepts input matrix 2 from user.
Spreadsheet	sheetMat3	Refer to previous table.	Accepts input matrix 3 from user.
Spreadsheet	sheetResultMat	Refer to previous table.	Displays result matrix
Label	lblAdd	Caption = Add	Labels **Add** option button.

Control Type	Control Name	Properties	Purpose
OptionButton	optOperation	Index = 0	Option button to perform addition.
Label	lblSub	Caption = Subtract	Labels **Subtract** option button.
OptionButton	optOperation	Index = 1	Option button to perform subtraction.
Label	lblMult	Caption = Multiply	Labels **Multiply** option button.
OptionButton	optOperation	Index = 2	Option button to perform multiplication.
Label	lblDivide	Caption = Divide	Labels **Divide** Option button.
OptionButton	optOperation	Index = 3	Option button to perform division.
Label	lblLeftDivide	Caption = Left Divide	Labels **Left Divide** Option button.
OptionButton	optOperation	Index = 4	Option button to perform left division.
Label	lblEig	Caption = Eigenvalue	Labels **Eigenvalue** Option button.
OptionButton	optOperation	Index = 5	Option button to calculate Eigenvalue of first matrix.
CommandButton	cmdEvaluate	Caption = Evaluate Default = True	Executes function.
CommandButton	cmdCancel	Caption = Cancel Cancel = True	Dismisses dialog box without executing function.

5 When the design is complete, save the project by clicking **File > Save**. When prompted for the project name, type `MatrixMathVB.vbp`, and for the form, type `frmMatrixMath.frm`.

6 To write the underlying code, right-click **frmMatrixMath** in the Project window, and click **View Code**.

The following code listing shows the code to implement. Note that this code references the control and variable names listed above. If you have given a different name to any of the controls or any global variable, change this code to reflect the differences.

```
Dim theMatCal As matrixMath.matrixMath

Private Sub Form_Initialize()
' Create an instance of the COM object and set the MWArray flags.
' If this fails, exit from the form.
On Error GoTo exit_form
    ' Create the object.
    Set theMatCal = New matrixMath.matrixMath
    ' Force the input to be of type double.
    theMatCal.MWFlags.DataConversionFlags.CoerceNumericToType = mwTypeDouble
    ' Set the AutoResizeOutput flag to True, so that you do not have to specify
    ' the size of the output variable as returned by the COM object.
    theMatCal.MWFlags.ArrayFormatFlags.AutoResizeOutput = True
    ' Get the results in a Matrix format.
    theMatCal.MWFlags.ArrayFormatFlags.OutputArrayFormat =_
mwArrayFormatMatrix
    Exit Sub
exit_form:
    ' Error handling routine. Since no object is created, display error '
    'message and unload the form.
    MsgBox ("Error: " & Err.Description)
    Unload Me
End Sub

Private Sub Form_Load()
    ' Set the run time properties for all the components.
    frmInputs.TabIndex = 1
    sheetMat1.AutoFit = True
```

```
    ' Set the tab order for each component and the viewable range.
    ' If you need a larger viewable range, you might want to turn the
    ' horizontal and vertical scroll bars to TRUE.
    sheetMat1.TabStop = True
    sheetMat1.TabIndex = 1
    sheetMat1.Width = 4875
    sheetMat1.ViewableRange = "A1:E5"

    sheetMat2.TabStop = True
    sheetMat2.TabIndex = 2
    sheetMat2.Width = 4875
    sheetMat2.ViewableRange = "A1:E5"

    sheetMat3.TabStop = True
    sheetMat3.TabIndex = 3
    sheetMat3.Width = 4875
    sheetMat3.ViewableRange = "A1:E5"

    sheetResultMatTabStop = False
    sheetResultMatTabIndex = 1
    sheetResultMatWidth = 4875
    sheetResultMat.ViewableRange = "A1:E5"

    frmOutput.TabIndex = 2
    optOperation(0).TabIndex = 3
    optOperation(1).TabIndex = 4
    optOperation(2).TabIndex = 5
    optOperation(3).TabIndex = 6
    optOperation(4).TabIndex = 7
    optOperation(5).TabIndex = 8
End Sub

Private Sub cmdCancel_Click()
    ' When the user clicks on the Cancel button, unload the form.
    Unload Me
End Sub

Private Sub cmdEval_Click()
    ' Declare the variables to be used in the code
```

```
Dim data1 As Range
' This is the temporary variable that holds the value entered in
' the spreadsheet.

'Dim finalRows As Double ' The number of
'Dim finalCols As Double

' Dim tempVal As Double
Dim matArray1 As Variant ' Variable to hold the value of input Matrix 1,
                         ' passed to the COM object directly.
Dim matArray2 As Variant ' Variable to hold the value of input Matrix 1,
                         ' passed via varArg variable.
Dim matArray3 As Variant ' Variable to hold the value of input Matrix 1,
                         ' passed via varArg variable.
Dim varArg(2) As Variant ' Variable to hold the value of input Matrix 1,,
' contains the two optional matrices and is passed to the COM object.

'Dim mat1() As Double
'Dim mat1Dimension2() As Variant

Dim tempRange As Range ' Take the range value as obtained from the
                       ' user input into a temporary range.
Dim resultMat As Variant ' Variable to take the result matrix in
Dim msg As String ' The message thrown by the COM object is taken
                  ' in this variable.

Call sheetResultMat.ActiveSheet.UsedRange.Clear

' Check if the COM object was created properly.
' If not exit
If theMatCal Is Nothing Then GoTo exit_form

' Get the used range of data from the sheetMat1, which will then be
' converted into matArray1.
Set data1 = sheetMat1.ActiveSheet.UsedRange

'finalRows = data1.Rows.Count
'finalCols = data1.Columns.Count

'ReDim mat1(1 To data1.Rows.Count)
```

```
    'ReDim mat1Dimension2(1 To data1.Columns.Count)
    ReDim matArray1(1 To data1.Rows.Count, 1 To data1.Columns.Count) As_
    Double
    For RowCount = 1 To data1.Rows.Count
        For ColCount = 1 To data1.Columns.Count
            ' Extract the values and populate input matrix 1.
            Set tempRange = data1.Cells(RowCount, ColCount)
            'tempVal = tempRange.Value
            'matArray1(RowCount, ColCount) = tempVal
            matArray1(RowCount, ColCount) = tempRange.Value
            'Set mat1(ColCount) = tempRange.Value
        Next ColCount
        ' mat1Dimension2(RowCount) = mat1()
    Next RowCount

    Set data1 = sheetMat2.ActiveSheet.UsedRange
    If (Not (data1.Value = "")) Then
        ReDim matArray2(1 To data1.Rows.Count, 1 To data1.Columns.Count) As_
        Double
        For RowCount = 1 To data1.Rows.Count
            For ColCount = 1 To data1.Columns.Count
                Set tempRange = data1.Cells(RowCount, ColCount)
                tempVal = tempRange.Value
                matArray2(RowCount, ColCount) = tempVal
            Next ColCount
        Next RowCount
        finalCols = data1.Columns.Count
        varArg(0) = matArray2
    End If

    Set data1 = sheetMat3.ActiveSheet.UsedRange
    If (Not (data1.Value = "")) Then
        ReDim matArray3(1 To data1.Rows.Count, 1 To data1.Columns.Count) As_
Double
        For RowCount = 1 To data1.Rows.Count
            For ColCount = 1 To data1.Columns.Count
                Set tempRange = data1.Cells(RowCount, ColCount)
                tempVal = tempRange.Value
                matArray3(RowCount, ColCount) = tempVal
            Next ColCount
```

```
            Next RowCount
            finalCols = data1.Columns.Count
            varArg(1) = matArray3
        End If

        ' Based on the operation selected by the user, call the appropriate method
        ' from the COM object.
        If optOperation.Item(0).Value = True Then ' Add
            Call theMatCal.addMatrices(2, resultMat, msg, matArray1, varArg)
        ElseIf optOperation.Item(1).Value = True Then ' Subtract
            Call theMatCal.subtractMatrices(2, resultMat, msg, matArray1, varArg)
        ElseIf optOperation.Item(2).Value = True Then ' Multiply
            Call theMatCal.multiplyMatrices(2, resultMat, msg, matArray1, varArg)
        ElseIf optOperation.Item(3).Value = True Then ' Divide
            Call theMatCal.divideMatrices(2, resultMat, msg, matArray1, varArg)
        ElseIf optOperation.Item(4).Value = True Then ' Left Divide
            Call theMatCal.leftDivideMatrices(2, resultMat, msg, matArray1,_
            varArg)
        ElseIf optOperation.Item(5).Value = True Then ' Eigen Value
            Call theMatCal.eigenValue(2, resultMat, msg, matArray1)
        End If

        ' If the result matrix is a scalar double, display it in the first cell.
        If (VarType(resultMat) = vbDouble) Then
            Set tempRange = sheetResultMat.Cells(1, 1)
            tempRange.Value = resultMat

        ' If the result matrix is not a scalar double, loop through it to display
        ' all the elements.
        Else
            For RowCount = 1 To UBound(resultMat, 1)
                For ColCount = 1 To UBound(resultMat, 2)
                    Set tempRange = sheetResultMat.Cells(RowCount, ColCount)
                    tempRange.Value = resultMat(RowCount, ColCount)
                Next ColCount
            Next RowCount
        End If
        Exit Sub
    exit_form:
        MsgBox ("Error: " & Err.Description)
```

```
    Unload Me
End Sub

' If the user changes the operation, clear the result matrix.
Private Sub optOperation_Click(Index As Integer)
    Call sheetResultMat.ActiveSheet.Cells.Clear
End Sub
```

Curve Fitting

This example is a demonstration of the optimal fitting of a nonlinear function to a set of data, using the curve-fitting demo `fitfun` provided with MATLAB. It uses `fminsearch`, an implementation of the Nelder-Mead simplex (direct search) algorithm, to minimize a nonlinear function of several variables.

This example shows you how to create the COM component using MATLAB Builder for COM and how to use this COM component in a Visual Basic application independent of MATLAB.

Note This example assumes that you have downloaded the M-code from `http://www.mathworks.com/matlabcentral/` to the *`matlab`*`/`*`work`* directory.

Building the Component

To build the component:

1. From the MATLAB command prompt, change directories to *`matlab`*`/`*`work`*.

2. Enter the command `comtool` to start the COM Builder graphical user interface.

3. Click **File > New Project**.

4. In the New Project Settings dialog, enter the following settings:

   a. In the **Component name** text block enter the component name `CurveFit`.

   b. Press the **Tab** key to move to the **Class name** text block.

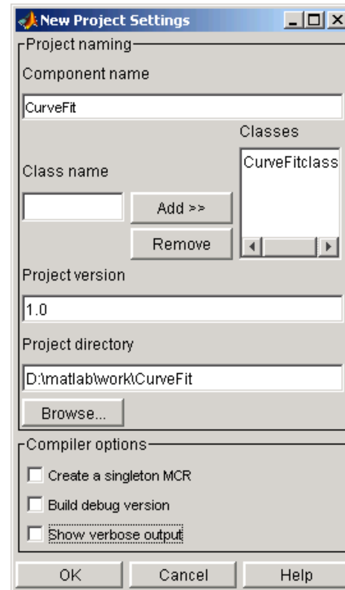
      This automatically fills in the **Class name** field with the name `CurveFitclass`.

   c. The version has a default of 1.0. Leave this number unchanged.

   d. The **Project directory** field contains a default of a combination of the directory where COM Builder was started and the component name, `CurveFit`. You can change this to any directory that you choose. If the directory you choose does not exist, you will be asked to create it.

e Leave all compiler options unselected.

The New Project Settings dialog box now looks as shown:



5 Click **OK** to create the `CurveFit` project.

Summary of Project Settings

Component name: `CurveFit`

Class name: `CurveFit`

Project version: `1.0`

Project directory: (*accept default or choose another directory*)

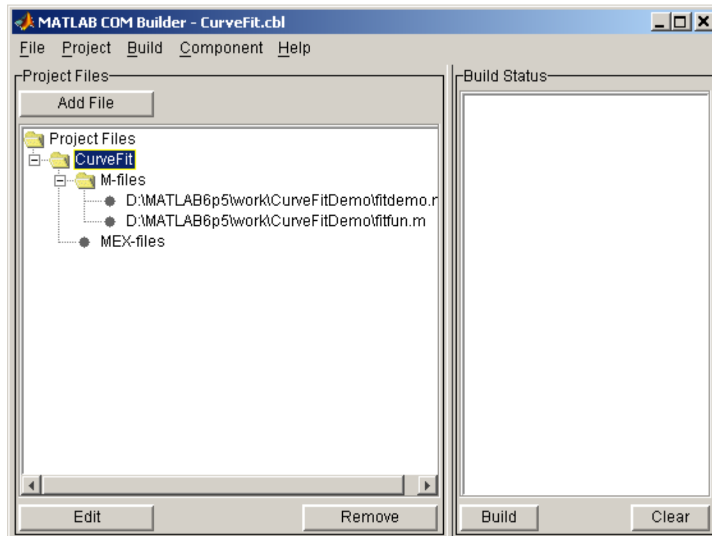
Compiler options: (*leave all unselected*)

Building the Project

To build the project:

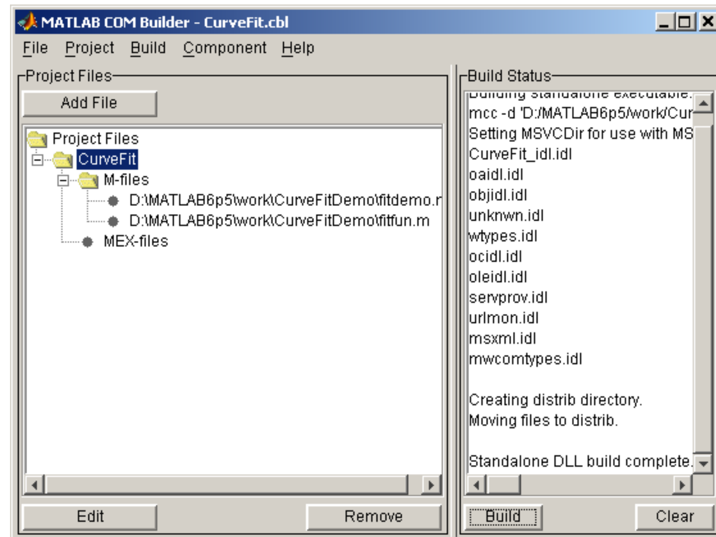
1 In the MATLAB Builder window, click **Add File**.

2 Add `fitfun.m` and `fitdemo.m` from the directory *`matlab`*`/`*`work`*`/CurveFitDemo`.



3 Click **COM Object > Build** to create the component.

The component is created and placed in the `distrib` directory within the `Class` directory, as shown:



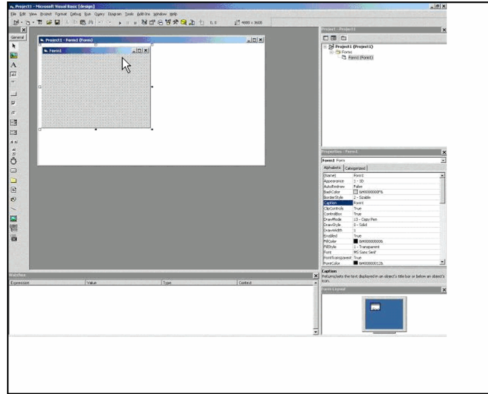
Using the Component in Visual Basic

You can call the component from any application that supports COM.

Follow these steps to create a Visual Basic project and add references to the necessary libraries.

1. Start Visual Basic.

2. Create a new Standard EXE project to display the design form as shown:



Note You might have different components on the left side of the window depending upon the components you have selected for viewing.

---

3 Click **Project > References**.

4 Check the following libraries:

```
CurveFit 1.0 Type Library

MWComUtil 7.1 Type Library
```

Note If you do not see these libraries, you may not have registered the libraries using `mwregsvr`. Refer to "Component Registration" on page 6-4 for information.

Creating the Visual Basic Form

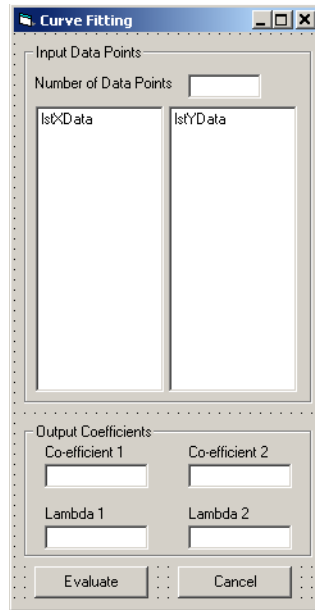
The next step is to create a front end or a Visual Basic form for the application. End users enter data on the form.

Follow these steps to create a new user form and populate it with the necessary controls.

1 Click **Projects > Component**. Alternatively, press **Ctrl+T**.

2 Make sure that **Microsoft Windows Common Controls 6.0** are selected. You will use the `ListView` control from this component library.

3 Add a series of controls to the blank form to create an interface as shown in the next figure.



The following table shows the components and properties that are required:

Control Type	Control Name	Properties	Purpose
`Form`	`frmCurveFit`	Caption = `Curve Fitting`	Container for all components.
`Frame`	`frmInput`	Name = `frmInput*` Caption = `Input Data Points`	Groups all input controls.

Control Type	Control Name	Properties	Purpose
Frame	frmOutput	Name = frmOutput* Caption = Output Coefficients	Groups all output controls.
Label	lblNumDataPoints	Caption = Number of Data Points	Labels the text box that takes the number of data points the user wants to enter.
TextBox	txtNumOfDatPoints	Text =	Holds number of data points the user wants to enter. Sets size of list box added later.
ListView	lstXData	Name = lstXData GridLines = TrueLabel Edit = lvwAutomatic View = lvwReport	X-data values. Set the view type to lvwReport to enable user to add data to the list view.
ListView	lxtYData	Name = lstYData GridLines = TrueLabel Edit = lvwAutomatic View = lvwReport	Y-data values.
Label	lblCoeff1*	Caption = Co-efficient 1	Labels text box for coefficient 1.
Label	lblCoeff2	Caption = Co-efficient 2	Labels text box for coefficient 2.
TextBox	txtCoeff1	Text =	Displays value of coefficient 1 as calculated by the COM module.

Control Type	Control Name	Properties	Purpose
TextBox	txtCoeff2	Text =	Displays value of coefficient 2 as calculated by the COM module.
Label	lblLambda1*	Caption = Lambda 1	Labels text box for lambda 1.
Label	lblLambda2	Caption = Lambda 2	Labels text box for lambda 2.
TextBox	txtLambda1	Text =	Displays value of lambda 1 as calculated by the COM module.
TextBox	txtLambda2	Text =	Displays value of lambda 2 as calculated by the COM module.
CommandButton	cmdEvaluate	Caption = Evaluate Default = True	Executes function.
CommandButton	cmdCancel	Caption = Cancel Cancel = True	Dismisses dialog box without executing function.

4 When the design is complete, save the project by clicking **File > Save**.

When prompted for the project name, type `CurveFitExample.vbp`, and for the form, type `frmCurveFit.frm`.

5 In the Project window, right-click `frmCurveFit` and click **View Code**.

The following code listing shows the code to implement. Note that this code references the control and variable names listed above. If you have

given a different name to any of the controls or any global variable, change this code to reflect the differences.

```
Dim theFit As CurveFit.CurveFit ' Variable to hold the COM Object

' This routine is exectued when the form is initialized.
Private Sub Form_Initialize()
' If the initialize routine fails, handle it accordingly.
On Error GoTo Exit_Form
    ' Create the COM object
    Set theFit = New CurveFit.CurveFit
    ' Set the flags such that the output is transposed.
    theFit.MWFlags.ArrayFormatFlags.TransposeOutput = True
    Exit Sub
Exit_Form:
    ' Display the error message and Unload the form if object
creation failed
    MsgBox ("Error: " & Err.Description)
    MsgBox ("Error: Could not create the COM object")
    Unload Me
End Sub

Private Sub Form_Load()
On Error GoTo Exit_Form
    ' Set the runtime properties of the components

    ' Set the headers of the column
    Call lstXData.ColumnHeaders.Add(, , "X Data")
    Call lstYData.ColumnHeaders.Add(, , "Y Data")

    ' Make labeledit property automatic so that you edit the label.
    lstXData.LabelEdit = lvwAutomatic
    lstYData.LabelEdit = lvwAutomatic

    ' Make the grid lines for the listbox visible.
    lstXData.GridLines = True
    lstYData.GridLines = True
    Exit Sub
Exit_Form:
    ' Error handling routine. Since cannot load the form,
```

```
    ' display the error message and unload the program.
    MsgBox ("Error: Could not load the form")
    MsgBox ("Error: " & Err.Description)
    Unload Me
End Sub

Private Sub cmdCancel_Click()
    ' If the user hits the cancel button, unload the form.
    Unload Me
End Sub

Private Sub txtNumOfDataPoints_Change()
    ' If user changes number of data points, clear XData and YData
    ' listboxes. Provide enough spaces for given number of points.
    Dim loopCount As Integer
    Call lstXData.ListItems.Clear
    Call lstYData.ListItems.Clear
    If (txtNumOfDataPoints.Text = "") Then
        Exit Sub
    End If
    For loopCount = 1 To CInt(txtNumOfDataPoints.Text)
        lstXData.ListItems.Add (loopCount)
        lstYData.ListItems.Add (loopCount)
    Next loopCount
End Sub

Private Sub cmdEvaluate_Click()
    Dim loopCount As Integer ' loop counter
    Dim numOfData As Integer ' variable to hold the number of data
                             ' points the user has entered
    Dim XData() As Double ' Column Vector for XData, will be passed
                          ' as input to the COM method.
    Dim YData() As Double ' Column Vector for YData, will be passed
                          ' as input to the COM method.
    Dim Coeff As Variant ' Coefficient values will be returned by
                         ' the COM method in this variable.
    Dim Lambda As Variant ' Lambda values will be returned by the
                          ' COM method in this variable.

    ' If there is an error, handle it accordingly.
```

```
On Error GoTo Handle_Error
    If txtNumOfDataPoints.Text = "" Then
        Exit Sub
    End If
    ' Get the number of data points.
    numOfData = CInt(txtNumOfDataPoints.Text)
    ReDim XData(1 To numOfData) As Double
    ReDim YData(1 To numOfData) As Double
    ' Read the input data into respective double arrays.
    For loopCount = 1 To numOfData
        XData(loopCount) = lstXData.ListItems.Item(loopCount)
        YData(loopCount) = lstYData.ListItems.Item(loopCount)
    Next loopCount

    ' Call the COM method
    Call theFit.fitdemo(2, Coeff, Lambda, XData, YData)

    ' Display values of coefficients returned by the COM method.
    txtCoeff1.Text = CStr(Format(Coeff(1, 1), "##.####"))
    txtCoeff2.Text = CStr(Format(Coeff(1, 2), "##.####"))
    txtLambda1.Text = CStr(Format(Lambda(1, 1), "##.####"))
    txtLambda2.Text = CStr(Format(Lambda(1, 2), "##.####"))
    Exit Sub
Handle_Error:
    ' Error handling routine
    MsgBox ("Error: " & Err.Description)
End Sub
```

Bouncing Ball Simulation

This example is adapted from the `ballode` demo provided with MATLAB. It demonstrates repeated event location, where the conditions are changed after each terminal event.

This demo computes 10 bounces with calls to `ode23`. A user-specified damping factor after each bounce attenuates the speed of the ball. The trajectory is plotted using the output function `odeplot`. In addition to the damping factor, the user can also provide the initial velocity, the maximum number of bounce to track, and the maximum time until demo is completed.

This example shows you how to create the COM component using MATLAB Builder for COM and how to use this COM component in a Visual Basic application independent of MATLAB.

Note This example assumes that you have downloaded the M-code to the *`matlab`*`/`*`work`* directory.

Building the Component

To build the component:

1. From the MATLAB command prompt change directories to *`matlab`*`/`*`work`*`/BallODE`.
2. Enter the command `comtool` to start the COM Builder graphical user interface.
3. Click **File > New Project**.
4. In the New Project Settings window, enter the following settings:
   a. In the **Component name** text block enter the component name `bouncingBall`.
   b. Press **Tab** to move to the **Class name** text block.

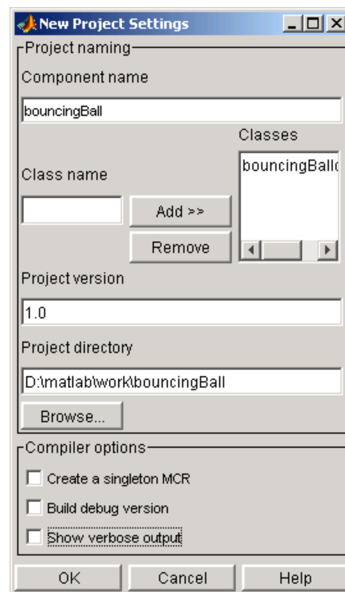
      This automatically fills in the **Class name** field with the name `bouncingBallclass`.

c The version has a default of 1.0. Leave this number unchanged.

d The **Project directory** field contains a default of a combination of the directory where COM Builder was started and the component name, `bouncingBall`. You can change this to any directory that you choose. If the directory you choose does not exist, you will be asked to create it.

e Leave all compiler options unselected.

The New Project Settings window now looks as shown:



5 Click **OK** to create the `bouncingBall` project.

Summary of Project Settings

Component name: `bouncingBall`

Class name: `bouncingBall`

Project version: `1.0`

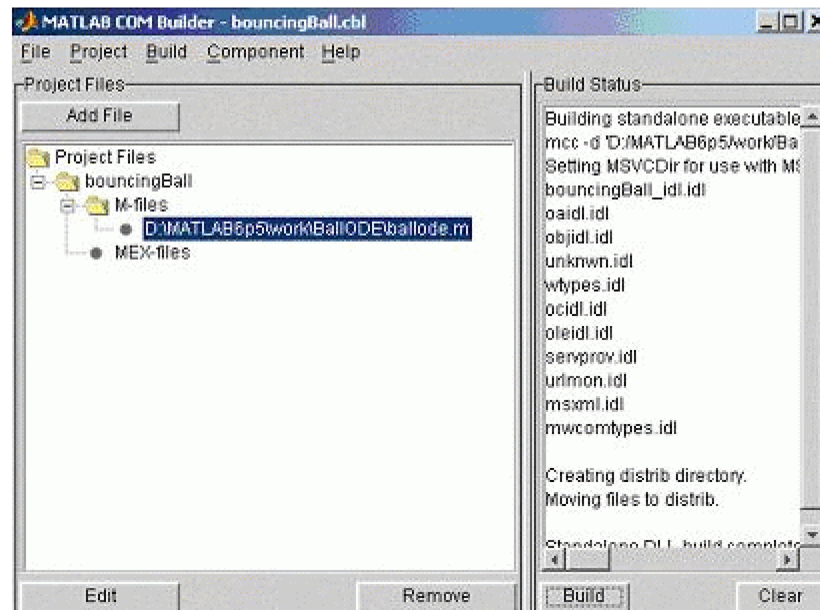
Project directory: (*accept default or choose another directory*)

Compiler options: (*leave all unselected*)

Building the Project

1 From the Project window, click **Add File.**

Add `ballode.m` from the directory *`matlab`*`/`*`work`*`/BallODE` as shown:



2 Click **Build > COM Object**.

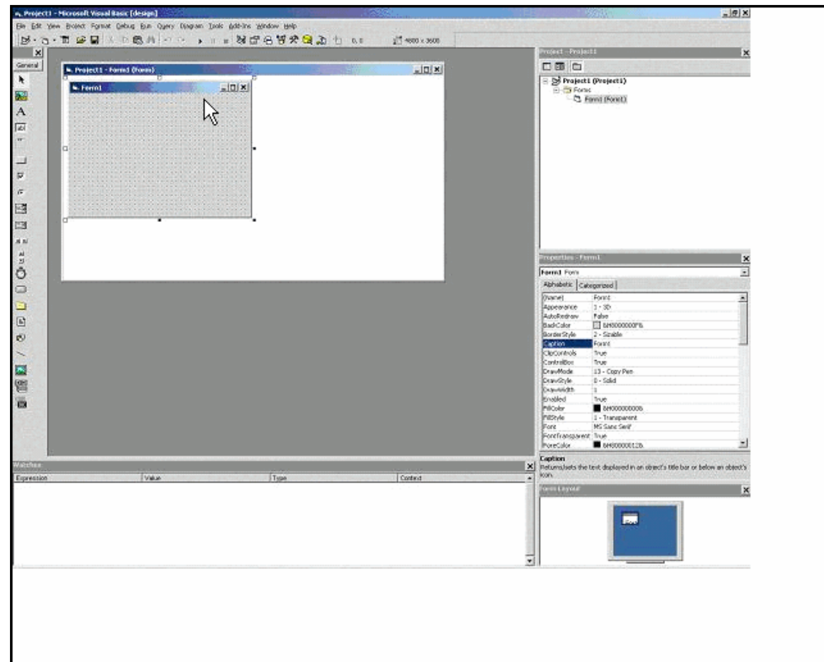
Using the Component in Visual Basic

You can call the component from any application that supports COM.

Follow these steps to create a Visual Basic project and add references to the necessary libraries.

1 Start Visual Basic.

2 Create a new Standard EXE project, which displays the design form as shown:



Note You might have different components on the left side of the window depending upon the components you have selected for viewing.

---

3 Click **Project > References**.

4 Select the following libraries:

- `bouncingBall 1.0 Type Library`.(If you named your class something other than `bouncingBall` and/or gave a different version number, click and use the appropriate component and corresponding type library.)
- `MWComUtil 7.1 Type Library`

Note If you do not see these libraries, you may not have registered the libraries using `mwregsvr`. Refer to "Component Registration" on page 6-4 for information on this.

Creating the Visual Basic Form

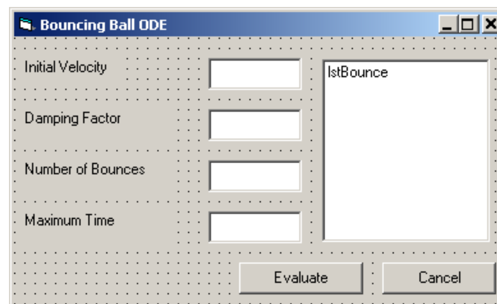
The next step is to create a front end or a Visual Basic form for the application. End users enter data with this form.

Follow these steps to create a new user form and populate it with the necessary controls.

1 Click **Projects > Component**. Alternatively, press **Ctrl+T**.

2 Check that **Microsoft Windows Common Controls 6.0** is selected. You will use the `ListView` control from this component library.

3 Add a series of controls to the blank form to create an interface as shown:



The following table lists the components to be added and the properties to be modified.

Control Type	Control Name	Properties	Purpose
Form	frmBallOde	Caption = Bouncing Ball ODE	Container for all components.
Frame	frmInput	Name = frmInput* Caption = Input Data Points	Groups all input controls.
Frame	frmOutput	Name = frmOutput* Caption = Output Coefficients	Groups all output controls.
Label	lblInitVal	Caption = Initial Velocity	Labels the text box txtInitVal.
TextBox	txtInitVal	Text =	Holds initial velocity by which ball is thrown into the air.
Label	lblDamp	Caption = Damping Factor	Labels the text box txtDamp.
TextBox	txtDamp	Text =	Holds damping factor for the bounce, that is, the factor by which the speed of the ball is reduced after it bounces.
Label	lblIter	Caption = Number of Bounces	Labels the text box txtIter.
TextBox	txtIter	Text =	Holds number of iterations or bounces to track.
Label	lblFinalTime	Caption = Maximum Time	Labels the text box txtFinalTime.
TextBox	txtFinalTime	Text =	Stores time until demo is completed.

Control Type	Control Name	Properties	Purpose
`ListView`	`lstBounce`	Name = `lstBounce` GridLines = `True` LabelEdit = `lvwManual` View = `lvwReport`	Displays time stamp when ball bounces off the ground.
`CommandButton`	`cmdEvaluate`	Caption = `Evaluate` Default = `True`	Executes function.
`CommandButton`	`cmdCancel`	Caption = `Cancel` Cancel = `True`	Dismisses dialog box without executing function.

4 When the design is complete, save the project by clicking **File > Save**. When prompted for the project name, type `BallOde.vbp`, and for the form, type `frmBallOde.frm`.

5 In the Project window right-click `frmBallOde` and click **View Code**.

The following code listing shows the code to implement. Note that this code references the control and variable names listed above. If you have given a different name to any of the controls or any global variable, change this code to reflect the differences.

```
Private theBall As Variant ' Variable to hold the COM object.

Private Sub cmdCancel_Click()
    ' If the user presses the Cancel button, unload the form.
    Unload Me
End Sub

Private Sub Form_Initialize()
    Dim Len1 As Long ' Used to set length of columns for list box.
    Dim Len2 As Long ' Used to set length of columns for list box.
    On Error GoTo Handle_Error
    ' Set length of the each column based on length of the listbox
    ' such that the two columns span the maximum area without
    ' creating a horizontal scroll bar.
    Len2 = lstBounce.Width / 2
```

```
    Len1 = (lstBounce.Width - Len2) - 300

    ' Add column headers to each column in the list box.
    Call lstBounce.ColumnHeaders.Add(, , "Bounce", Len1)
    Call lstBounce.ColumnHeaders.Add(, , "Time", Len2)

    ' Set tab indices for each component.
    txtInitVel.TabIndex = 1
    txtDamp.TabIndex = 2
    txtIter.TabIndex = 3
    txtFinalTime.TabIndex = 4
    cmdEvaluate.TabIndex = 5
    cmdCancel.TabIndex = 6
    lstBounce.TabStop = False

    ' Create the COM object
    ' If there is an error, handle it accordingly.
    Set theBall = CreateObject("bouncingBall.bouncingBall.1_0")
    Exit Sub
Handle_Error:
    ' Error handling code
    MsgBox ("Error " & Err.Description)
End Sub
Private Sub cmdEvaluate_Click()
    ' Dim R As Range
    Dim zeroTime As Variant ' Result variable object.
    Dim loopCount As Integer
    Dim item As ListItem

    ' Check if the object was created properly.
    ' If not, go to the error handling routine.
    If theBall Is Nothing Then GoTo Exit_Form

    ' If there is an error, continue with the code.
    On Error Resume Next

    ' Process inputs
    ' If the user does not provide the values for input parameters,
    ' use the default values.
    If txtDamp.Text = Empty Then
```

```
        txtDamp.Text = 0.9
    End If
    If txtInitVel.Text = Empty Then
        txtInitVel.Text = 20
    End If
    If txtIter.Text = Empty Then
        txtIter.Text = 15
    End If
    If txtFinalTime.Text = Empty Then
        txtFinalTime.Text = 20
    End If

    'Compute Bouncing ball data
    Call theBall.ballode(1, zeroTime, CDbl(txtIter.Text),_
    CDbl(txtDamp.Text), CDbl(txtFinalTime.Text),_
    CDbl(txtInitVel.Text))

    ' Display the output values (time stamp when ball bounces on
    ' the ground).
    Call lstBounce.ListItems.Clear
    For loopCount = LBound(zeroTime, 1) To UBound(zeroTime, 1)
        Set item = lstBounce.ListItems.Add(, , Format(loopCount))
        Call item.ListSubItems.Add(, , Format(zeroTime(loopCount,_
        1), "##.###"))
    Next
    Call lstBounce.Refresh

    GoTo Exit_Form
Handle_Error:
    ' Error handling routine
    MsgBox (Err.Description)
Exit_Form:
End Sub
```

Troubleshooting

The following table shows diagnostic messages you might encounter, probable causes for the message, and suggested solutions.

Note MATLAB Builder for COM uses the MATLAB Compiler to generate components. This means that you might see diagnostic messages from the MATLAB Compiler. See the "Troubleshooting" section of the MATLAB Compiler documentation for more information about those messages.

MATLAB Builder for COM Diagnostic Messages and Suggested Solutions

Message	Probable Cause	Suggested Solution
`MBUILD.BAT: Error: The chosen compiler does not support building COM objects.`	The chosen compiler does not support building COM objects.	Rerun `mbuild -setup` and choose a supported compiler.
`Error in` *`component_name`*`.`*`class_name`*`.x: Error getting data conversion flags.`	Usually caused by `mwcomutil.dll` not being registered.	**1** Open a DOS window. **2** Change directories to *`matlabroot`*`\bin\win32`. **3** Run the following command: `mwregsvr mwcomutil.dll` (*`matlabroot`* is your root MATLAB directory.)
`Error in VBAProject: ActiveX component can't create object.`	1. Project DLL is not registered. 2. An incompatible MATLAB DLL exists somewhere on the system path.	If the DLL is not registered, **1** Open a DOS window. **2** Change directories to *`projectdir`*`\distrib`. **3** Run the following command: `mwregsvr` *`projectdll`*`.dll` (*`projectdir`* represents the location of your project files).

MATLAB Builder for COM Diagnostic Messages and Suggested Solutions (Continued)

Message	Probable Cause	Suggested Solution
`Error in VBAProject: Automation error The specified module could not be found.`	This usually occurs if MATLAB is not on the system path.	See Required Locations to Develop and Use Components on page 5-4
`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 usually occurs if MATLAB is not on the system path.	See Required Locations to Develop and Use Components on page 5-4
	You might also get this error if you try to deploy your component without adding the path for the DLL to the system path on the target machine.	On the target machine where the COM component is to be used: **1** Use the `extractCTF.exe` utility to decompress the `.ctf` file generated by COM Builder when you built the COM component. **2** Look at the files in the CTF, and note the path for the DLL. **3** Add this path to the system path. See the MATLAB Compiler documentation for more information about `extractctf.exe`.

If your application generates a diagnostic message indicating that a module cannot be found, it could be that the MCR is not located properly on your path or that the CTF file is not in the proper directory. How to fix this problem depends on whether it occurs on a development machine (where you are using COM Builder to create a component) or the target machine (where you are trying to use a component in your application).

Required Locations to Develop and Use Components

	Development Machine	Target Machine
MCR	Make sure that *matlabroot*\bin\win32 appears on your system path ahead of any other MATLAB installations. (*matlabroot* is your root MATLAB directory.)	Verify that *mcr_root**ver*\runtime\win32 appears on your system path. (*mcr_root* is your root MCR directory.)
CTF	Verify that the CTF file is in the same directory as your program's executable file.	

6

How MATLAB Builder for COM Works Internally

Overview of Internal Processes (p. 6-2)	Descibes the steps in the build process
Component Registration (p. 6-4)	Describes the registration process for MATLAB Builder for COM components.
Data Conversion Rules (p. 6-8)	Converting between MATLAB and COM variants.
Calling Conventions (p. 6-22)	Describes the calling conventions for MATLAB Builder for COM components.

Overview of Internal Processes

The process of creating a MATLAB Builder for COM component is completely automatic from a user point of view. You specify a list of M-files to process and a few additional pieces of information, such as the component name, the class names, and the version number.

The internal build process involves the following steps:

1 "Code Generation" on page 6-2

2 "Create Interface Definitions" on page 6-2

3 "C++ Compilation" on page 6-3

4 "Linking and Resource Binding" on page 6-3

5 "Component Registration" on page 6-3

Code Generation

The first step in the build process generates all source code and other supporting files needed to create the component. It also creates the main source file (`mycomponent_dll.cpp`) containing the implementation of each exported function of the DLL. The compiler additionally produces an Interface Description Language (IDL) file (`mycomponent_idl.idl`), containing the specifications for the component's type library, interface, and class, with associated GUIDs. (GUID is an acronym for Globally Unique Identifier, a 128-bit integer guaranteed always to be unique.)

Created next are the C++ class definition and implementation files (`myclass_com.hpp` and `myclass_com.cpp`). In addition to these source files, the compiler generates a DLL exports file (`mycomponent.def`), a resource script (`mycomponent.rc`), and a Component Technology File (`mycomponent.ctf`). See the MATLAB Compiler documentation for a discussion of `ctf` files.

Create Interface Definitions

The second step of the build process invokes the IDL compiler on the IDL file generated in step 1 (`mycomponent_idl.idl`), creating the interface header

file (`mycomponent_idl.h`), the interface GUID file (`mycomponent_idl_i.c`), and the component type library file (`mycomponent_idl.tlb`). The interface header file contains type definitions and function declarations based on the interface definition in the IDL file. The interface GUID file contains the definitions of the GUIDs from all interfaces in the IDL file. The component type library file contains a binary representation of all types and objects exposed by the component.

C++ Compilation

The third step compiles all C/C++ source files generated in steps 1 and 2 into object code. One additional file containing a set of C++ template classes (`mclcomclass.h`) is included at this point. This file contains template implementations of all necessary COM base classes, as well as error handling and registration code.

Linking and Resource Binding

The fourth step produces the finished DLL for the component. This step invokes the linker on the object files generated in step 3 and the necessary MATLAB libraries to produce a DLL component (`mycomponent_1_0.dll`). The resource compiler is then invoked on the DLL, along with the resource script generated in step 1, to bind the type library file generated in step 2 into the completed DLL.

Component Registration

The final build step registers the DLL on the system, as described in the next section.

Component Registration

When MATLAB Builder for COM creates a component, it automatically generates a binary file called a *type library*. As a final step of the build, this file is bound with the resulting DLL as a resource.

Self-Registering Components

COM Builder components are all *self-registering*. A self-registering component contains all the necessary code to add or remove a full description of itself to or from the system registry. The `mwregsvr` utility, distributed with the MCR, registers self-registering DLLs. For example, to register a component called `mycomponent_1_0.dll`, issue this command at the DOS command prompt.

```
mwregsvr mycomponent_1_0.dll
```

When `mwregsvr` completes the registration process, it displays a message indicating success or failure. Similarly, the command

```
mwregsvr /u mycomponent_1_0.dll
```

unregisters the component.

A COM Builder component installed onto a particular machine must be registered with `mwregsvr`. If you move a component into a different directory on the same machine, you must repeat the registration process. When deleting a component from a specific machine, first unregister it to ensure that the registry does not retain erroneous information.

Note The `mwregsvr` utility invokes a process that is similar to `regsvr32.exe`, except that `mwregsvr` does not require interaction with a user at the console. The `regsvr32.exe` process belongs to the Windows OS and is used to register dynamic-link libraries and ActiveX controls in the registry. This program is important for the stable and secure running of your computer and should not be terminated. You can use `regsvr32.exe` as an alternative to `mwregsvr` to register your library.

Globally Unique Identifier (GUID)

Information is stored in the registry as keys with one or more associated named values. The keys themselves have values of primarily two types: readable strings and GUIDs. (GUID is an acronym for Globally Unique Identifier, a 128-bit integer guaranteed always to be unique.)

COM Builder automatically generates GUIDs for COM classes, interfaces, and type libraries that are defined within a component at build time, and codes these keys into the component's self-registration code.

The interface to the system registry is directory based. COM-related information is stored under a top-level key called `HKEY_CLASSES_ROOT`. Under `HKEY_CLASSES_ROOT` are several other keys under which COM Builder writes component information.

See the following table for a list of the keys and their definitions.

Key	Definition
`HKEY_CLASSES_ROOT\CLSID`	Information about COM classes on the system. Each component creates a new key under `HKEY_CLASSES_ROOT\CLSID` for each of its COM classes. The key created has a value of the GUID that has been assigned the class and contains several subkeys with information about the class.
`HKEY_CLASSES_ROOT\Interface`	Information about COM interfaces on the system. Each component creates a new key under `HKEY_CLASSES_ROOT\Interface` for each interface it defines. This key has the value of the GUID assigned to the interface and contains subkeys with information about the interface.

Key	Definition
HKEY_CLASSES_ROOT\TypeLib	Information about type libraries on the system. Each component creates a key for its type library with the value of the GUID assigned to it. Under this key a new key is created for each version of the type library. Therefore, new versions of type libraries with the same name reuse the original GUID but create a new subkey for the new version.
HKEY_CLASSES_ROOT\<ProgID>, HKEY_CLASSES_ROOT\<VerIndProgID>	These two keys are created for the component's Program ID and Version Independent Program ID. These keys are constructed from strings of the following forms: *component-name.class-name* *component-name.class-name version-number*. These keys are useful for creating a class instance from the component and class names instead of the GUIDs.

Versioning

MATLAB Builder for COM components support a simple versioning mechanism designed to make building and deploying multiple versions of the same component easy to implement. The version number of a component appears as part of the DLL name, as well as part of the version-dependent ID in the system registry.

When a component is created, you can specify a version number. (The default is 1.0). During the development of a specific version of a component, the version number should be kept constant. When this is done, the MATLAB Compiler, in certain cases, reuses type library, class, and interface GUIDs for each subsequent build of the component. This avoids the creation of an

excessive number of registry keys for the same component during multiple builds, as occurs if new GUIDs are generated for each build.

When a new version number is introduced, the MATLAB Compiler generates new class and interface GUIDs so that the system recognizes them as distinct from previous versions, even if the class name is the same. Therefore, once you deploy a built component, use a new version number for any changes made to the component. This ensures that after you deploy the new component, it is easy to manage the two versions.

The MATLAB Compiler implements the versioning rules for a specific component name, class name, and version number by querying the system registry for an existing component with the same name:

- If an existing component has the same version, it uses the GUID of the existing component's type library. If the name of the new class matches the previous version, it reuses the class and interface GUIDs. If the class names do not match, it generates new GUIDs for the new class and interface.
- If it finds an existing component with a different version, it uses the existing type library GUID and creates a new subkey for the new version number. It generates new GUIDs for the new class and interface.
- If it does not find an existing component of the specified name, it generates new GUIDs for the component's type library, class, and interface.

Data Conversion Rules

This section describes the data conversion rules for MATLAB Builder for COM components. These components are dual interface COM objects that support data types compatible with Automation.

Note *Automation* (formerly called OLE Automation) is a technology that allows software packages to expose their unique features to scripting tools and other applications. Automation uses the Component Object Model (COM), but may be implemented independently from other OLE features, such as in-place activation.

When a method is invoked on a COM Builder component, the input parameters are converted to MATLAB internal array format and passed to the compiled MATLAB function. When the function exits, the output parameters are converted from MATLAB internal array format to COM Automation types.

The COM client passes all input and output arguments in the compiled MATLAB functions as type `VARIANT`. The COM `VARIANT` type is a union of several simple data types. A type `VARIANT` variable can store a variable of any of the simple types, as well as arrays of any of these values.

The Win32 Application Program Interface (API) provides many functions for creating and manipulating `VARIANT`s in C/C++, and Visual Basic provides native language support for this type. See the Visual Studio documentation for definitions and API support for COM `VARIANT`s. `VARIANT` variables are self describing and store their type code as an internal field of the structure.

Note This discussion of data refers to both `VARIANT` and `Variant` data types. `VARIANT` is the C++ name and `Variant` is the corresponding data type in Visual Basic.

See `VARIANT` Type Codes Supported on page 6-9 for a list of the `VARIANT` type codes supported by COM Builder components.

See MATLAB to COM VARIANT Conversion Rules on page 6-11 and COM VARIANT to MATLAB Conversion Rules on page 6-15 for conversion rules between COM VARIANTs and MATLAB arrays.

VARIANT Type Codes Supported

VARIANT Type Code (C/C++)	C/C++ Type	Variant Type Code (Visual Basic)	Visual Basic Type	Definition
VT_EMPTY	-	vbEmpty	-	Uninitialized VARIANT
VT_I1	char	-	-	Signed one-byte character
VT_UI1	unsigned char	vbByte	Byte	Unsigned one-byte character
VT_I2	short	vbInteger	Integer	Signed two-byte integer
VT_UI2	unsigned short	-	-	Unsigned two-byte integer
VT_I4	long	vbLong	Long	Signed four-byte integer
VT_UI4	unsigned long	-	-	Unsigned four-byte integer
VT_R4	float	vbSingle	Single	IEEE four-byte floating-point value
VT_R8	double	vbDouble	Double	IEEE eight-byte floating-point value
VT_CY	CY+	vbCurrency	Currency	Currency value (64-bit integer, scaled by 10,000)
VT_BSTR	BSTR+	vbString	String	String value

VARIANT Type Codes Supported (Continued)

VARIANT Type Code (C/C++)	C/C++ Type	Variant Type Code (Visual Basic)	Visual Basic Type	Definition
VT_ERROR	SCODE+	vbError	-	A HRESULT (Signed four-byte integer representing a COM error code)
VT_DATE	DATE+	vbDate	Date	Eight-byte floating point value representing date and time
VT_INT	int	-	-	Signed integer; equivalent to type int
VT_UINT	unsigned int	-	-	Unsigned integer; equivalent to type unsigned int
VT_DECIMAL	DECIMAL+	vbDecimal	-	96-bit (12-byte) unsigned integer, scaled by a variable power of 10
VT_BOOL	VARIANT_BOOL+	vbBoolean	Boolean	Two-byte Boolean value (0xFFFF = True; 0x0000 = False)
VT_DISPATCH	IDispatch*	vbObject	Object	IDispatch* pointer to an object
VT_VARIANT	VARIANT+	vbVariant	Variant	VARIANT (can only be specified if combined with VT_BYREF or VT_ARRAY)

VARIANT Type Codes Supported (Continued)

VARIANT Type Code (C/C++)	C/C++ Type	Variant Type Code (Visual Basic)	Visual Basic Type	Definition
<*anything*>|VT_ARRAY				Bitwise combine VT_ARRAY with any basic type to declare as an array
<*anything*>|VT_BYREF				Bitwise combine VT_BYREF with any basic type to declare as a reference to a value
+ Denotes Windows-specific type. Not part of standard C/C++.				

MATLAB to COM VARIANT Conversion Rules

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
cell	A 1-by-1 cell array converts to a single VARIANT with a type conforming to the conversion rule for the MATLAB data type of the cell contents.	A multidimensional cell array converts to a VARIANT of type VT_VARIANT|VT_ARRAY with the type of each array member conforming to the conversion rule for the MATLAB data type of the corresponding cell.	

MATLAB to COM VARIANT Conversion Rules (Continued)

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
structure	VT_DISPATCH	VT_DISPATCH	A MATLAB struct array is converted to an MWStruct object. (See "Class MWStruct" on page 8-16.) This object is passed as a VT_DISPATCH type.
char	A 1-by-1 char matrix converts to a VARIANT of type VT_BSTR with string length = 1.	A 1-by-L char matrix is assumed to represent a string of length Lin MATLAB. This case converts to a VARIANT of type VT_BSTR with a string length = L. char matrices of more than one row, or of a higher dimensionality convert to a VARIANT of type VT_BSTR|VT_ARRAY. Each string in the converted array is of length 1 and corresponds to each character in the original matrix.	Arrays of strings are not supported as char matrices. To pass an array of strings, use a cell array of 1-by-L char matrices.
sparse	VT_DISPAATCH	VT_DISPATCH	A MATLAB sparse array is converted to an MWSparse object. (See "Class MWSparse" on page 8-26.) This object is passed as a VT_DISPATCH type.

MATLAB to COM VARIANT Conversion Rules (Continued)

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
double	A real 1-by-1 double matrix converts to a VARIANT of type VT_R8. A complex 1-by-1 double matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional double matrix converts to a VARIANT of type VT_R8|VT_ARRAY. A complex multidimensional double matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled M-functions using the MWComplex class. See "Class MWComplex" on page 8-24.)
single	A real 1-by-1 single matrix converts to a VARIANT of type VT_R4. A complex 1-by-1 single matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional single matrix converts to a VARIANT of type VT_R4|VT_ARRAY. A complex multidimensional single matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled M-functions using the MWComplex class.
int8	A real 1-by-1 int8 matrix converts to a VARIANT of type VT_I1. A complex 1-by-1 int8 matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional int8 matrix converts to a VARIANT of type VT_I1|VT_ARRAY. A complex multidimensional int8 matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled M-functions using the MWComplex class.

MATLAB to COM VARIANT Conversion Rules (Continued)

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
uint8	A real 1-by-1 uint8 matrix converts to a VARIANT of type VT_UI1. A complex 1-by-1 uint8 matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional uint8 matrix converts to a VARIANT of type VT_UI1|VT_ARRAY.A complex multidimensional uint8 matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled M-functions using the MWComplex class.
int16	A real 1-by-1 int16 matrix converts to a VARIANT of type VT_I2. A complex 1-by-1 int16 matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional int16 matrix converts to a VARIANT of type VT_I2|VT_ARRAY. A complex multidimensional int16 matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled M-functions using the MWComplex class.
uint16	A real 1-by-1 uint16 matrix converts to a VARIANT of type VT_UI2. A complex 1-by-1 uint16 matrix converts to a VARIANT of type VT_DISPATCH.	A real multidimensional uint16 matrix converts to a VARIANT of type VT_UI2|VT_ARRAY. A complex multidimensional uint16 matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled M-functions using the MWComplex class.

MATLAB to COM VARIANT Conversion Rules (Continued)

MATLAB Data Type	VARIANT Type for Scalar Data	VARIANT Type for Array Data	Comments
int32	A 1-by-1 int32 matrix converts to a VARIANT of type VT_I4. A complex 1-by-1 int32 matrix converts to a VARIANT of type VT_DISPATCH.	A multidimensional int32 matrix converts to a VARIANT of type VT_I4|VT_ARRAY. A complex multidimensional int32 matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled M-functions using the MWComplex class.
uint32	A 1-by-1 uint32 matrix converts to a VARIANT of type VT_UI4. A complex 1-by-1 uint32 matrix converts to a VARIANT of type VT_DISPATCH.	A multidimensional uint32 matrix converts to a VARIANT of type VT_UI4|VT_ARRAY. A complex multidimensional uint32 matrix converts to a VARIANT of type VT_DISPATCH.	Complex arrays are passed to and from compiled M-functions using the MWComplex class.
Function handle	VT_EMPTY	VT_EMPTY	Not supported
Java class	VT_EMPTY	VT_EMPTY	Not supported
User class	VT_EMPTY	VT_EMPTY	Not supported
logical	VT_Bool	VT_Bool|VT_ARRAY	

COM VARIANT to MATLAB Conversion Rules

VARIANT Type	MATLAB Data Type (scalar or array data)	Comments
VT_EMPTY	N/A	Empty array created.
VT_I1	int8	
VT_UI1	uint8	

COM VARIANT to MATLAB Conversion Rules (Continued)

VARIANT Type	MATLAB Data Type (scalar or array data)	Comments
VT_I2	int16	
VT_UI2	uint16	
VT_I4	int32	
VT_UI4	uint32	
VT_R4	single	
VT_R8	double	
VT_CY	double	
VT_BSTR	char	A VARIANT of type VT_BSTR converts to a 1-by-L MATLAB char array, where L = the length of the string to be converted. A VARIANT of type VT_BSTR|VT_ARRAY converts to a MATLAB cell array of 1-by-L char arrays.
VT_ERROR	int32	
VT_DATE	double	VARIANT dates are stored as doubles starting at midnight Dec. 31, 1899. MATLAB dates are stored as doubles starting at 0/0/00 00:00:00. Therefore, a VARIANT date of 0.0 maps to a MATLAB numeric date of 693960.0. VARIANT dates are converted to MATLAB double types and incremented by 693960.0. VARIANT dates can be optionally converted to strings. See "Data Conversion Flags" on page 6-20

COM VARIANT to MATLAB Conversion Rules (Continued)

VARIANT Type	MATLAB Data Type (scalar or array data)	Comments
		for more information on type coercion.
VT_INT	int32	
VT_UINT	unit32	
VT_DECIMAL	double	
VT_BOOL	logical	
VT_DISPATCH	*(varies)*	IDispatch* pointers are treated within the context of what they point to. Objects must be supported types with known data extraction and conversion rules, or expose a generic Value property that points to a single VARIANT type. Data extracted from an object is converted based upon the rules for the particular VARIANT obtained. Currently, support exists for Excel Range objects as well as COM Builder types MWStruct, MWComplex, MWSparse, and MWArg. See "Utility Library Classes" on page 8-3 for information on COM Builder types.

COM VARIANT to MATLAB Conversion Rules (Continued)

VARIANT Type	MATLAB Data Type (scalar or array data)	Comments
<*anything*>|VT_BYREF	(*varies*)	Pointers to any of the basic types are processed according to the rules for what they point to. The resulting MATLAB array contains a deep copy of the values.
<*anything*>|VT_ARRAY	(*varies*)	Multidimensional VARIANT arrays convert to multidimensional MATLAB arrays, each element converted according to the rules for the basic types. Multidimensional VARIANT arrays of type VT_VARIANT|VT_ARRAY convert to multidimensional cell arrays, each cell converted according to the rules for that specific type.

Array Formatting Flags

COM Builder components have flags that control how array data is formatted in both directions. Generally, you should develop client code that matches the intended inputs and outputs of the MATLAB functions with the corresponding methods on the compiled COM objects, in accordance with the rules listed in MATLAB to COM VARIANT Conversion Rules on page 6-11 and COM VARIANT to MATLAB Conversion Rules on page 6-15. In some cases this is not possible, for example, when existing MATLAB code is used in conjunction with a third-party product like Excel.

The following table shows the array formatting flags.

Array Formatting Flags

Flag	Description
InputArrayFormat	Defines the array formatting rule used on input arrays. An input array is a VARIANT array, created by the client, sent as an input parameter to a method call on a compiled COM object. Valid values for this flag are mwArrayFormatAsIs, mwArrayFormatMatrix, and mwArrayFormatCell. mwArrayFormatAsIs passes the array unchanged. mwArrayFormatMatrix (default) formats all arrays as matrices. When the input VARIANT is of type VT_ARRAY| *type*, where *type* is any numeric type, this flag has no effect. When the input VARIANT is of type VT_VARIANT|VT_ARRAY, VARIANTs in the array are examined. If they are single-valued and homogeneous in type, a MATLAB matrix of the appropriate type is produced instead of a cell array. mwArrayFormatCell interprets all arrays as MATLAB cell arrays.
InputArrayIndFlag	Sets the input array indirection level used with the InputArrayFormat flag (applicable only to nested arrays, i.e., VARIANT arrays of VARIANTs, which themselves are arrays). The default value for this flag is zero, which applies the InputArrayFormat flag to the outermost array. When this flag is greater than zero, e.g., equal to N, the formatting rule attempts to apply itself to the Nth level of nesting.
OutputArrayFormat	Defines the array formatting rule used on output arrays. An output array is a MATLAB array, created by the compiled COM object, sent as an output parameter from a method call to the client. The values for this flag, mwArrayFormatAsIs, mwArrayFormatMatrix, and mwArrayFormatCell, cause the same behavior as the corresponding InputArrayFormat flag values.

Array Formatting Flags (Continued)

Flag	Description
OutputArrayIndFlag	(Applies to nested cell arrays only.) Output array indirection level used with the OutputArrayFormat flag. This flag works exactly like InputArrayIndFlag.
AutoResizeOutput	(Applies to Excel ranges only.) When the target output from a method call is a range of cells in an Excel worksheet and the output array size and shape is not known at the time of the call, set this flag to True to resize each Excel range to fit the output array.
TransposeOutput	Set this flag to True to transpose the output arguments. Useful when calling a COM Builder component from Excel where the MATLAB function returns outputs as row vectors, and you want the data in columns.

Data Conversion Flags

COM Builder components contain flags to control the conversion of certain VARIANT types to MATLAB types. These flags are as follows:

- “CoerceNumericToType” on page 6-20
- “InputDateFormat” on page 6-21
- “OutputAsDate As Boolean” on page 6-21
- “DateBias As Long” on page 6-21

CoerceNumericToType

This flag tells the data converter to convert all numeric VARIANT data to one specific MATLAB type. VARIANT type codes affected by this flag are VT_I1, VT_UI1, VT_I2, VT_UI2, VT_I4, VT_UI4, VT_R4, VT_R8, VT_CY, VT_DECIMAL, VT_INT, VT_UINT, VT_ERROR, VT_BOOL, and VT_DATE. Valid values for this flag are mwTypeDefault, mwTypeChar, mwTypeDouble, mwTypeSingle, mwTypeLogical, mwTypeInt8, mwTypeUint8, mwTypeInt16, mwTypeUint16, mwTypeInt32, and mwTypeUint32.

The default for this flag, `mwTypeDefault`, converts numeric data according to the rules listed in "Data Conversion Rules" on page 6-8.

InputDateFormat

This flag tells the data converter how to convert `VARIANT` dates to MATLAB dates. Valid values for this flag are `mwDateFormatNumeric` (default) and `mwDateFormatString`. The default converts `VARIANT` dates according to the rule listed in `VARIANT` Type Codes Supported on page 6-9 . The `mwDateFormatString` flag converts a `VARIANT` date to its string representation. This flag only affects `VARIANT` type code `VT_DATE`.

OutputAsDate As Boolean

This flag instructs the data converter to process an output argument as a date. By default, numeric dates that are output parameters from compiled MATLAB functions are passed as `Doubles` that need to be decremented by the COM date bias (693960) as well as coerced to COM dates. Set this flag to `True` to convert all output values of type `Double`.

DateBias As Long

This flag sets the date bias for performing COM to MATLAB numeric date conversions. The default value of this property is 693960, which represents the difference between the COM `Date` type and MATLAB numeric dates. This flag allows existing MATLAB code that already performs the increment of numeric dates by 693960 to be used unchanged with COM Builder components. To process dates with such code, set this property to `0`.

Calling Conventions

When you use encapsulated MATLAB functions in your code, you might need to understand some or all of the following aspects of MATLAB Builder for COM processes.

Producing a COM Class

Producing a COM class requires the generation of

- A class definition file in Interface Description Language (IDL)
- One or more associated C++ class definition/implementation files

COM Builder automatically produces the necessary IDL and C/C++ code to build each COM class in the component. This process is generally transparent to you when you use COM Builder to generate a COM component, and to users of the COM component when they program with it.

For information about IDL and C++ coding rules for building COM objects and for mappings to other languages, see articles in the MSDN library.

The following table shows the mapping of a generic M-function to IDL code and to Visual Basic.

Generic M-Code	`function [Y1, Y2, ..., varargout] = foo(X1, X2, ..., varargin)`

IDL Code	`HRESULT foo([in] long nargout,`
	`[in,out] VARIANT* Y1,`
	`[in,out] VARIANT* Y2,`
	`.`
	`.`
	`[in,out] VARIANT* varargout,`
	`[in] VARIANT X1,`
	`[in] VARIANT X2,`
	`.`
	`.`
	`[in] VARIANT varargin);`

Visual Basic Code	`Sub foo(nargout As Long, _`
	`Y1 As Variant, _`
	`Y2 As Variant, _`
	`.`
	`.`
	`varargout As Variant, _`
	`X1 As Variant, _`
	`X2 As Varaint, _`
	`.`
	`.`
	`varargin As Variant)`

IDL Mapping

The IDL function definition is generated by producing a function with the same name as the original M-function and an argument list containing all inputs and outputs of the original plus one additional parameter, `nargout`.

When present, the `nargout` parameter is an `[in]` parameter of type `long`. It is always the first argument in the list. This parameter allows correct passage of the MATLAB `nargout` parameter to the compiled M-code. The `nargout`

parameter is not produced if you encapsulate an M-function containing no outputs.

Following the `nargout` parameter, the outputs are listed in the order they appear on the left side of the MATLAB function, and are tagged as `[in,out]`, meaning that they are passed in both directions.

The function inputs are listed next, appearing in the same order as they do on the right side of the original function. All inputs are tagged as `[in]` parameters.

When present, the optional `varargin`/`varargout` parameters are always listed as the last input parameters and the last output parameters. All parameters other than `nargout` are passed as COM `VARIANT` types. "Data Conversion Rules" on page 6-8 lists the rules for conversion between MATLAB arrays and COM `VARIANT`s.

Visual Basic Mapping

Visual Basic provides native support for COM VARIANTs with the `Variant` type, as well as implicit conversions for all Visual Basic basic types to and from Variants. In general, arrays/scalars of any Visual Basic basic type, as well as arrays/scalars of Variant types, can be passed as arguments.

COM Builder components also provide direct support for the Excel `Range` object, used by Visual Basic for Applications to represent a range of cells in an Excel worksheet.

See the Visual Basic for Applications documentation included with Microsoft Excel for more information on Visual Basic data types.

See the MSDN Library for more information about Visual Basic and about Excel `Range` manipulation.

Functions — Alphabetical List

componentinfo

Purpose

Query system registry for details about a component created with MATLAB Builder for COM

Syntax

```
componentinfo
componentinfo (component_name)
componentinfo (component_name, major_revision_number)
componentinfo (component_name, major_revision_number, minor_revision_number)
```

Arguments

component_name	MATLAB string providing the name of a MATLAB Builder for COM component. Names are case sensitive. If this argument is not supplied, the function returns information on all installed components.
major_revision_number	Component major revision number. If this argument is not supplied, the function returns information on all major revisions.
minor_revision_number	Component minor revision number. Default value is `0`.

Description

`componentinfo`

returns information for all components installed on the system.

`componentinfo (`*`component_name`*`)`

returns information for all revisions of *`component_name`*.

`componentinfo (`*`component_name`*`, `*`major_revision_number`*`)`

returns information for the most recent minor revision corresponding to *`major_revision_number`* of *`component_name`*..

`componentinfo (`*`component_name`*`, `*`major_revision_number`*`, `*`minor_revision_number`*`)`

returns information for the specific major and minor version of *component_name*.

The return value is an array of structures representing all the registry and type information needed to load and use the component.

When you supply a component name, *major_revision_number* and *minor_revision_number* are interpreted as shown below.

Value	Information Returned
> 0	Information on a specific major and minor revision
0	Information on the most recent revision When omitted, *minor_revision_number* is assumed to be equal to 0.
< 0	Information on all versions

The information about a component has the fields shown in the following table.

Registry Information Returned by componentinfo

Field	Description
Name	Component name
TypeLib	Component type library
LIBID	Component type library GUID
MajorRev	Major version number
MinorRev	Minor version number
FileName	Type library filename and path. Since all Excel Builder components have the type library bound into the DLL, this filename is the same as the DLL name and path.

Field	Description
`Interfaces`	An array of structures defining all interface definitions in the type library. Each structure contains two fields: • `Name` — Interface name • `IID` — Interface GUID

`CoClasses`	An array of structures defining all COM classes in the component. Each structure contains these fields:
	• `Name` — Class name
	• `CLSID` — GUID of the class
	• `ProgID` — Version dependent program ID
	• `VerIndProgID` — Version independent program ID
	• `InprocServer32` — Full name and path to component DLL
	• `Methods` — A structure containing function prototypes of all class methods defined for this interface. This structure contains four fields:
	  - `IDL` — An array of Interface Description Language function prototypes
	  - `M` — An array of MATLAB function prototypes
	  - `C` — An array of C-language function prototypes
	  - `VB` — An array of VBA function prototypes
	  • `Properties` — A cell array containing the names of all class properties.

	• `Events` — A structure containing function prototypes of all events defined for this class. This structure contains four fields: • `IDL` — An array of IDL (Interface Description Language) function prototypes. • `M` — An array of MATLAB function prototypes. • `C` — An array of C-Language function prototypes. • `VB` — An array of VBA function prototypes

Examples

Function Call	Returns
`Info = componentinfo`	Information for all installed components.
`Info = componentinfo('mycomponent')`	Information for all revisions of `mycomponent`.
`Info = componentinfo('mycomponent',1,0)`	Information for revision 1.0 of `mycomponent`.

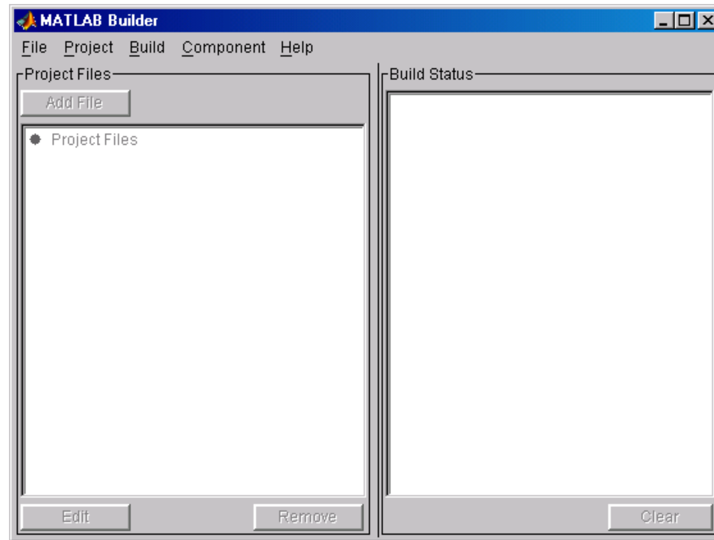
comtool

Purpose Open graphical user interface to MATLAB Builder for COM

Syntax

```
comtool
```

Description The `comtool` command displays the MATLAB Builder window, which is the graphical user interface (GUI) for MATLAB Builder for COM.



8

Utility Library

Referencing the Utility Classes (p. 8-2)	Referencing the classes in your programming environment
Utility Library Classes (p. 8-3)	Describes the classes provided in the Utility Library.
Enumerations (p. 8-31)	Describes the three provided sets of constants.

Referencing the Utility Classes

This section describes the `MWComUtil` library provided with MATLAB Builder for COM. This library is freely distributable and includes several functions used in array processing, as well as type definitions used in data conversion. This library is contained in the file `mwcomutil.dll`. It must be registered once on each machine that uses COM Builder components.

Register the `MWComUtil` library at the DOS command prompt with the command

```
mwregsvr mwcomutil.dll
```

The `MWComUtil` library includes seven classes (see "Utility Library Classes" on page 8-3) and three enumerated types (see "Enumerations" on page 8-31). Before using these types, you must make explicit references to the `MWComUtil` type libraries in the Visual Basic IDE. To do this select **Tools>References** from the main menu of the VB editor. The References dialog box appears with a scrollable list of available type libraries. From this list select **MWComUtil 1.0 Type Library** and click **OK**.

Utility Library Classes

The MATLAB Builder for COM Utility library provides several classes:

- "Class MWUtil" on page 8-3
- "Class MWFlags" on page 8-10
- "Class MWStruct" on page 8-16
- "Class MWField" on page 8-23
- "Class MWComplex" on page 8-24
- "Class MWSparse" on page 8-26
- "Class MWArg" on page 8-29

Class MWUtil

The `MWUtil` class contains a set of static utility methods used in array processing and application initialization. This class is implemented internally as a singleton (only one global instance of this class per instance of Excel). It is most efficient to declare one variable of this type in global scope within each module that uses it. The methods of `MWUtil` are

- "Sub MWInitApplication(pApp As Object)" on page 8-3
- "Sub MWPack(pVarArg, [Var0], [Var1], ... ,[Var31])" on page 8-5
- "Sub MWUnpack(VarArg, [nStartAt As Long], [bAutoResize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])" on page 8-6
- "Sub MWDate2VariantDate(pVar)" on page 8-8

The function prototypes use Visual Basic syntax.

Sub MWInitApplication(pApp As Object)

Initializes the library with the current instance of Excel.

Parameters.

Argument	Type	Description
pApp	Object	A valid reference to the current Excel application

Return Value. None.

Remarks. This function must be called once for each session of Excel that uses COM Builder components. An error is generated if a method call is made to a member class of any COM Builder component, and the library has not been initialized.

Example. This Visual Basic sample initializes the `MWComUtil` library with the current instance of Excel. A global variable of type `Object` named `MCLUtil` holds an instance of the `MWUtil` class, and another global variable of type `Boolean` named `bModuleInitialized` stores the status of the initialization process. The private subroutine `InitModule()` creates an instance of the `MWComUtil` class and calls the `MWInitApplication` method with an argument of `Application`. Once this function succeeds, all subsequent calls exit without recreating the object.

```
Dim MCLUtil As Object
Dim bModuleInitialized As Boolean

Private Sub InitModule()
    If Not bModuleInitialized Then
        On Error GoTo Handle_Error
        If MCLUtil Is Nothing Then
            Set MCLUtil = CreateObject("MWComUtil.MWUtil")
        End If
        Call MCLUtil.MWInitApplication(Application)
        bModuleInitialized = True
        Exit Sub
Handle_Error:
        bModuleInitialized = False
    End If
End Sub
```

Sub MWPack(pVarArg, [Var0], [Var1], ... ,[Var31])

Packs a variable length list of `Variant` arguments into a single `Variant` array. This function is typically used for creating a `varargin` cell from a list of separate inputs. Each input in the list is added to the array only if it is nonempty and nonmissing. (In Visual Basic, a missing parameter is denoted by a `Variant` type of `vbError` with a value of `&H80020004`.)

Parameters.

Argument	Type	Description
`pVarArg`	`Variant`	Receives the resulting array
`[Var0], [Var1], ...`	`Variant`	Optional list of `Variants` to pack into the array. From 0 to 32 arguments can be passed.

Return Value. None.

Remarks. This function always frees the contents of `pVarArg` before processing the list.

Example. This example uses `MWPack` in a formula function to produce a `varargin` cell to pass as an input parameter to a method compiled from a MATLAB function with the signature

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

The function returns the sum of the elements in `varargin`. Assume that this function is a method of a class named `myclass` that is included in a component named `mycomponent` with a version of 1.0. The Visual Basic function allows up to 10 inputs, and returns the result `y`. If an error occurs, the function returns the error string. This function assumes that `MWInitApplication` has been previously called.

```
Function mysum(Optional V0 As Variant, _
               Optional V1 As Variant, _
               Optional V2 As Variant, _
               Optional V3 As Variant, _
               Optional V4 As Variant, _
               Optional V5 As Variant, _
               Optional V6 As Variant, _
               Optional V7 As Variant, _
               Optional V8 As Variant, _
               Optional V9 As Variant) As Variant
Dim y As Variant
Dim varargin As Variant
Dim aClass As Object
Dim aUtil As Object

    On Error Goto Handle_Error
    Set aClass = CreateObject("mycomponent.myclass.1_0")
    Set aUtil = CreateObject("MWComUtil.MWUtil")
    Call aUtil.MWPack(varargin,V0,V1,V2,V3,V4,V5,V6,V7,V8,V9)
    Call aClass.mysum(1, y, varargin)
    mysum = y
    Exit Function
Handle_Error:
    mysum = Err.Description
End Function
```

Sub MWUnpack(VarArg, [nStartAt As Long], [bAutoResize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])

Unpacks an array of `Variants` into individual `Variant` arguments. This function provides the reverse functionality of `MWPack` and is typically used to process a `varargout` cell into individual `Variants`.

Parameters.

Argument	Type	Description
VarArg	Variant	Input array of Variants to be processed
nStartAt	Long	Optional starting index (zero-based) in the array to begin processing. Default = 0.
bAutoResize	Boolean	Optional auto-resize flag. If this flag is True, any Excel range output arguments are resized to fit the dimensions of the Variant to be copied. The resizing process is applied relative to the upper left corner of the supplied range. Default = False.
[pVar0],[pVar1], ...	Variant	Optional list of Variants to receive the array items contained in VarArg. From 0 to 32 arguments can be passed.

Return Value. None.

Remarks. This function can process a Variant array in one single call or through multiple calls using the nStartAt parameter.

Example. This example uses MWUnpack to process a varargout cell into several Excel ranges, while auto-resizing each range. The varargout parameter is supplied from a method that has been compiled from the MATLAB function.

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

This function produces a sequence of `nargout` random column vectors, with the length of the ith vector equal to i. Assume that this function is included in a class named `myclass` that is included in a component named `mycomponent` with a version of 1.0. The Visual Basic subroutine takes no arguments and places the results into Excel columns starting at A1, B1, C1, and D1. If an error occurs, a message box displays the error text. This function assumes that `MWInitApplication` has been previously called.

```
Sub GenVectors()
    Dim aClass As Object
    Dim aUtil As Object
    Dim v As Variant
    Dim R1 As Range
    Dim R2 As Range
    Dim R3 As Range
    Dim R4 As Range

    On Error GoTo Handle_Error
    Set aClass = CreateObject("mycomponent.myclass.1_0")
    Set aUtil = CreateObject("MWComUtil.MWUtil")
    Set R1 = Range("A1")
    Set R2 = Range("B1")
    Set R3 = Range("C1")
    Set R4 = Range("D1")
    Call aClass.randvectors(4, v)
    Call aUtil.MWUnpack(v,0,True,R1,R2,R3,R4)
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
```

Sub MWDate2VariantDate(pVar)

Converts output dates from MATLAB to `Variant` dates.

Parameters.

Argument	Type	Description
pVar	Variant	Variant to be converted

Return Value. None.

Remarks. MATLAB handles dates as double-precision floating-point numbers with 0.0 representing 0/0/00 00:00:00. By default, numeric dates that are output parameters from compiled MATLAB functions are passed as Doubles that need to be decremented by the COM date bias as well as coerced to COM dates. The MWDate2VariantDate method performs this transformation and additionally converts dates in string form to COM date types.

Example. This example uses MWDate2VariantDate to process numeric dates returned from a method compiled from the following MATLAB function.

```
function x = getdates(n, inc)
    y = now;
    for i=1:n
        x(i,1) = y + (i-1)*inc;
    end
```

This function produces an n-length column vector of numeric values representing dates starting from the current date and time with each element incremented by inc days. Assume that this function is included in a class named myclass that is included in a component named mycomponent with a version of 1.0. The subroutine takes an Excel range and a Double as inputs and places the generated dates into the supplied range. If an error occurs, a message box displays the error text. This function assumes that MWInitApplication has been previously called.

```
Sub GenDates(R As Range, inc As Double)
    Dim aClass As Object
    Dim aUtil As Object

    On Error GoTo Handle_Error
    Set aClass = CreateObject("mycomponent.myclass.1_0")
```

```
    Set aUtil = CreateObject("MWComUtil.MWUtil")
    Call aClass.getdates(1, R, R.Rows.Count, inc)
    Call aUtil.MWDate2VariantDate(R)
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
```

Class MWFlags

The `MWFlags` class contains a set of array formatting and data conversion flags (See "Data Conversion Rules" on page 6-8 for more information on conversion between MATLAB and COM Automation types). All COM Builder components contain a reference to an `MWFlags` object that can modify data conversion rules at the object level. This class contains these properties:

- "Property ArrayFormatFlags As MWArrayFormatFlags" on page 8-10
- "Property DataConversionFlags As MWDataConversionFlags" on page 8-13
- "Sub Clone(ppFlags As MWFlags)" on page 8-15

Property ArrayFormatFlags As MWArrayFormatFlags

The `ArrayFormatFlags` property controls array formatting (as a matrix or a cell array) and the application of these rules to nested arrays. The `MWArrayFormatFlags` class is a noncreatable class accessed through an `MWFlags` class instance. This class contains six properties:

- "Property InputArrayFormat As mwArrayFormat" on page 8-11
- "Property InputArrayIndFlag As Long" on page 8-11
- "Property OutputArrayFormat As mwArrayFormat" on page 8-12
- "Property OutputArrayIndFlag As Long" on page 8-12
- "Property AutoResizeOutput As Boolean" on page 8-13
- "Property TransposeOutput As Boolean" on page 8-13

Property InputArrayFormat As mwArrayFormat. This property of type `mwArrayFormat` controls the formatting of arrays passed as input parameters to COM Builder class methods. The default value is `mwArrayFormatMatrix`. The behaviors indicated by this flag are listed in the next table.

Array Formatting Rules for Input Arrays

Value	Behavior
`mwArrayFormatAsIs`	Converts arrays according to the default conversion rules listed in "Data Conversion Rules" on page 6-8.
`mwArrayFormatCell`	Coerces all arrays into cell arrays. Input scalar or numeric array arguments are converted to cell arrays with each cell containing a scalar value for the respective index.
`mwArrayFormatMatrix`	Coerces all arrays into matrices. When an input argument is encountered that is an array of `Variants` (the default behavior is to convert it to a cell array), the data converter converts this array to a matrix if each `Variant` is single valued, and all elements are homogeneous and of a numeric type. If this conversion is not possible, creates a cell array.

Property InputArrayIndFlag As Long. This property governs the level at which to apply the rule set by the `InputArrayFormat` property for nested arrays (an array of `Variants` is passed and each element of the array is an array itself). It is not necessary to modify this flag for `varargin` parameters. The data conversion code automatically increments the value of this flag by 1 for `varargin` cells, thus applying the `InputArrayFormat` flag to each cell of a `varargin` parameter. The default value is `0`.

Property OutputArrayFormat As mwArrayFormat. This property of type `mwArrayFormat` controls the formatting of arrays passed as output parameters to COM Builder class methods. The default value is `mwArrayFormatAsIs`. The behaviors indicated by this flag are listed in the next table.

Array Formatting Rules for Output Arrays

Value	Behavior
`mwArrayFormatAsIs`	Converts arrays according to the default conversion rules listed in MATLAB to COM `VARIANT` Conversion Rules on page 6-11.
`mwArrayFormatMatrix`	Coerces all arrays into matrices. When an output cell array argument is encountered (the default behavior converts it to an array of `Variants`), the data converter converts this array to a `Variant` that contains a simple numeric array if each cell is single valued, and all elements are homogeneous and of a numeric type. If this conversion is not possible, an array of `Variants` is created.
`mwArrayFormatCell`	Coerces all output arrays into arrays of `Variants`. Output scalar or numeric array arguments are converted to arrays of `Variants`, each `Variant` containing a scalar value for the respective index.

Property OutputArrayIndFlag As Long. This property is similar to the `InputArrayIndFalg` property, as it governs the level at which to apply the rule set by the `OutputArrayFormat` property for nested arrays. As with the input case, this flag is automatically incremented by 1 for a `varargout` parameter. The default value of this flag is `0`.

Property AutoResizeOutput As Boolean. This flag applies to Excel ranges only. When the target output from a method call is a range of cells in an Excel worksheet, and the output array size and shape is not known at the time of the call, setting this flag to `True` instructs the data conversion code to resize each Excel range to fit the output array. Resizing is applied relative to the upper left corner of each supplied range. The default value for this flag is `False`.

Property TransposeOutput As Boolean. Setting this flag to `True` transposes the output arguments. This flag is useful when processing an output parameter from a method call on an COM Builder component, where the MATLAB function returns outputs as row vectors, and you desire to place the data into columns. The default value for this flag is `False`.

Property DataConversionFlags As MWDataConversionFlags

The `DataConversionFlags` property controls how input variables are processed when type coercion is needed. The `MWDataConversionFlags` class is a noncreatable class accessed through an `MWFlags` class instance. This class contains these properties:

- "Property CoerceNumericToType As mwDataType" on page 8-13
- "Property InputDateFormat As mwDateFormat" on page 8-14
- "Example" on page 8-14
- "PropertyOutputAsDate As Boolean" on page 8-15
- "PropertyDateBias As Long" on page 8-15

Property CoerceNumericToType As mwDataType. This property converts all numeric input arguments to one specific MATLAB type. This flag is useful is when variables maintained within the Visual Basic code are different types, e.g., `Long`, `Integer`, etc., and all variables passed to the compiled MATLAB code must be doubles. The default value for this property is `mwTypeDefault`, which uses the default rules in COM `VARIANT` to MATLAB Conversion Rules on page 6-15COM `VARIANT` to MATLAB Conversion Rules on page 6-15.

Property InputDateFormat As mwDateFormat. This property converts dates passed as input parameters to method calls on COM Builder classes. The default value is `mwDateFormatNumeric`. The behaviors indicated by this flag are shown in the following table..

Conversion Rules for Input Dates

Value	Behavior
`mwDateFormatNumeric`	Convert dates to numeric values as indicated by the rule listed in COM `VARIANT` to MATLAB Conversion Rules on page 6-15.
`mwDateFormatString`	Convert input dates to strings.

Example. This example uses data conversion flags to reshape the output from a method compiled from a MATLAB function that produces an output vector of unknown length.

```
function p = myprimes(n)
if length(n)~=1, error('N must be a scalar'); end
if n < 2, p = zeros(1,0); return, end
p = 1:2:n;
q = length(p);
p(1) = 2;
for k = 3:2:sqrt(n)
    if p((k+1)/2)
        p(((k*k+1)/2):k:q) = 0;
    end
end
p = (p(p>0));
```

This function produces a row vector of all the prime numbers between 0 and `n`. Assume that this function is included in a class named `myclass` that is included in a component named `mycomponent` with a version of 1.0. The subroutine takes an Excel range and a `Double` as inputs, and places the generated prime numbers into the supplied range. The MATLAB function produces a row vector, although you want the output in column format. It also produces an unknown number of outputs, and you do not want to truncate any output. To handle these issues, set the `TransposeOutput` flag and the

AutoResizeOutput flag to True. In previous examples, the Visual Basic CreateObject function creates the necessary classes. This example uses an explicit type declaration for the aClass variable. As with previous examples, this function assumes that MWInitApplication has been previously called.

```
Sub GenPrimes(R As Range, n As Double)
    Dim aClass As mycomponent.myclass

    On Error GoTo Handle_Error
    Set aClass = New mycomponent.myclass
    aClass.MWFlags.ArrayFormatFlags.AutoResizeOutput = True
    aClass.MWFlags.ArrayFormatFlags.TransposeOutput = True
    Call aClass.myprimes(1, R, n)
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
```

PropertyOutputAsDate As Boolean. This property processes an output argument as a date. By default, numeric dates that are output parameters from compiled MATLAB functions are passed as Doubles that need to be decremented by the COM date bias (693960) as well as coerced to COM dates. Set this flag to True to convert all output values of type Double.

PropertyDateBias As Long. This property sets the date bias for performing COM to MATLAB numeric date conversions. The default value of this property is 693960, representing the difference between the COM Date type and MATLAB numeric dates. This flag allows existing MATLAB code that already performs the increment of numeric dates by 693960 to be used unchanged with COM Builder components. To process dates with such code, set this property to 0.

Sub Clone(ppFlags As MWFlags)

Creates a copy of an MWFlags object.

Parameters.

Argument	Type	Description
ppFlags	MWFlags	Reference to an uninitialized MWFlags object that receives the copy

Return Value. None

Remarks. Clone allocates a new MWFlags object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

Class MWStruct

The MWStruct class passes or receives a Struct type to or from a compiled class method. This class contains seven properties/methods:

- "Sub Initialize([varDims], [varFieldNames])" on page 8-16
- "Property Item([i0], [i1], ..., [i31]) As MWField" on page 8-18
- "Property NumberOfFields As Long" on page 8-21
- "Property NumberOfDims As Long" on page 8-21
- "Property Dims As Variant" on page 8-21
- "Property FieldNames As Variant" on page 8-21
- "Sub Clone(ppStruct As MWStruct)" on page 8-22

Sub Initialize([varDims], [varFieldNames])

This method allocates a structure array with a specified number and size of dimensions and a specified list of field names.

Parameters.

Argument	Type	Description
varDims	Variant	Optional array of dimensions
varFieldNames	Variant	Optional array of field names

Return Value. None.

Remarks. When created, an `MWStruct` object has a dimensionality of 1-by-1 and no fields. The `Initialize` method dimensions the array and adds a set of named fields to each element. Each time you call `Initialize` on the same object, it is redimensioned. If you do not supply the `varDims` argument, the existing number and size of the array's dimensions unchanged. If you do not supply the `varFieldNames` argument, the existing list of fields is not changed. Calling `Initialize` with no arguments leaves the array unchanged.

Example. The following Visual Basic code illustrates use of the `Initialize` method to dimension struct arrays.

```
Sub foo ()
    Dim x As MWStruct
    Dim y As MWStruct

    On Error Goto Handle_Error
    'Create 1X1 struct arrays with no fields for x, and y
    Set x = new MWStruct
    Set y = new MWStruct

    'Initialize x to be 2X2 with fields "red", "green", and "blue"
    Call x.Initialize(Array(2,2), Array("red", "green", "blue"))
    'Initialize y to be 1X5 with fields "name" and "age"
    Call y.Initialize(5, Array("name", "age"))

    'Re-dimension x to be 3X3 with the same field names
    Call x.Initialize(Array(3,3))

    'Add a new field to y
```

```
    Call y.Initialize(, Array("name", "age", "salary"))

    Exit Sub
Handle_Error:
    MsgBox(Err.Description)
End Sub
```

Property Item([i0], [i1], ..., [i31]) As MWField

The `Item` property is the default property of the `MWStruct` class. This property is used to set/get the value of a field at a particular index in the structure array.

Parameters.

Argument	Type	Description
i0,i1, ..., i31	Variant	Optional index arguments. Between 0 and 32 index arguments can be entered. To reference an element of the array, specify all indexes as well as the field name.

Remarks. When accessing a named field through this property, you must supply all dimensions of the requested field as well as the field name. This property always returns a single field value, and generates a bad index error if you provide an invalid or incomplete index list. Index arguments have four basic formats:

- Field name only

  This format may be used only in the case of a 1-by-1 structure array and returns the named field's value. For example:

  ```
  x("red") = 0.2
  x("green") = 0.4
  x("blue") = 0.6
  ```

In this example, the name of the `Item` property was neglected. This is possible since the `Item` property is the default property of the `MWStruct` class. In this case the two statements are equivalent:

```
x.Item("red") = 0.2
x("red") = 0.2
```

- Single index and field name

This format accesses array elements through a single subscripting notation. A single numeric index `n` followed by the field name returns the named field on the `n`th array element, navigating the array linearly in column-major order. For example, consider a 2-by-2 array of structures with fields `"red"`, `"green"` , and `"blue"` stored in a variable `x`. These two statements are equivalent:

```
y = x(2, "red")
y = x(2, 1, "red")
```

- All indices and field name

This format accesses an array element of an multidimensional array by specifying `n` indices. These statements access all four of the elements of the array in the previous example:

```
For I From 1 To 2
       For J From 1 To 2
              r(I, J) = x(I, J, "red")
              g(I, J) = x(I, J, "green")
              b(I, J) = x(I, J, "blue")
       Next
Next
```

- Array of indices and field name

This format accesses an array element by passing an array of indices and a field name. The next example rewrites the previous example using an index array:

```
Dim Index(1 To 2) As Integer
```

```
For I From 1 To 2
       Index(1) = I
       For J From 1 To 2
              Index(2) = J
              r(I, J) = x(Index, "red")
              g(I, J) = x(Index, "green")
              b(I, J) = x(Index, "blue")
       Next
Next
```

With these four formats, the `Item` property provides a very flexible indexing mechanism for structure arrays. Also note:

- You can combine the last two indexing formats. Several index arguments supplied in either scalar or array format are concatenated to form one index set. The combining stops when the number of dimensions has been reached. For example:

  ```
  Dim Index1(1 To 2) As Integer
  Dim Index2(1 To 2) As Integer

  Index1(1) = 1
  Index1(2) = 1
  Index2(1) = 3
  Index2(2) = 2
  x(Index1, Index2, 2, "red") = 0.5
  ```

The last statement resolves to

```
x(1, 1, 3, 2, 2, "red") = 0.5
```

- The field name must be the last index in the list. The following statement produces an error:

  ```
  y = x("blue", 1, 2)
  ```

- Field names are case sensitive.

Property NumberOfFields As Long

The read-only `NumberOfFields` property returns the number of fields in the structure array.

Property NumberOfDims As Long

The read-only `NumberOfDims` property returns the number of dimensions in the struct array.

Property Dims As Variant

The read-only `Dims` property returns an array of length `NumberOfDims` that contains the size of each dimension of the struct array.

Property FieldNames As Variant

The read-only `FieldNames` property returns an array of length `NumberOfFields` that contains the field names of the elements of the structure array.

Example. The next Visual Basic code sample illustrates how to access a two-dimensional structure array's fields when the field names and dimension sizes are not known in advance.

```
Sub foo ()
      Dim x As MWStruct
      Dim Dims as Variant
      Dim FieldNames As Variant


      On Error Goto Handle_Error
      '
      '... Call a method that returns an MWStruct in x
      '
      Dims = x.Dims
      FieldNames = x.FieldNames
      For I From 1 To Dims(1)
            For J From 1 To Dims(2)
                  For K From 1 To x.NumberOfFields
                        y = x(I,J,FieldNames(K))
                        ' ... Do something with y
```

```
                  Next
            Next
      Next
Exit Sub
Handle_Error:
      MsgBox(Err.Description)
End Sub
```

Sub Clone(ppStruct As MWStruct)

Creates a copy of an `MWStruct` object.

Parameters.

Argument	Type	Description
`ppStruct`	`MWStruct`	Reference to an uninitialized `MWStruct` object to receive the copy

Return Value. None

Remarks. `Clone` allocates a new `MWStruct` object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

Example. The following Visual Basic example illustrates the difference between assignment and `Clone` for `MWStruct` objects.

```
Sub foo ()
    Dim x1 As MWStruct
    Dim x2 As MWStruct
    Dim x3 As MWStruct

    On Error Goto Handle_Error
    Set x1 = new MWStruct
    x1("name") = "John Smith"
    x1("age") = 35
```

```
    'Set reference of x1 to x2
     Set x2 = x1
    'Create new object for x3 and copy contents of x1 into it
     Call x1.Clone(x3)
    'x2's "age" field is also modified 'x3's "age" field unchanged
     x1("age") = 50
         .
         .
         .
     Exit Sub
Handle_Error:
     MsgBox(Err.Description)
End Sub
```

Class MWField

The MWField class holds a single field reference in an MWStruct object. This class is noncreatable and contains four properties/methods:

- "Property Name As String" on page 8-23
- "Property Value As Variant" on page 8-23
- "Property MWFlags As MWFlags" on page 8-23
- "Sub Clone(ppField As MWField)" on page 8-24

Property Name As String

The name of the field (read only).

Property Value As Variant

Stores the field's value (read/write). The Value property is the default property of the MWField class. The value of a field can be any type that is coercible to a Variant, as well as object types.

Property MWFlags As MWFlags

Stores a reference to an MWFlags object. This property sets or gets the array formatting and data conversion flags for a particular field. Each field in a

structure has its own MWFlags property. This property overrides the value of any flags set on the object whose methods are called.

Sub Clone(ppField As MWField)

Creates a copy of an MWField object.

Parameters.

Argument	Type	Description
ppField	MWField	Reference to an uninitialized MWField object to receive the copy

Return Value. None.

Remarks. Clone allocates a new MWField object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

Class MWComplex

The MWComplex class passes or receives a complex numeric array into or from a compiled class method. This class contains four properties/methods:

- "Property Real As Variant" on page 8-24
- "Property Imag As Variant" on page 8-25
- "Property MWFlags As MWFlags" on page 8-26
- "Sub Clone(ppComplex As MWComplex)" on page 8-26

Property Real As Variant

Stores the real part of a complex array (read/write). The Real property is the default property of the MWComplex class. The value of this property can be any type coercible to a Variant, as well as object types, with the restriction that the underlying array must resolve to a numeric matrix (no cell data allowed).

Valid Visual Basic numeric types for complex arrays include `Byte`, `Integer`, `Long`, `Single`, `Double`, `Currency`, and `Variant/vbDecimal`.

Property Imag As Variant

Stores the imaginary part of a complex array (read/write). The `Imag` property is optional and can be `Empty` for a pure real array. If the `Imag` property is nonempty and the size and type of the underlying array do not match the size and type of the `Real` property's array, an error results when the object is used in a method call.

Example. The following Visual Basic code creates a complex array with the following entries:

```
    x = [ 1+i 1+2i
           2+i 2+2i ]
Sub foo()
    Dim x As MWComplex
    Dim rval(1 To 2, 1 To 2) As Double
    Dim ival(1 To 2, 1 To 2) As Double

    On Error Goto Handle_Error
    For I = 1 To 2
        For J = 1 To 2
                rval(I,J) = I
                ival(I,J) = J
        Next
    Next
    Set x = new MWComplex
    x.Real = rval
    x.Imag = ival
        .
        .
        .
    Exit Sub
Handle_Error:
    MsgBox(Err.Description)
End Sub
```

Property MWFlags As MWFlags

Stores a reference to an `MWFlags` object. This property sets or gets the array formatting and data conversion flags for a particular complex array. Each `MWComplex` object has its own `MWFlags` property. This property overrides the value of any flags set on the object whose methods are called.

Sub Clone(ppComplex As MWComplex)

Creates a copy of an `MWComplex` object.

Parameters.

Argument	Type	Description
`ppComplex`	`MWComplex`	Reference to an uninitialized `MWComplex` object to receive the copy

Return Value. None

Remarks. `Clone` allocates a new `MWComplex` object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

Class MWSparse

The `MWSparse` class passes or receives a two-dimensional sparse numeric array into or from a compiled class method. This class has seven properties/methods:

- "Property NumRows As Long" on page 8-27
- "Property NumColumns As Long" on page 8-27
- "Property RowIndex As Variant" on page 8-27
- "Property ColumnIndex As Variant" on page 8-27
- "Property Array As Variant" on page 8-27
- "Property MWFlags As MWFlags" on page 8-28
- "Sub Clone(ppSparse As MWSparse)" on page 8-28

Property NumRows As Long

Stores the row dimension for the array. The value of `NumRows` must be nonnegative. If the value is zero, the row index is taken from the maximum of the values in the `RowIndex` array.

Property NumColumns As Long

Stores the column dimension for the array. The value of `NumColumns` must be nonnegative. If the value is zero, the row index is taken from the maximum of the values in the `ColumnIndex` array.

Property RowIndex As Variant

Stores the array of row indices of the nonzero elements of the array. The value of this property can be any type coercible to a `Variant`, as well as object types, with the restriction that the underlying array must resolve to or be coercible to a numeric matrix of type `Long`. If the value of `NumRows` is nonzero and any row index is greater than `NumRows`, a bad-index error occurs. An error also results if the number of elements in the `RowIndex` array does not match the number of elements in the `Array` property's underlying array.

Property ColumnIndex As Variant

Stores the array of column indices of the nonzero elements of the array. The value of this property can be any type coercible to a `Variant`, as well as object types, with the restriction that the underlying array must resolve to or be coercible to a numeric matrix of type `Long`. If the value of `NumColumns` is nonzero and any column index is greater than `NumColumns`, a bad-index error occurs. An error also results if the number of elements in the `ColumnIndex` array does not match the number of elements in the `Array` property's underlying array.

Property Array As Variant

Stores the nonzero array values of the sparse array. The value of this property can be any type coercible to a `Variant`, as well as object types, with the restriction that the underlying array must resolve to or be coercible to a numeric matrix of type `Double` or `Boolean`.

Property MWFlags As MWFlags

Stores a reference to an `MWFlags` object. This property sets or gets the array formatting and data conversion flags for a particular sparse array. Each `MWSparse` object has its own `MWFlags` property. This property overrides the value of any flags set on the object whose methods are called.

Sub Clone(ppSparse As MWSparse)

Creates a copy of an `MWSparse` object.

Parameters.

Argument	Type	Description
`ppSparse`	`MWSparse`	Reference to an uninitialized `MWSparse` object to receive the copy

Return Value. None.

Remarks. `Clone` allocates a new `MWSparse` object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

Example. The following Visual Basic sample creates a 5-by-5 tridiagonal sparse array with the following entries:

```
X = [ 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 ]

Sub foo()
    Dim x As MWSparse
    Dim rows(1 To 13) As Long
    Dim cols(1 To 13) As Long
    Dim vals(1 To 13) As Double
```

```
    Dim I As Long, K As Long

    On Error GoTo Handle_Error
    K = 1
    For I = 1 To 4
        rows(K) = I
        cols(K) = I + 1
        vals(K) = -1
        K = K + 1
        rows(K) = I
        cols(K) = I
        vals(K) = 2
        K = K + 1
        rows(K) = I + 1
        cols(K) = I
        vals(K) = -1
        K = K + 1
    Next
    rows(K) = 5
    cols(K) = 5
    vals(K) = 2
    Set x = New MWSparse
    x.NumRows = 5
    x.NumColumns = 5
    x.RowIndex = rows
    x.ColumnIndex = cols
    x.Array = vals
        .
        .
        .
    Exit Sub
Handle_Error:
    MsgBox (Err.Description)
End Sub
```

Class MWArg

The MWArg class passes a generic argument into a compiled class method. This class passes an argument for which the data conversion flags are changed for that one argument. This class has three properties/methods:

- "Property Value As Variant" on page 8-30
- "Property MWFlags As MWFlags" on page 8-30
- "Sub Clone(ppArg As MWArg)" on page 8-30

Property Value As Variant

The `Value` property stores the actual argument to pass. Any type that can be passed to a compiled method is valid for this property.

Property MWFlags As MWFlags

Stores a reference to an `MWFlags` object. This property sets or gets the array formatting and data conversion flags for a particular argument. Each `MWArg` object has its own `MWFlags` property. This property overrides the value of any flags set on the object whose methods are called.

Sub Clone(ppArg As MWArg)

Creates a copy of an `MWArg` object.

Parameters.

Argument	Type	Description
`ppArg`	`MWArg`	Reference to an uninitialized `MWArg` object to receive the copy

Return Value. None.

Remarks. Clone allocates a new `MWArg` object and creates a deep copy of the object's contents. Call this function when a separate object is required instead of a shared copy of an existing object reference.

Enumerations

The MATLAB Builder for COM Utility library provides three enumerations (sets of constants):

- "Enum mwArrayFormat" on page 8-31
- "Enum mwDataType" on page 8-31
- "Enum mwDateFormat" on page 8-32

Enum mwArrayFormat

The `mwArrayFormat` enumeration is a set of constants that denote an array formatting rule for data conversion.

mwArrayFormat Values

Constant	Numeric Value	Description
`mwArrayFormatAsIs`	0	Do not reformat the array.
`mwArrayFormatMatrix`	1	Format the array as a matrix.
`mwArrayFormatCell`	2	Format the array as a cell array.

Enum mwDataType

The `mwDataType` enumeration is a set of constants that denote a MATLAB numeric type.

mwDataType Values

Constant	Numeric Value	MATLAB Type
`mwTypeDefault`	0	`N/A`
`mwTypeLogical`	3	`logical`
`mwTypeChar`	4	`char`

mwDataType Values (Continued)

Constant	Numeric Value	MATLAB Type
mwTypeDouble	6	double
mwTypeSingle	7	single
mwTypeInt8	8	int8
mwTypeUint8	9	uint8
mwTypeInt16	10	int16
mwTypeUint16	11	uint16
mwTypeInt32	12	int32
mwTypeUint32	13	uint32

Enum mwDateFormat

The `mwDateFormat` enumeration is a set of constants that denote a formatting rule for dates.

mwDateFormat Values

Constant	Numeric Value	Description
mwDateFormatNumeric	0	Format dates as numeric values
mwDateFormatString	1	Format dates as strings

Examples

Use this list to find examples in the documentation.

Calling a COM Object in a C++ Program

"Calling a COM Object in a C++ Program" on page 3-12

Passing Arguments

"Creating and Using a varargin Array in Visual Basic Programs" on page 3-21
"Creating and using varargout in Visual Basic programs" on page 3-22
"Using Array Formatting Flags" on page 3-25
"Using Data Conversion Flags" on page 3-26

Using MATLAB Global Variables

"Using MATLAB Global Variables in Visual Basic" on page 3-30

Querying the Registry

"Obtaining Registry Information" on page 3-33

Basic Usage Example: Visual Basic

"Magic Square Example" on page 4-2

Creating a Comprehensive Excel Add-in

"Spectral Analysis Example" on page 4-11

Comprehensive Examples

"Univariate Interpolation" on page 4-27
"Matrix Calculator" on page 4-39

Index

A

access 3-3
array formatting flags 3-23

C

capabilities 6-2
.cbl file 1-5
class method
  calling 3-6
Class MWFlags 8-10
Class MWUtil 8-3
class name 1-10
  changing default 1-4
class properties
  properties, class 3-30
COM
  defined 1-10
COM class
  producing 6-22
COM `VARIANT` 6-8
`comltool`
  purpose 2-2
command line interface 1-12
component
  access 3-3
component name 1-4
Component Object Model (COM)
  defined 1-10
`componentinfo` function 7-2
`comtool` function 7-6
`CreateObject` function 3-6

D

data conversion flags 3-23
data conversion rules 6-8
DLL
  Dynamic Link Library 1-4
Dynamic Link Library 1-4

E

Enumeration
  `mwArrayFormat` 8-31
  `mwDataType` 8-31
  `mwDateFormat` 8-32
enumerations 8-31
errors
  COM Builder 5-2
examples
  magic square 4-2
  spectral analysis 4-11

F

flags
  array formatting 3-23
  data conversion 3-23

G

global variables 3-30
Globally Unique Identifier (GUID) 6-5
GUID (Globally Unique Identifier) 6-5

I

IDL mapping 6-22

L

limitations 1-15

M

magic square example 4-2
MCR
  MATLAB Component Runtime 1-5
methods 1-10
missing parameter 8-5
`MWFlags` class 8-10
`mwregsvr` utility 6-4

MWUtil class 8-3

N

New operator 3-7

P

project
  creating 1-3
  elements of 1-10
  settings 2-6
Project
  directory 1-5
  version number 1-4

R

requirements
  system 1-15
restrictions 1-15

S

self-registering component 6-4
singleton MCR option 1-5
spectral analysis example 4-11
system requirements 1-15

T

troubleshooting 5-2
type library 6-4

U

unregistering components 6-4
utility library 8-3

V

VARIANT variable 6-8
version number 1-10 6-6
versioning 1-10
versioning rules 6-7
Visual Basic mapping 6-24