MATLAB® Excel Builder

The Language of Technical Computing

Computation

Visualization

Programming

The MathWorks

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

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

MATLAB Excel Builder User's Guide

Printing History:	December 2001	Online only	New for Version 1.0
	July 2002	First printing	Version 1.1 (Release 13)

Contents

Preface

1 Overview

2 Graphical User Interface

3 Programming with Excel Builder Components

4 Usage Examples

5 Function Wizard

C Registration and Versioning

D Utility Library

E Troubleshooting

Preface

What Is MATLAB Excel Builder?

MATLAB® Excel Builder provides the capability to incorporate seamlessly and quickly MATLAB models and functions into Excel worksheets. The graphical user interface enables you to build and deploy Excel add-ins containing functionality designed in MATLAB but accessed from the Excel environment.

Suggested Background

Users of this product need to be familiar with

- MATLAB and the MATLAB Compiler
- Microsoft Excel
- Visual Basic for Applications (VBA)

It is helpful to have some background in Component Object Model (COM) objects (DLLs).

See the documentation provided by the vendors for detailed information.

Requirements for MATLAB Excel Builder

System Requirements

System requirements and restrictions on use for Excel Builder are almost identical to those listed in the MATLAB Compiler documentation. For specific information see the "System Requirements" section under "Microsoft Windows on PCs" in your Compiler document.

Compiler Requirements

Because not all compilers are capable of producing Microsoft-compatible COM objects, Excel Builder supports only these compiler choices:

- Borland C++ Builder 4
- Borland C++ Builder 5
- Borland C++ Builder 6
- Microsoft Visual Studio 5.0
- Microsoft Visual Studio 6.0
- Microsoft Visual Studio .NET

After installing Excel Builder, you must run the MATLAB Compiler `mbuild` tool with the `-setup` argument. You can find information about `mbuild` in the MATLAB Compiler documentation in the section "Building Stand-Alone Applications on PCs." You also need to run the MATLAB command `mccsavepath` one time to set up the MATLAB Compiler search path. Type `help mccsavepath` at the MATLAB command line for a description of this command.

Excel Requirements

There is no specific requirement to use any particular version of Excel. However, in building an Excel add-in file (`.xla`), it is important that you build on the same version of Excel that you intend to distribute to. For example, if you intend to distribute to Excel 2000, you must build on Excel 2000. An add-in built on Excel 97, for example, will not work with Excel 2000.

Limitations and Restrictions

In general, limitations and restrictions on the use of MATLAB Excel Builder are the same as those for the MATLAB Compiler. See the “Limitations and Restrictions” section of the MATLAB Compiler documentation for details. Note that although the Compiler supports some usage of the MATLAB `input` command, MATLAB Excel Builder does not support this command at all.

Upgrading from a Previous Release

If you have used MATLAB Excel Builder with a previous MATLAB release, you must rebuild and redeploy all components with MATLAB Release 13.

Related Products

The MathWorks provides several products relevant to the tasks you can perform with MATLAB Excel Builder.

For more information about any of these products, see either:

- The online documentation for that product if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the “products” section

---

Note The toolboxes listed below all include functions that extend the capabilities of MATLAB.

Product	Description
MATLAB Compiler	Convert MATLAB M-files to C and C++ code
MATLAB Runtime Server	Deploy run-time versions of MATLAB applications
MATLAB Web Server	Use MATLAB with HTML Web applications

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names, syntax, filenames, directory/folder names, and user input	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Buttons and keys	**Boldface** with book title caps	Press the **Enter** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'`**`whole`**`')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.

MATLAB output	`Monospace font`	MATLAB responds with
		`A =`
		`5`

Menu and dialog box titles	**Boldface** with book title caps	Choose the **File Options** menu.
New terms and for emphasis	*Italics*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'`*`method`*`')`

1

Overview

Building a Deployable Application

Using MATLAB Excel Builder to create a deployable application is a simple process requiring a sequence of six steps. For details see

- "Creating a Project" on page 1-3
- "Managing M-Files and MEX-Files" on page 1-6
- "Building a Project" on page 1-8
- "Testing the Model" on page 1-8
- "Application Deployment" on page 1-9
- "Packaging and Distributing the Component" on page 1-9

This section references various menus provided by the Excel Builder graphical user interface (GUI). For a full discussion of these menus, see Chapter 2, "Graphical User Interface."

Elements of an Excel Builder Project

A project consists of all the elements necessary to build a deployable application using the MATLAB Excel Builder. Excel Builder *components* are COM objects accessible from Microsoft Excel through Visual Basic for Applications. COM is an acronym for Component Object Model, which is Microsoft's binary standard for object interoperability. Each COM object exposes a *class* to the Visual Basic programming environment. The class contains a set of functions called methods, corresponding to the original MATLAB functions included in the component's project.

Note Currently, MATLAB Excel Builder components support one class per component.

Classes

When creating a component, you must additionally provide a class name. The component name represents the name of the DLL file to be created. The class name denotes the name of the class that performs a call on a specific method at run-time. The relationship between component name and class name, and which methods (MATLAB functions) go into a particular class, are purely organizational. As a general rule, when compiling many MATLAB functions, it helps to determine a scheme of function categories and to create a separate class for each category. The name of each class should be descriptive of what the class does. Organizing related functions into classes in this way has the added advantage of reducing the amount of code to rebuild and redeploy when one function is changed.

Versions

MATLAB Excel Builder components also support a simple versioning mechanism. A version number is attached to a given component. This number gets automatically built into the DLL file name and the system registry information. As a general rule, the first version of a component is 1.0 (the default value if none is chosen). Changes made to the component before deployment keep the same version number. After deployment, change the version number for all subsequent changes, so that you can easily manage the new and old versions. The system sees classes in different versions of the same component as distinct, even if they have the same name.

Creating a Project

To begin project creation, enter the MATLAB command `mxltool` at the command line. The **MATLAB Excel Builder** main window appears.

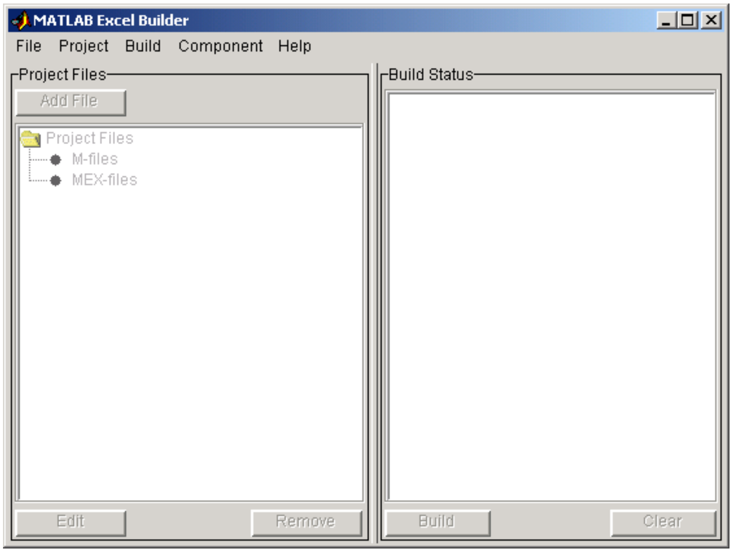


Figure 1-1: MATLAB Excel Builder Main Window

For a complete description of the features available from this window, see "Graphical User Interface Menus" on page 2-2.

Select **File -> New Project** on this window to view the **New Project Settings** dialog box.

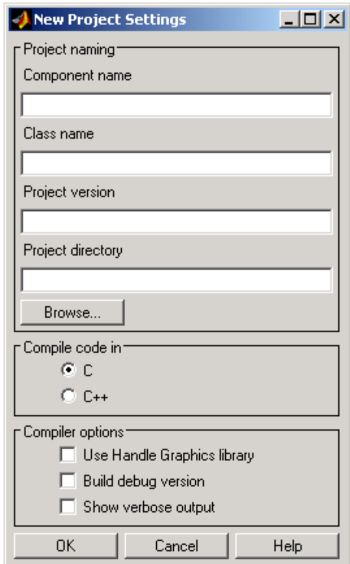


Figure 1-2: New Project Settings Dialog Box

Component name denotes the name of the DLL created later in the build process. After you enter the component name, the GUI automatically enters a **Class name** identical to the component name. You can change the class name to something more descriptive. Although the component name and class name can be the same, the component name cannot match the name of any M- or MEX-files added to the project later.

The **Project version** default value is 1.0. See "Versions" on page 1-3 for additional information about **Project version**.

Project directory specifies where any project and build files are written when compiling and packaging your models. The project directory is automatically generated from the name of your current directory and the component name.

Note You can accept the automatically generated project directory path or choose another of your liking. Once you click **OK** on this menu, this path is saved. If you later decide to move the project or change anything on its path, you need to redo the entire project specification process, including adding files to the project (see "Project Settings" on page 2-6) and respecifying the project directory path.

You can choose to generate C or C++ code. Components written in C give better performance, while C++ components are more readable, allowing easier modification of the generated code if needed. The files generated pertaining to the COM interface are always C++ files regardless of which option you choose.

If your models contain MATLAB Handle Graphics® calls, include the MATLAB C/C++ Graphics library in your project by selecting **Use Handle Graphics library**.

You can also create a debug version of your compiled models and can specify verbose output when you invoke the MATLAB Compiler. A debug version of your component:

- Enables backtraces so that any reported error shows the M-file and line where the error occurred. The full backtrace is reported. Without debugging you get the error without any indication of where it came from in your MATLAB code.
- Allows full debugging using the Visual Studio debugger.

Once you accept these settings on the **New Project Settings** dialog box by clicking **OK**, they become part of your project workspace and are saved to the project file along with the names of any M- or MEX- files you subsequently add to the project. A project file of the name `<component_name>.mxl` is automatically saved to the project directory.

Managing M-Files and MEX-Files

After you create a project, you enable the **Project**, **Build** and **Component** menu options on the **MATLAB Excel Builder** main window.

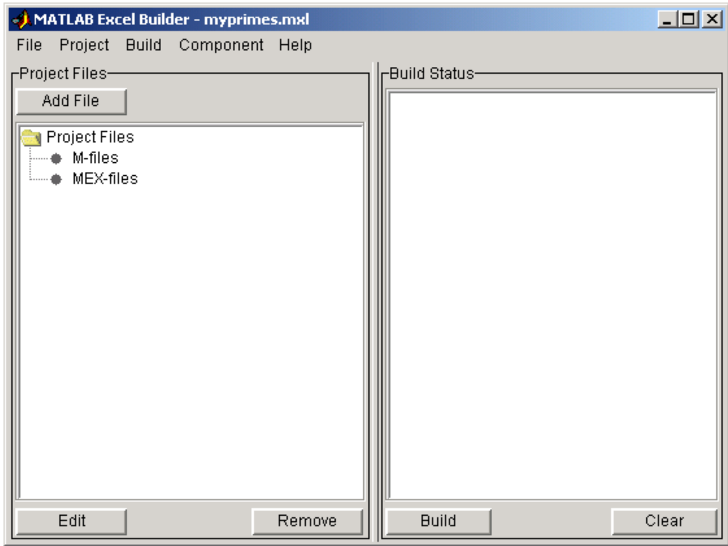


Figure 1-3: Main Window with Options Activated

Add M- and/or MEX-files to the project by clicking the **Add File** button or selecting the **Project -> Add File...** menu choice. You can add only a single file at a time to the project.

Note The name of any file added to the project cannot duplicate the name of any function existing in the library of precompiled functions.

The **Remove** button or **Project -> Remove File** menu choice removes any selected M- or MEX-files. You can highlight multiple files for removal at one time.

The **Edit** button, the **Project -> Edit File...** or double-clicking an M-file name opens the selected M-file(s) in the MATLAB editor for modification or debugging. You cannot edit MEX files.

Building a Project

After you define your project settings and add the desired M- and MEX-functions, you can build a deployable DLL and the necessary Visual Basic for Applications (VBA) code that allows Excel to access the DLL. Choosing **Build -> EXCEL/COM Files** or clicking the **Build** button invokes the MATLAB Compiler, writing the intermediate source files to <project_dir>\src and the output files necessary for deployment to <project_dir>\distrib.

Build Status

The **Build status** panel shows the output of the build process and informs you of any problems encountered. The files appearing in the <project_dir>\distrib directory will be a DLL and a VBA file (.bas). The resulting DLL is automatically registered on your system.

To clear the **Build status** panel, select **Build -> Clear Status**. The output of the build process is saved in the file <project_dir>\build.log. To open the Build Log, choose **Build -> Open Build Log**. The Build Log provides a record of the build process that you can refer to after you have cleared the **Build status** panel. If you ever contact MathWorks Technical Support with a question about the build process, you will be asked to provide a copy of this log.

Testing the Model

At this point, you can test the model by importing the VBA file (.bas) into the Excel Visual Basic editor and invoking one of the functions from the Excel worksheet. To import the VBA code into Excel's Visual Basic editor, open Excel and choose **Tools -> Macros -> Visual Basic Editor**. From the Visual Basic editor, choose **File -> Import** and select the created VBA file from the <project_dir>\distrib directory.

The Visual Basic module created when you build the project contains the necessary initialization code and a VBA formula function for each MATLAB function processed. Each supplied formula function essentially wraps a call to the respective compiled function in a format that can be accessed from a cell in an Excel worksheet. This function takes a list of inputs corresponding to the inputs of the original MATLAB function and returns a single output corresponding to the first output argument. Formula functions of this type are most useful to access a function of one or more inputs that returns a single scalar value. When you require multiple outputs or outputs representing ranges of data, you need a more general Visual Basic subroutine. For a more detailed discussion on integrating MATLAB Excel Builder components into Microsoft Excel via Visual Basic for Applications, see Chapter 3, "Programming with Excel Builder Components."

Application Deployment

Now create an Excel add-in (`.xla`) from your VBA code. Return to the Excel worksheet window and save the file as an `.xla` file to the `<project_dir>\distrib` directory.

Here are the steps necessary to create an Excel add-in from the generated VBA code. If these steps do not work, refer to your Excel documentation on creating a `.xla` file:

1. Start Excel.
2. Choose **Tools -> Macros -> Visual Basic Editor**.
3. In the **Microsoft Visual Basic** window, choose **File -> Import**.
4. Select VBA file (`.bas`) from the `<projectdir>distrib` directory.
5. Close the Visual Basic Editor.
6. In the Excel worksheet window, choose **File -> Save As...** .
7. .Set the **Save as** type to Microsoft Excel add-in (`*.xla`).
8. Save the `.xla` file to `<projectdir>\distrib`.

You can also deploy files in `*.xls` and `*.bas` formats. To deploy in `*.xls` format, follow the steps above but change the **Save as** type in Step 7 to `*.xls`. To deploy as VBA code, follow Steps 1 - 4 above only.

Packaging and Distributing the Component

Once you have successfully compiled your models and created the Excel add-in, you are ready to package the component for distribution to your end users.

Choose **Component -> Package Component** to create a self-extracting executable containing these files.

File	Purpose
`_install.bat`	Script run by the self-extracting executable
`<componentname_projectversion>.dll`	Compiled component
`mglinstaller.exe`	MATLAB math and graphics installer
`mwcomutil.dll`	Excel Builder utility library
`mwregsvr.exe`	Executable that registers DLLs on target machines
`*.xla`	Any Excel add-in files found in the `<projectdir>\distrib` directory

The self-extracting executable is named `<componentname>.exe`.

Running the installer on a target machine performs these steps:

- `mglinstaller` installs the MATLAB C/C++ Math and Graphics libraries.

  Action: Add the `<application>\bin\win32` directory that `mglinstaller` creates to your path. (`<application>` represents the deployed application's root directory, the directory where the deployed application resides on your system.)
- `mwregsvr` registers `mwcomutil.dll` and `<componentname>_<projectversion>.dll`.
- `mglinstaller` writes any Excel add-ins (`*.xla`) to the current directory location.

  Action: To use the Excel add-ins, start Excel, choose **Tools -> Add-Ins**, and select the desired `.xla` file.

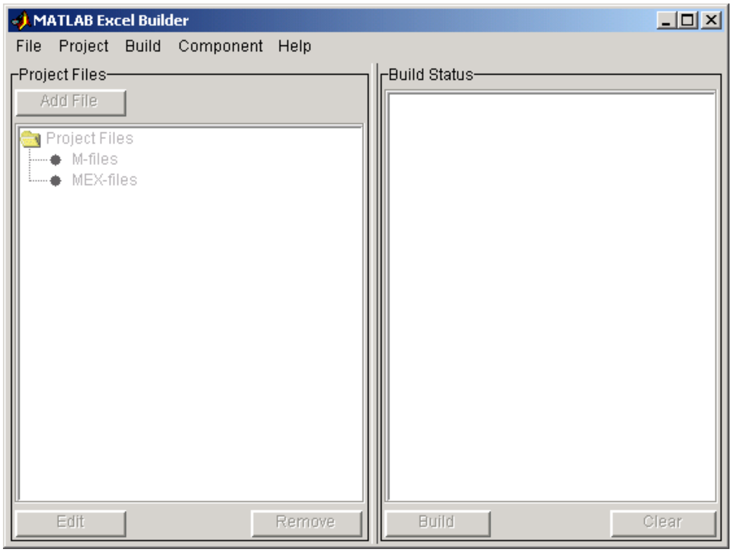
You must repeat this distribution process on each target machine.

2

Graphical User Interface

Graphical User Interface Menus

The MATLAB function `mxltool` displays the MATLAB Excel Builder graphical user interface (GUI) main window.

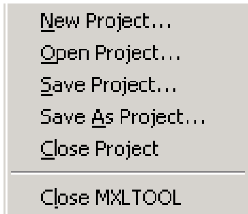


The information below describes the use of the various menus that the main window provides. These menus are:

- "File Menu" on page 2-3
- "Project Menu" on page 2-3
- "Build Menu" on page 2-4
- "Component Menu" on page 2-4
- "Help Menu" on page 2-5

File Menu

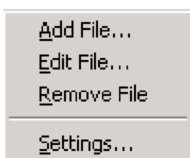
The **File** menu creates and manages MATLAB Excel Builder projects.



- **New Project** opens the project settings dialog box. This menu item creates a project workspace where you can add M- and MEX-files to the project and store project settings.
- **Open Project** allows you to load a previously saved project.
- **Save Project** saves the current project. If you have not yet saved the current project, you are prompted for a filename.
- **Save As Project** saves the current project after prompting for a filename.
- **Close Project** closes the current project.
- **Close MXLTOOL** closes the Excel Builder interface.

Project Menu

The **Project** menu controls the management of the current project's files.

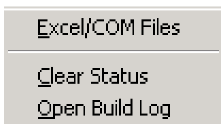


- **Add File** adds an M-file or MEX-file to the current project. (The **Add File** button in the **Project files** frame of the main window performs the same task).
- **Edit File** allows you to edit the selected M-file. (The **Edit** button in the **Project files** frame of the main window performs the same task.)

- **Remove File** removes the currently selected files from the project. (The **Remove** button in the **Project files** frame of the main window performs the same task.)
- **Settings** opens the project settings dialog box showing the current project's information. See "Project Settings" on page 2-6 for details.

Build Menu

The **Build** menu controls the building of the project's files into an Excel-accessible COM object.



- **Excel/COM Files** builds project files into an Excel-accessible COM object and generates Visual Basic Application code necessary to create an Excel add-in. The Excel add-in adds the new function(s) to the Excel function name space.
- **Clear Status** clears the **Build status** window.
- **Open Build Log** displays project status that has been saved in this log file.

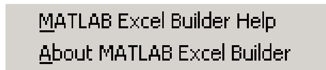
Component Menu

The **Component** menu completes the process of building a deployable application.

- **Package Component** readies files for deployment. The deployable files are packaged in a self-extracting executable.
- **Component Info** displays a dialog box with information about the current project's component and component versions. See "Component Information" on page 2-7 for details.

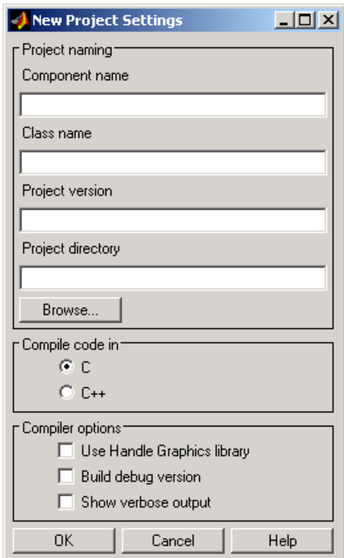
Help Menu

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

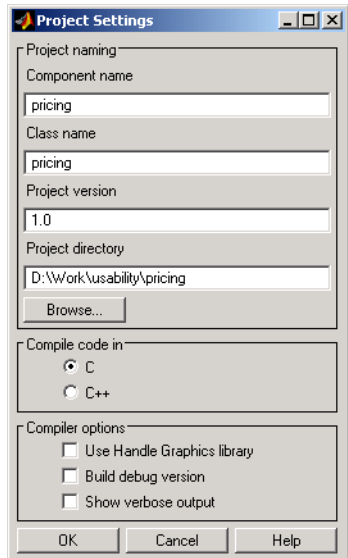


Project Settings

Choosing **New Project** or **Open Project** from the **File** menu or **Settings** from the **Project** menu opens the appropriate **Project Settings** dialog box.



New Project Settings



Existing Project Settings

See "Versioning" on page C-4 for a description of **Component name**, **Class name** and **Project version**. **Project directory** is the location of any project output files.

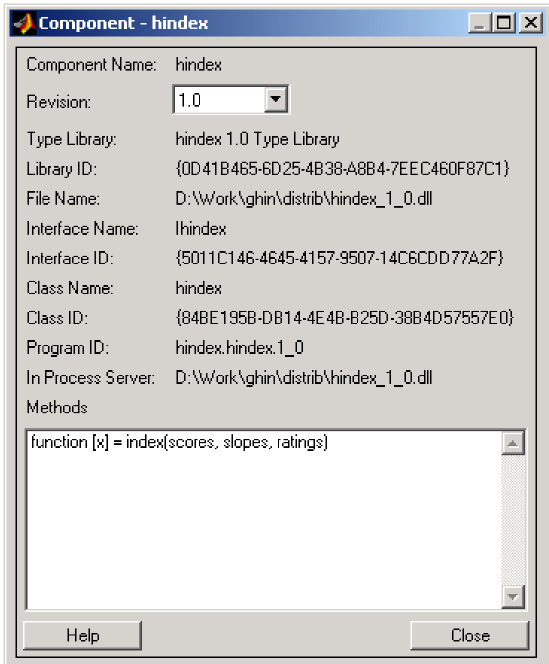
You can choose to generate C or C++ code. Components written in C give better performance, while C++ components are more readable, making it easier for you to modify the generated code if needed.

If your models contain MATLAB Handle Graphics calls, select **Use Handle Graphics library**.

You can create a debug version of your compiled models and can specify verbose output when you invoke the MATLAB Compiler.

Component Information

The **Component Info** choice under the **Component** menu displays the **Component** dialog box.



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

See Table C-2, Registry Information Returned by componentinfo, on page C-7, for an explanation of these fields. The **Methods** listbox shows the name and M-file calling syntax of each function within the component.

3

Programming with Excel Builder Components

Overview

Each MATLAB Excel Builder component is built as a stand-alone COM object. You access a component from Microsoft Excel through Visual Basic for Applications (VBA). This section provides general information on how to integrate MATLAB Excel Builder components into Excel using the VBA programming environment. It assumes that you have a working knowledge of VBA and is not intended to be a discussion on how to program in Visual Basic. Refer to the VBA documentation provided with Excel for general programming information.

You can easily integrate MATLAB Excel Builder components into a VBA project by creating a simple code module with functions and/or subroutines that load the necessary components, call methods as needed, and process any errors. In general, you need to address seven items in any code written to use MATLAB Excel Builder components:

- "When to Use a Formula Function or a Subroutine" on page 3-3
- "Initializing Excel Builder Libraries with Excel" on page 3-4
- "Creating an Instance of a Class" on page 3-6
- "Calling the Methods of a Class Instance" on page 3-9
- "Processing varargin and varargout Arguments" on page 3-11
- "Handling Errors During a Method Call" on page 3-13
- "Modifying Flags" on page 3-14

Note All code samples in this section are for illustration purposes and reference a hypothetical class named `myclass` contained in a component named `mycomponent` with a version number of 1.0. See Chapter 4, "Usage Examples" for a list of working code samples.

When to Use a Formula Function or a Subroutine

Visual Basic for Applications (VBA) provides two basic procedure types, functions and subroutines. You access a VBA function directly from a cell in a worksheet as a formula function and access a subroutine as a general macro. Function procedures are useful when the original MATLAB function takes one or more inputs and returns one scalar output. When the original MATLAB function returns an array of values or multiple outputs, you need a subroutine procedure to map these outputs into multiple cells/ranges in the worksheet. When you create a MATLAB Excel Builder component, you produce a VBA module (`.bas` file). This file contains simple call wrappers, each implemented as a function procedure for each method of the class.

Initializing Excel Builder Libraries with Excel

Before you use any MATLAB Excel Builder component, initialize the supporting libraries with the current instance of Excel. Do this once for an Excel session that uses MATLAB Excel Builder components. To do this initialization, call the utility library function `MWInitApplication`, a member of the `MWUtil` class. This class is part of the `MWComUtil` library. See the section "Utility Library Classes" on page D-3 for a detailed discussion of the functionality provided with this library.

One way to add this initialization code into a VBA module is to provide a subroutine that does the initialization once, and simply exits for all subsequent calls. The following Visual Basic code sample initializes the libraries 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 reinitializing.

```
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
```

This code is similar to the default initialization code generated in the VBA module created when the component is built. Each function that uses MATLAB Excel Builder components can include a call to `InitModule` at the beginning to ensure that the initialization always gets performed as needed.

Creating an Instance of a Class

Before calling a class method (compiled MATLAB function), you must create an instance of the class that contains the method. VBA provides two techniques for doing this:

- `CreateObject` function
- Visual Basic `New` operator

CreateObject Function

This method uses the Visual Basic application program interface (API) `CreateObject` function to create an instance of the class. To use this method, dimension a variable of type `Object` to hold a reference to the class instance and call `CreateObject` with the class' programmatic identifier (`ProgID`) as an argument as shown in the next 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. Before using this method, you must reference the type library containing the class in the current VBA project. Do this by selecting the **Tools** menu from the Visual Basic editor, and then selecting **References...** to display the **Available References** list. From this list select the necessary type library.

The following example illustrates using the `New` operator to create a class instance. It assumes that you have selected **mycomponent 1.0 Type Library** from the **Available References** list before calling this function.

```
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`.

Both methods are equivalent in functionality. The first method does not require a reference to the type library in the VBA project, while the second results in faster code execution. The second method has the added advantage of enabling the **Auto-List-Members** and **Auto-Quick-Info** capabilities of the VBA editor to work with your classes. The default function wrappers created with each built component all use the first method for object creation.

In the previous two examples, the class instance used to make the method call was a local variable of the procedure. This creates and destroys a new class instance for each call. An alternative approach is to declare one single module-scoped class instance that is reused by all function calls, as in the initialization code of the previous example.

The following example illustrates this technique with the second method.

```
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 have created a class instance, you can call the class methods to access the compiled MATLAB functions. MATLAB Excel Builder applies a standard mapping from the original MATLAB function syntax to the method's argument list. See section "Calling Conventions" on page A-7 for a detailed description of the mapping from MATLAB functions to COM class method calls.

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 compiled function and specifies how many outputs are requested. Methods that do not have output arguments do not pass a `nargout` argument. Following `nargout` are the output parameters listed in the same order as they appear on the left side of the original MATLAB function. Next come the input parameters listed in the same order as they appear on the right side of the original MATLAB function. All input and output arguments are typed as `Variant`, the default Visual Basic data type.

The `Variant` type can hold any of the basic VBA types, arrays of any type, and object references. Appendix B, "Data Conversion" describes in detail how to convert `Variants` 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 `UDTs`. You can also pass Excel `Range` objects directly as input and output arguments. 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.

The following examples illustrate the process of passing input and output parameters from VBA to MATLAB Excel Builder component class methods.

The first example is a formula function that takes two inputs and returns one output. This function dispatches the 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 same function as a subroutine and uses Excel ranges for input and output.

```
Sub foo(Rout As Range, Rin1 As Range, Rin2 As Range)
    Dim aClass As Object

    On Error Goto Handle_Error
    aClass = CreateObject("mycomponent.myclass.1_0")
    Call aClass.foo(1,Rout,Rin1,Rin2)
    Exit Sub
Handle_Error:
    MsgBox(Err.Description)
End Sub
```

Processing varargin and varargout Arguments

When varargin and/or varargout are present in the original MATLAB function, these parameters are added to the argument list of the class method as the last input/output parameters in the list. 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.

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

```
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
```

The MWUtil class included in the MWComUtil utility library provides the MWPack helper function to create varargin parameters. See Appendix D, “Utility Library” for more details.

The next example processes a varargout parameter into three separate Excel Ranges. This function makes use of the MWUnpack function in the utility library. The MATLAB function used is varargout = foo(x1,x2).

```
Sub foo(Rout1 As Range, Rout2 As Range, Rout3 As Range, _
        Rin1 As Range, Rin2 As Range)
   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,Rin1,Rin2)
   Call aUtil.MWUnpack(v,0,True,Rout1,Rout2,Rout3)
   Exit Sub
Handle_Error:
   MsgBox(Err.Description)
End Sub
```

Handling Errors During a Method Call

Errors that occur while creating a class instance or during a class method create an exception in the current procedure. 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 are handled this way, including errors within the original MATLAB code. An exception creates a Visual Basic `ErrObject` object in the current context in a variable called `Err`. (See the Visual Basic for Applications documentation for a detailed discussion on VBA error handling.) All of the examples in this section illustrate the typical error trapping logic used in function call wrappers for MATLAB Excel Builder components.

Modifying Flags

Each MATLAB Excel Builder component exposes a single read/write property named `MWFlags` of type `MWFlags`. The `MWFlags` property consists of two sets of constants: *array formatting flags* and *data conversion flags*. The data conversion flags change selected behaviors of the data conversion process from `Variants` to MATLAB types and vice versa. By default, MATLAB Excel Builder components allow setting data conversion flags at the class level through the `MWFlags` class property. This holds true for all Visual Basic types, with the exception of the Excel Builder `MWStruct`, `MWField`, `MWComplex`, `MWSparse`, and `MWArg` types. Each of these types exposes its own `MWFlags` property and ignores the properties of the class whose method is being called. The `MWArg` class is supplied specifically for the case when a particular argument needs different settings from the default class properties.

This section provides a general discussion of how to set these flags and what they do. See “Class MWFlags” on page D-9 for a detailed discussion of the `MWFlags` type, as well as additional code samples.

Array Formatting Flags

Array formatting flags guide the data conversion to produce either a MATLAB cell array or matrix from general `Variant` data on input or to produce an array of `Variants` or a single `Variant` containing an array of a basic type on output.

The following examples assume that you have referenced the `MWComUtil` library in the current project by selecting **Tools -> References...** and selecting **MWComUtil 1.0 Type Library** from the list.

```
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
```

Here, two `Variant` variables, `var1` and `var2` are constructed with the same numerical data, but internally they are structured differently. `var1` is a 2-by-2 array of `Variants` with each element containing a 1-by-1 `Double`, while `var2` is a 1-by-1 `Variant` containing a 2-by-2 array of `Doubles`. According to the default data conversion rules listed in Table B-3, COM VARIANT to MATLAB Conversion Rules, on page B-10, `var1` converts to a 2-by-2 cell array with each cell occupied by a 1-by-1 `double`, and `var2` converts directly to a 2-by-2 double matrix. The `InputArrayFormat` flag controls how arrays of these two types are handled. As it turns out, the two arrays in the previous example both convert to double matrices because the default value for the `InputArrayFormat` flag is `mwArrayFormatMatrix`. This default is used because, as it turns out, array data originating from Excel ranges is always in the form of an array of `Variants` (like `var1` of the previous example), and MATLAB functions most often deal with matrix arguments. But what if you really want a cell array? In this case, you set the `InputArrayFormat` flag to `mwArrayFormatCell`. Do this by adding the following line after creating the class and before the method call.

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

Setting this flag presents all array input to the compiled MATLAB function as cell arrays.

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.

`AutoResizeOutput` is used for Excel `Range` objects passed directly as output parameters. 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.

The `TransposeOutput` flag transposes all array output. This flag is useful when dealing with MATLAB functions that output one-dimensional arrays. By default, MATLAB realizes one-dimensional arrays as 1-by-n matrices (row vectors) that become rows in an Excel worksheet.

You may prefer worksheet columns from row vector output. This example auto-resizes and transposes an output range.

```
Sub foo(Rout As Range, Rin As Range )
   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.foo(1,Rout,Rin)
   Exit Sub
Handle_Error:
   MsgBox(Err.Description)
End Sub
```

Data Conversion Flags

Data conversion flags deal with type conversions of individual array elements. The two data conversion flags, `CoerceNumericToType` and `InputDateFormat`, govern how numeric and date types are converted from VBA to MATLAB. Consider the example

```
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
```

This example converts `var1` of type `Variant/Integer` to an `int16` and `var2` of type `Variant/Double` to a `double`. If the original MATLAB function expects `double`s 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. In such a case set the `CoerceNumericToType` flag to `mwTypeDouble`, causing the data converter to convert all numeric input to `double`. In the previous example, place the following line after creating the class and before calling the methods.

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

The `InputDateFormat` flag controls how the VBA `Date` type is converted. This 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 y1 = New MWArg
    y1.MWFlags.DataConversionFlags.OutputAsDate = True
    Call aClass.foo(2, ytemp, y2, today)
    y1 = ytemp
    Exit Sub
Handle_Error:
    MsgBox(Err.Description)
End Sub
```

4

Usage Examples

Magic Square Examples

The M-file `mymagic` takes a single input, an integer, and creates a magic square of that size.

The Excel file `mymagic.xls` uses this function in three different ways:

- The first illustration calls the function `mymagic` with a value of 4. The function returns a magic square of size 4 and populates a range of Excel cells with that magic square.
- The second illustration uses the transpose flag to transpose a magic square of size 4.
- The third illustration resizes the output to a higher value and moves its location within the Excel worksheet.

Note To get started, copy the distributed directory `xlmagic` from `<matlab>\toolbox\matlabxl\examples\xlmagic` to `<matlab>\work`.

Creating the Project

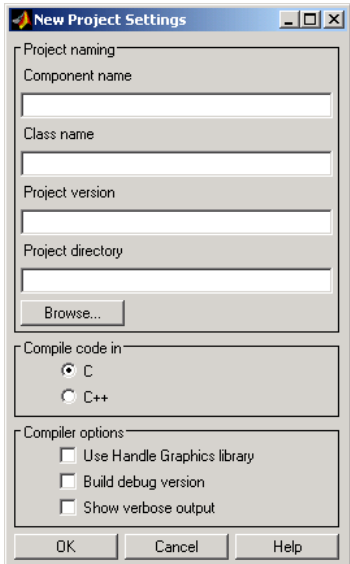


Figure 4-1: Empty New Project Settings Dialog Box

From the MATLAB command prompt change directories to `<matlab>\work`. Enter the command `mxltool` to start the MATLAB Excel Builder graphical user interface. From the **File** menu select **New Project**. This opens the **New Project Settings** dialog.

On the **New Project Settings** dialog, enter the settings as listed below.

- In the **Component name** text block enter the component name `xlmagic`. Press the **Tab** key to move to the **Class name** text block.
- This automatically fills in the **Class name** field with the name `xlmagic`. Leave this text in the **Class name** field.
- The version has a default of 1.0. Leave this version as is.
- The **Project directory** field contains a default of a combination of the directory where Excel Builder was started, `<matlab>\work`, and the **Component name**, `xlmagic`. 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.

- Select **C** as the code to compile in.
- Leave all **Compiler options** unselected.

The **New Project Settings** dialog now looks like Figure 4-2.

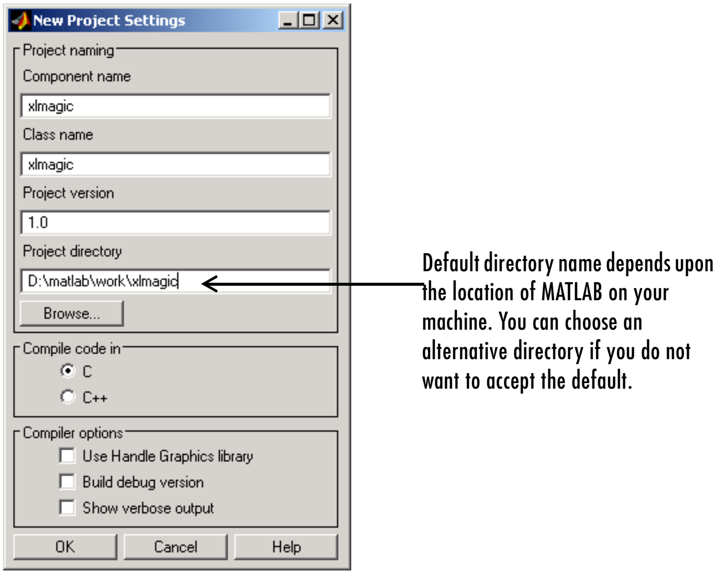


Figure 4-2: New Project Settings with Entries

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

Summary of Project Settings

Component name: `xlmagic`

Class name: `xlmagic`

Project version: 1.0

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

Compile code in: C

Compiler options: (*leave unselected*)

Use Handle Graphics library = No

Build debug version = No

Show verbose output = No

Building the Project

- From the Excel Builder graphical user interface click **Add File ...** .
- Select the file mymagic.m from the directory <matlab>\work\xlmagic and click **Open**.
- Click **Build** or select **Excel/COM Files** from the **Build** menu.

Adding the Excel Builder COM Function to Excel

- Start Excel on your system.
- Open the file <matlab>\work\xlmagic\mymagic.xls.

Note If you receive an Excel prompt informing you that this file contains macros, select the **Enable Macros** option to run this example.

Illustration 1. Output Magic Square Results to Excel

From the main Excel window (not the Visual Basic Editor), display the **Macro** dialog either by selecting the **Alt** and **F8** keys at the same time or by selecting the **Macros** option from **Tools -> Macro**.

Select mymagic from the list and click **Run**. This procedure returns a magic square of size 4 beginning in cell B2.

	A	B	C	D	E	F
1						
2	4	16	2	3	13	
3		5	11	10	8	
4		9	7	6	12	
5		4	14	15	1	
6						
7	The above example runs the macro "mymagic" which					
8	populates the cells B2 through E5 with a magic square of 4					
9	Select Tools->Macro-> Macros to run this example					

Figure 4-3: Magic Square Returned to Excel Worksheet

Illustration 2. Transpose the Output

Reopen the **Macro** dialog, select the `mymagic_transpose` macro and click the **Run** button. This procedure returns a magic square of size 4 transposed, beginning in cell B14.

13					
14	4	16	5	9	4
15		2	11	7	14
16		3	10	6	15
17		13	8	12	1
18					
19	The above example runs the macro "mymagic_transpose" which				
20	transposes the results of a magic square of 4 and populates the				
21	cells B14 through E17				
22	Select Tools->Macro-> Macros to run this example				

Figure 4-4: Transposed Magic Square

Illustration 3. Resize the Output

Reopen the **Macro** dialog, select the `mymagic_resize` macro, and click **Run**. This procedure returns a magic square of size 4 beginning in cell B32.

Change the value of 4 in cell A32 to a higher value and rerun this macro. A magic square of the size you specified in cell A32 is returned, beginning in cell B32.

27	The below example runs the macro "mymagic_resize" which								
28	has an initial range for a magic square of 4 but will resize if								
29	the output is larger. Gradually increase the number in cell A32 and rerun the macro.								
30	CAUTION: Resizing will over write any existing data in the target cells								
31									
32	8	64	2	3	61	60	6	7	57
33		9	55	54	12	13	51	50	16
34		17	47	46	20	21	43	42	24
35		40	26	27	37	36	30	31	33
36		32	34	35	29	28	38	39	25
37		41	23	22	44	45	19	18	48
38		49	15	14	52	53	11	10	56
39		8	58	59	5	4	62	63	1
40									

Figure 4-5: Resized Magic Square

Inspecting the Visual Basic Code

On the Excel main window select **Visual Basic Editor** from the **Tools -> Macro** menu.

From the Visual Basic Editor, in the **Project - VBAProject** window, double-click to expand the project **VBAProject (mymagic.xls)**.

Expand the `Modules` folder and double-click on the `Module1` module. This opens the **VB Code** window with the code for this project.

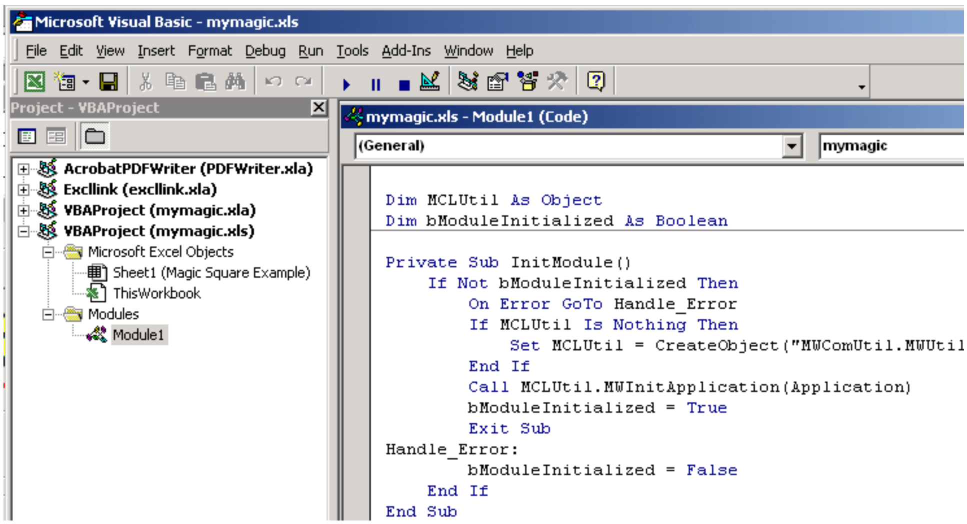


Figure 4-6: Visual Basic Code Window

Using Multiple Files and Variable Arguments

The M-file, `myplot`, takes a single integer input and plots a line from 1 to that number.

The M-file, `mysum`, takes an input of `varargin` of type `integer`, adds all the numbers, and returns the result.

The M-file, `myprimes`, takes a single integer input `n` and returns all the prime numbers less than or equal to `n`.

The Microsoft Excel file, `mymulti.xls`, demonstrates these functions in a multiple of ways.

Note To get started copy the distributed directory `xlmulti` from `<matlab>\toolbox\matlabxl\examples\xlmulti` to `<matlab>\work`.

Creating the Project

From the MATLAB command prompt, change directories to `<matlab>\work`. Enter the command `mxltool` to start the MATLAB Excel Builder graphical user interface. From the **File** menu select **New Project**. This opens the **New Project Settings** dialog box.

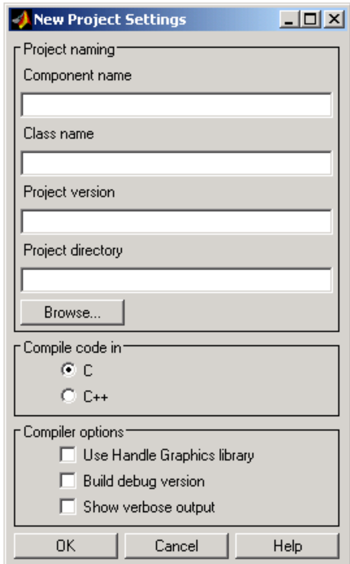


Figure 4-7: Empty New Project Settings Dialog

On the **New Project Settings** dialog, enter the settings as listed below.

- In the **Component name** text block enter the component name `xlmulti`. Press the **Tab** key to move to the **Class name** text block.
- This automatically fills in the **Class name** field with the name `xlmulti`. Leave this text in the **Class name** field.
- The version has a default of 1.0. Leave this version as is.
- The **Project directory** field contains a default of a combination of the directory where Excel Builder was started, `<matlab>\work`, and the **Component name**, `xlmulti`. 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.
- Select **C++** as the code to compile in.
- Under **Compiler options** check **Use Handle Graphics Library**. Leave all other **Compiler options** unchecked.

The **New Project Settings** dialog now looks like Figure 4-8.

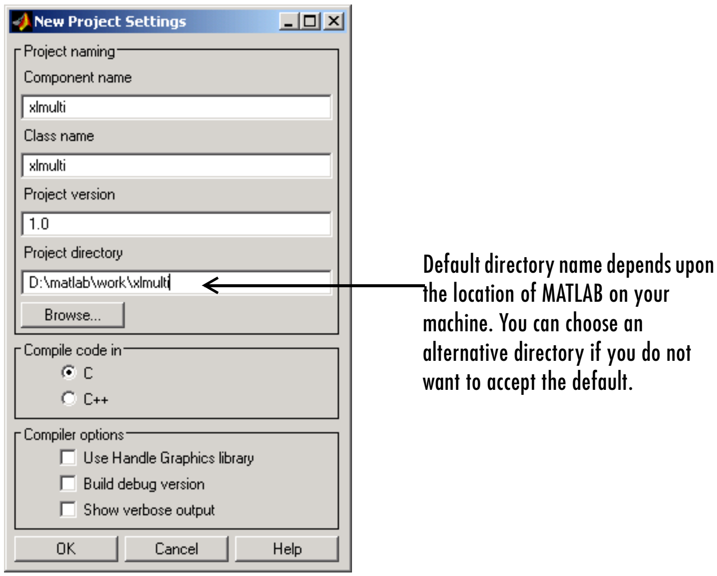


Figure 4-8: New Project Settings with Entries

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

Summary of Project Settings

Component name: `xlmulti`

Class name: `xlmulti`

Project version: 1.0

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

Compile code in: C++

Use Handle Graphics library = Yes

Build debug version = No

Show verbose output = No

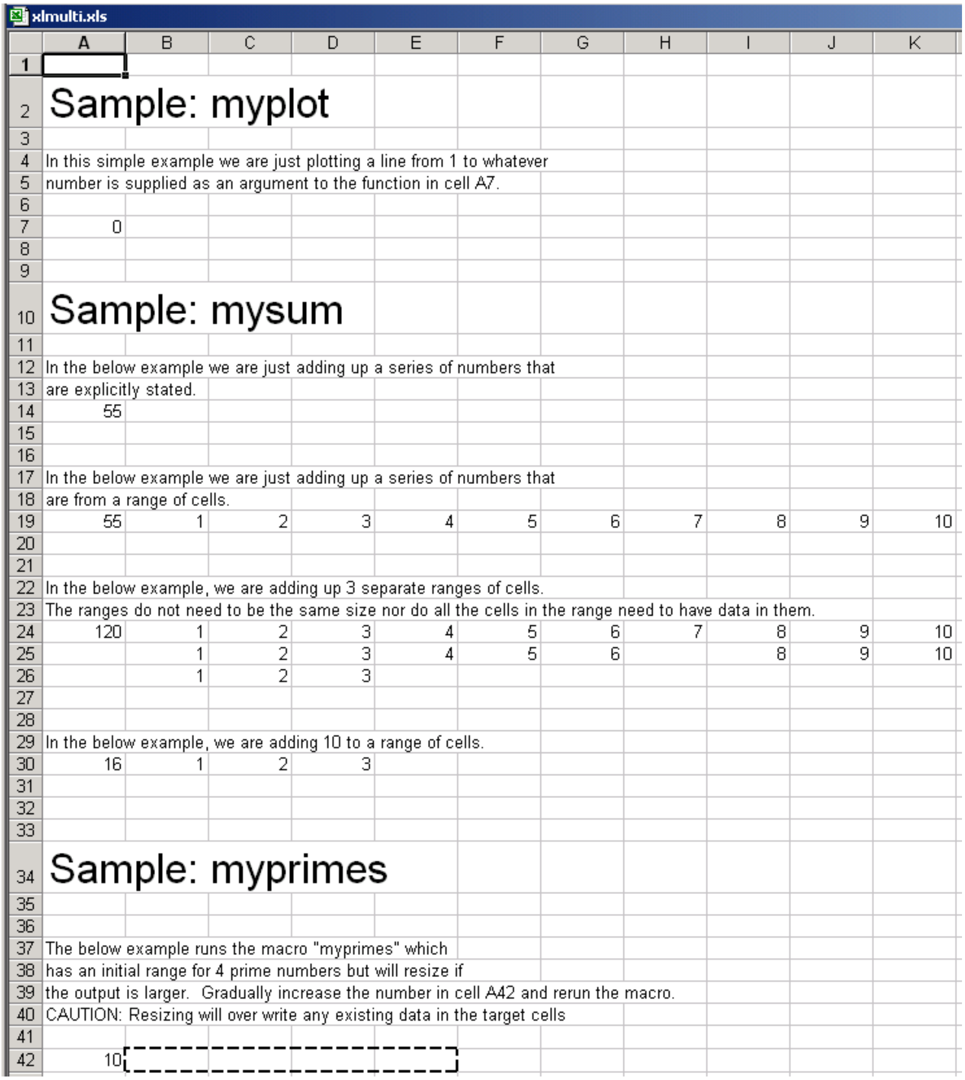
Building the Project

- From the Excel Builder graphical user interface click **Add File ...** .
- Select the file `myplot.m` from the directory `<matlab>\work\xlmulti` and click **Open**.
- Repeat the above steps to add the files `myprimes.m` and `mysum.m`.
- Click **Build** or select **Excel/COM Files** from the **Build** menu.

Adding the Excel Builder COM Functions to Excel

- Start Excel on your system.
- Open the file `<matlab>\work\xlmulti\mymulti.xls`.

Note If you receive an Excel prompt informing you that this file contains macros, select the **Enable Macros** option to run this example.


xlmulti.xls

	A	B	C	D	E	F	G	H	I	J	K
1											
2	Sample: myplot										
3											
4	In this simple example we are just plotting a line from 1 to whatever										
5	number is supplied as an argument to the function in cell A7.										
6											
7	0										
8											
9											
10	Sample: mysum										
11											
12	In the below example we are just adding up a series of numbers that										
13	are explicitly stated.										
14	55										
15											
16											
17	In the below example we are just adding up a series of numbers that										
18	are from a range of cells.										
19	55	1	2	3	4	5	6	7	8	9	10
20											
21											
22	In the below example, we are adding up 3 separate ranges of cells.										
23	The ranges do not need to be the same size nor do all the cells in the range need to have data in them.										
24	120	1	2	3	4	5	6	7	8	9	10
25		1	2	3	4	5	6		8	9	10
26		1	2	3							
27											
28											
29	In the below example, we are adding 10 to a range of cells.										
30	16	1	2	3							
31											
32											
33											
34	Sample: myprimes										
35											
36											
37	The below example runs the macro "myprimes" which										
38	has an initial range for 4 prime numbers but will resize if										
39	the output is larger. Gradually increase the number in cell A42 and rerun the macro.										
40	CAUTION: Resizing will over write any existing data in the target cells										
41											
42	10										

Figure 4-9: mymulti.xls

Illustration 4: Calling myplot

This illustration calls the function `myplot` with a value of 4. To execute the function, make A7 the active cell. Press **F2** and then **Enter**.

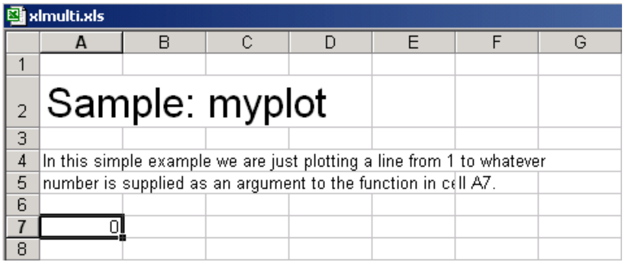


Figure 4-10: Calling myplot with a Value of 4

This procedure plots a line from 1 to 4 in a MATLAB figure window. This graphic can be manipulated as if it were called from MATLAB directly. The calling cell contains 0 because the function does not return a value.

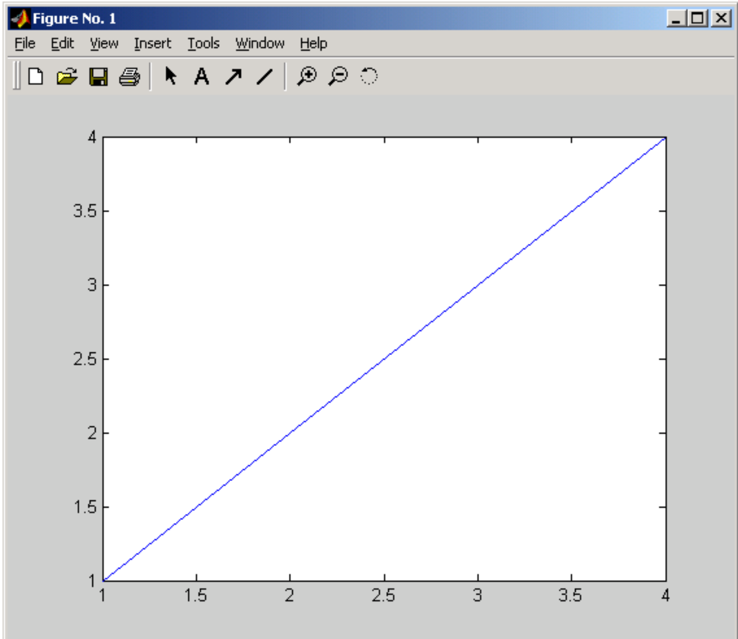


Figure 4-11: myplot Output

Illustration 5: Calling mysum Four Different Ways

This illustration calls the function `mysum` in four different ways. The first (cell A14) takes the values 1 through 10, adds them, and returns the result of 55. The second (cell A19) takes a range object that is a range of cells with the values 1 through 10, adds them, and returns the result of 55. The third (cell A24) takes several range objects, adds them, and returns the result of 120. This illustration demonstrates that the ranges do not need to be the same size and that all the cells do not have to have a value. The fourth (cell A30) takes a combination of a range object and explicitly stated values, adds them, and returns the result of 16.

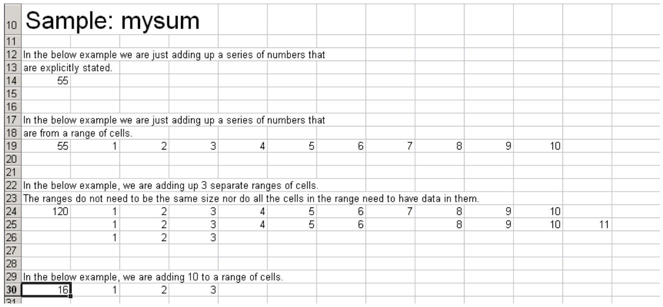


Figure 4-12: Four Different Calls to mysum

This illustration runs when the Excel file is opened. To reactivate the illustration, make the appropriate cell active. Then press **F2** followed by **Enter**.

Illustration 6: myprimes Macro

In this illustration the macro `myprimes` calls the function `myprimes.m` with an initial value of 10 in cell A42. The function returns all the prime numbers less than 10 to cells B42 through E42.

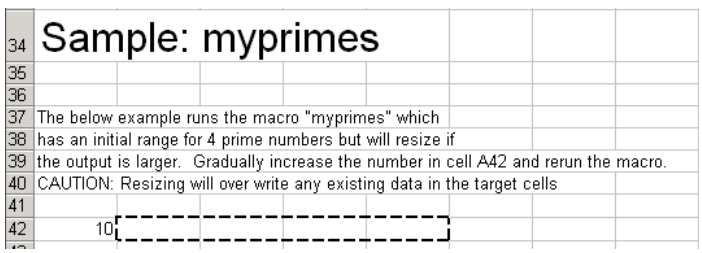


Figure 4-13: myprimes Macro

To execute the macro, from the main Excel window (not the Visual Basic Editor), display the **Macro** dialog either by selecting the **Alt** and **F8** keys at the same time or by selecting the **Macros** option from **Tools -> Macro**.

Select `myprimes` from the list and click **Run**.

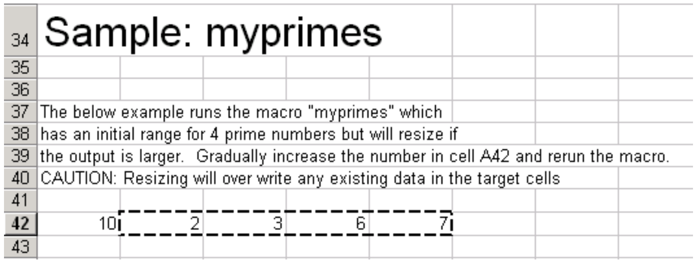


Figure 4-14: myprimes Output for Value of 10

This function automatically resizes if the returned output is larger than the output range specified. Change the value in cell A42 to a number larger than 10. Then rerun the macro. The output returns all prime numbers less than the number you entered in cell A42.

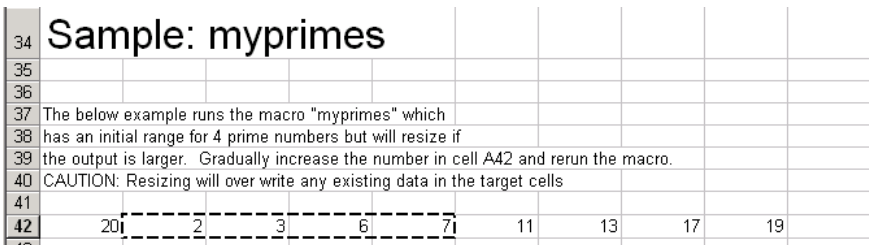


Figure 4-15: myprimes Output for Value > 10

Inspecting the Visual Basic Code

From Excel select **Visual Basic Editor** from the **Tools -> Macro** menu.

From the Visual Basic Editor, in the **Project - VBAProject** window, expand the project **VBAProject (mymulti.xls)**.

Expand the `Modules` folder and double click on the `Module1` module. This opens the **VB Code** window with the code for this project.

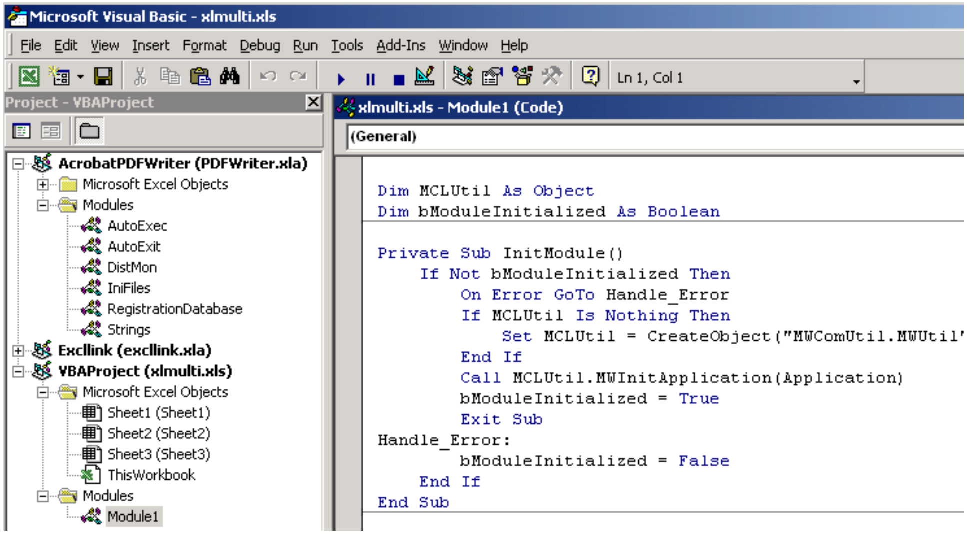


Figure 4-16: Visual Basic Code for mymulti.xls

Spectral Analysis Example

This example illustrates the creation of 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 VBA documentation for a complete discussion of these topics.

The example creates an Excel add-in that performs an 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 GUI.

To create this add-in requires four basic steps:

1. Build a standalone COM component from MATLAB code.
2. Implement the necessaryVBA 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`.

```
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 proceed with the actual building of the component, follow these steps:

1 Start `mxltool`. See “Graphical User Interface Menus” on page 2-2 for a discussion of using `mxltool` to build a COM component from a collection of MATLAB M-files.

2 Create a new project with these settings:

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

Check **Use Handle Graphics library**.

See "Project Settings" 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. Make note of the project directory because you will refer to it later when you save your add-in.

5 Click **Build** to create the component.

Integrating the Component with Visual Basic for Applications

Having built your component, you can implement the necessary VBA code to integrate it 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, select **Tools->Macro->Visual Basic Editor**.

3 When the Visual Basic Editor starts, select **Tools->References** to display the **Project References Dialog**. Check **Fourier 1.0 Type Library** and **MWComUtil 1.0 Type Library** on the list.

Creating the Main VB Code Module For the Application

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 on the **VBAProject** item in the project window and select **Insert->Module** from the pop-up menu.

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

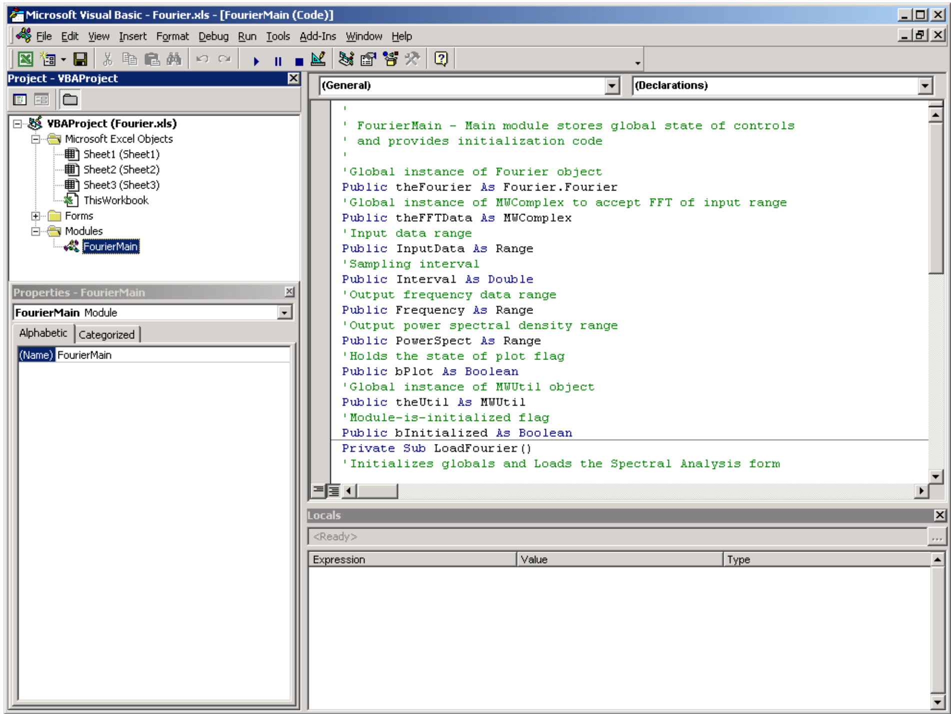


Figure 4-17: VBA Project: Insert->Module

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

```
'
' FourierMain - Main module stores global state of controls
' and provides initialization code
'
Public theFourier As Fourier.Fourier 'Global instance of Fourier object
Public theFFTData As MWComplex   'Global instance of MWComplex to accept FFT
Public InputData As Range         'Input data range
Public Interval As Double         'Sampling interval
Public Frequency As Range         'Output frequency data range
Public PowerSpect As Range        'Output power spectral density range
Public bPlot As Boolean           'Holds the state of plot flag
Public theUtil As MWUtil          'Global instance of MWUtil object
Public bInitialized As Boolean    'Module-is-initialized flag
```

```
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 theUtil Is Nothing Then
        Set theUtil = New MWUtil
        Call theUtil.MWInitApplication(Application)
    End If
    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 in the integration process develops a user interface for your add-in using the Visual Basic Editor. Follow the steps outlined here to create a new user form and populate it with the necessary controls.

1 Right-click on the **VBAProject** item in the project window and select **Insert->UserForm** from the pop-up menu.

2 A new form appears under `Forms` in the VBA Project. In the form's property page, set the name property to `frmFourier` and the `Caption` property to `Spectral Analysis`.

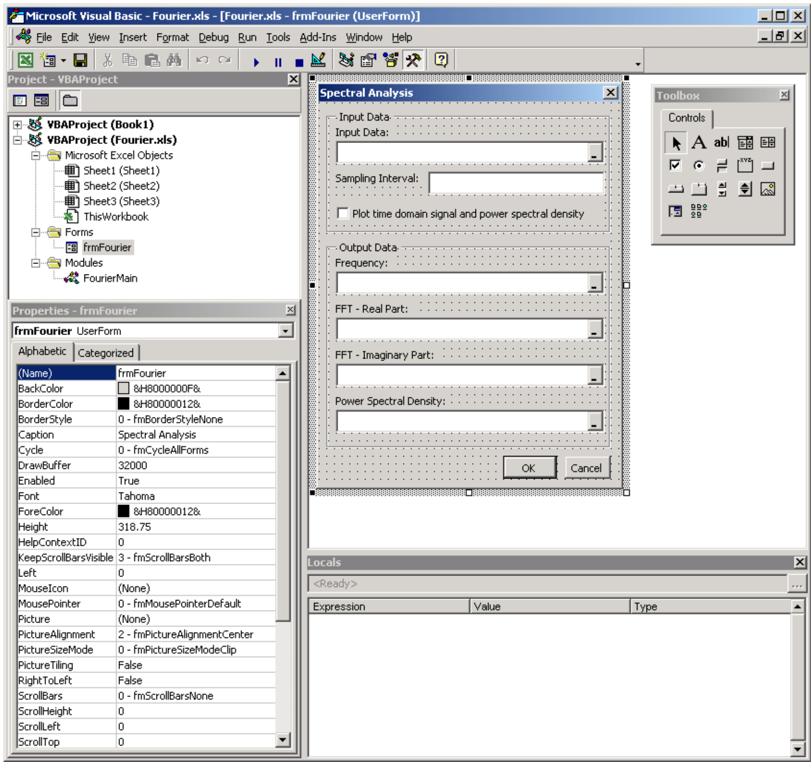


Figure 4-18: Creating the Visual Basic Form

3 Now 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.

Control Type	Control Name	Properties	Purpose
`RefEdit`	`refedtInput`		Selects range for input data.
`Label`	`Label2`	Caption = `Sampling Interval`	Labels the `TextBox` for 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.
`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.

Control Type	Control Name	Properties	Purpose
`CommandButton`	`btnOK`	Caption = `OK` Default = True	Executes the function and dismisses the dialog
`CommandButton`	`btnCancel`	Caption = `Cancel` Cancel = True	Dismisses the dialog without executing the function.

Figure 4-19, Layout of Controls on Main Form, on page 4-26 shows the controls layout on the form.

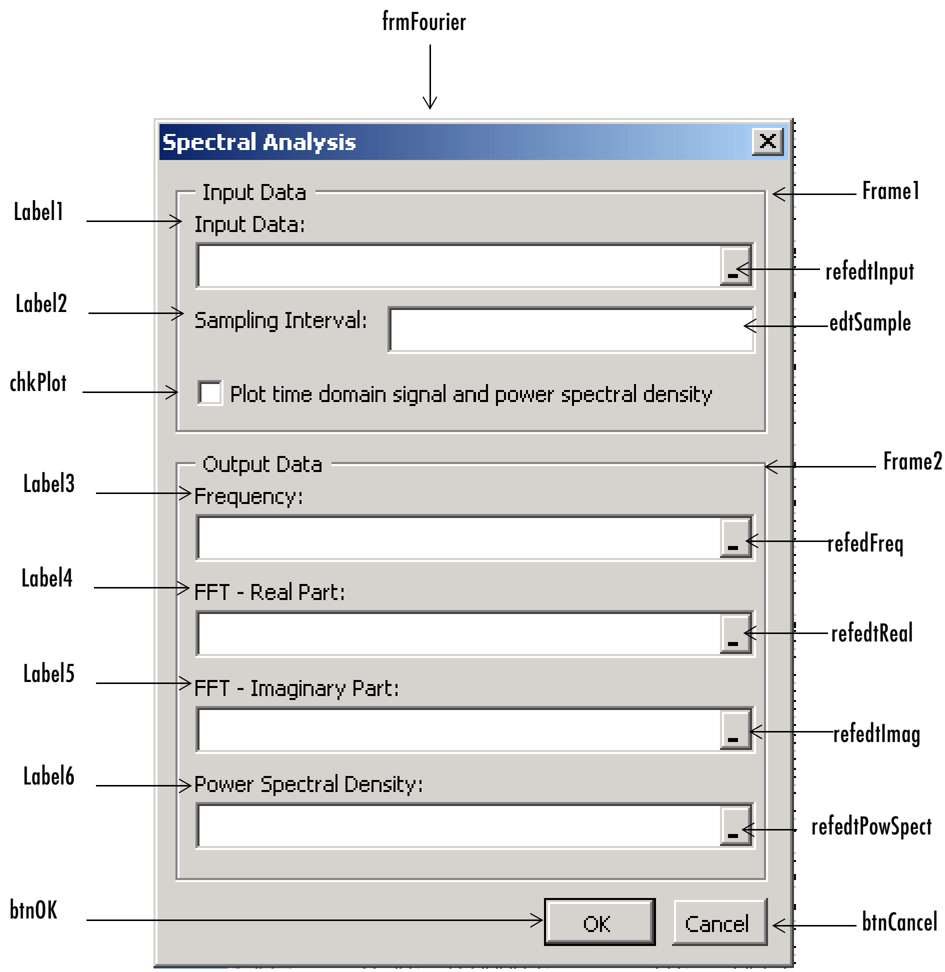


Figure 4-19: Layout of Controls on Main Form

When the form and controls are complete, right-click on the form and select **View Code** from the pop-up menu. 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 adds 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 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 on the **ThisWorkbook** item in the Visual Basic project window and select **View Code** from the pop-up menu. See the next figure.

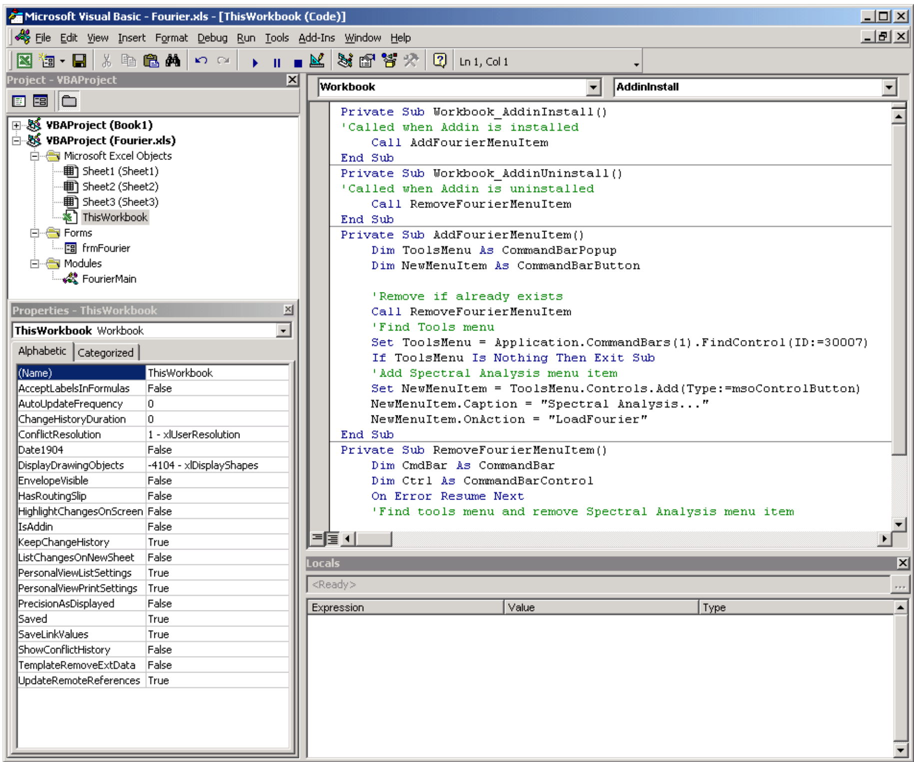


Figure 4-20: Adding a Menu Item to Excel

2 Place the code below into 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

Now that the Visual Basic coding is complete, you can save the add-in. Save this file into the `<project-directory>\distrib` directory that `mxltool` created when building the project. Here, `<project-directory>` refers to the project directory that `mxltool` used to save the Fourier project. Name the add-in `Spectral Analysis`.

Follow these steps to save the add-in.

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

2 When the **Workbook Properties** dialog appears, select the **Summary** tab and enter `Spectral Analysis` as the workbook title.

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

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

5 When the **Save As** dialog appears, select **Microsoft Excel Add-In (*.xla)** as the file type, and browse to `<project-directory>\distrib`.

6 Enter `Fourier.xla` as the file name and 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 Select **Tools->Add-Ins** from the main menu.

3 When the **Add-Ins** dialog comes up, click **Browse**.

4 Browse to the `<project-directory>\distrib` directory, select `Fourier.xla` and click **OK**.

5 The **Spectral Analysis** add-in appears in the available **Add-Ins** list and is checked.

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 selecting **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 sec. 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 Click and hold on the lower right hand corner of cell A2 and drag the formula 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()"`. 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 Select **Tools->Spectral Analysis...** from the main menu.

2 Click on the **Input Data** box.

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

4 Click on the **Sampling Interval** box and type 0.01.

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

6 Enter C1:C1001 for frequency output, and likewise 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 next figure shows the output.

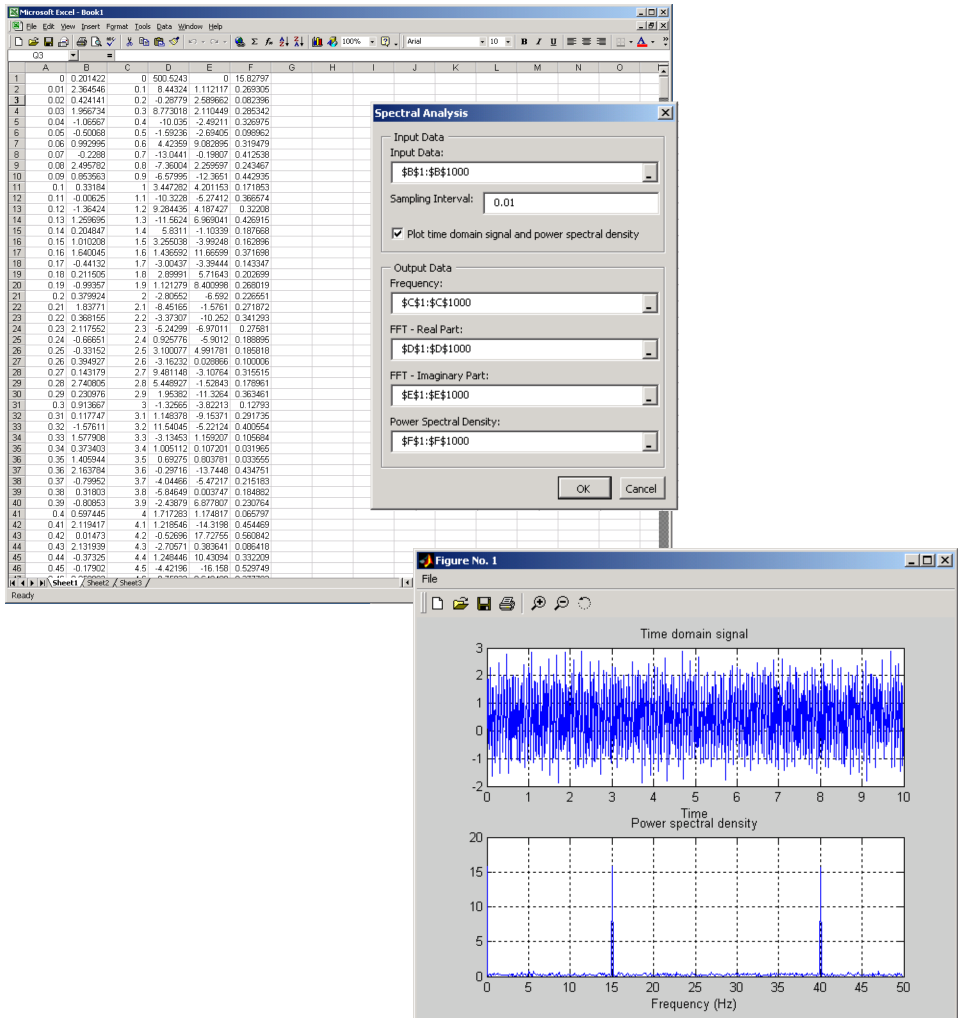


Figure 4-21: Worksheet with Inputs and Outputs for Test Problem

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

Package the Add-in

As a final step, package the add-in, the COM component, and all supporting libraries into a self-extracting executable. This package can now be installed onto other computers that need to use the Spectral Analysis add-in.

To package the add-in, follow these steps.

1 Return to `mxltool`. If `mxltool` has been dismissed, start it again and reload the Fourier project.

2 Select **Component->Package Component**.

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

5

Function Wizard

Introduction

The Function Wizard enables you to pass Microsoft Excel (Excel 200 or later) worksheet values to a compiled MATLAB model and to return model output to a cell or range of cells in the worksheet. The Function Wizard provides an intuitive interface to Excel worksheets. Knowledge of Visual Basic for Applications (VBA) programming is not required.

The Function Wizard reflects any changes that you make in the worksheets, such as range selections. Going in the opposite direction, you can use the Function Wizard to control the placement and output of data from MATLAB functions to the worksheets.

The Function Wizard does not currently support the MATLAB `struct`, `sparse`, and `complex` data types.

Installing the Function Wizard Add-in

The Function Wizard GUI is contained in an Excel add-in (`mlfunction.xla`) residing in the `<matlab>\toolbox\matlabxl\matlabxl` directory. You must install this add-in before using the Function Wizard.

Follow these steps to install the add-in:

1 Select **Tools->Add-Ins** from the Excel main menu.

2 If the Function Wizard was previously installed, a reference to **MATLAB Function Wizard** appears in the list. Uncheck the item and click **OK**.

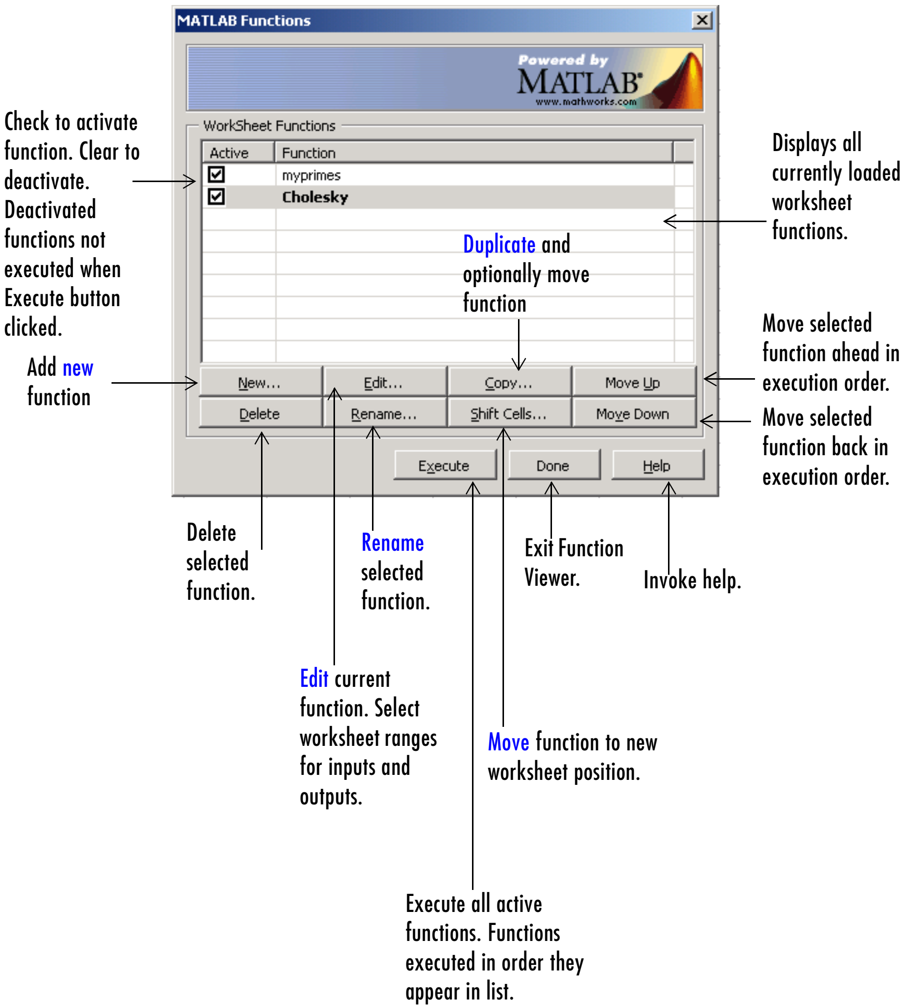
   If the Function Wizard was not previously installed, click **Browse** and proceed to the `<matlab>\toolbox\matlabxl\matlabxl` directory. Select `mlfunction.xla`. Click **OK** on this dialog box and on the preceeding one.

The Function Wizard is also packaged with all deployed components. When a component is installed onto a separate machine, the Function Wizard is placed into the top-level directory of the installed component. In this case sse the instructions above, substituting the installed component's directory.

Starting the Function Wizard

To start the Function Wizard, click on **Tools -> MATLAB Functions** on the Excel menu bar. The starting point of the Function Wizard, called the Function Viewer, now displays.

Function Viewer



The Function Viewer controls the execution of worksheet functions. Use the Function Viewer to organize the list of all currently loaded MATLAB Excel Builder functions.

Using the Function Viewer

The Function Viewer displays the names of all loaded functions. You can edit this name to provide a more descriptive identifier. A check box for each entry denotes the active/inactive state of each function. Inactive functions are not executed when you click the **Execute** button.

Below the function list is a panel of eight buttons. To add a new component to the list of loaded worksheet functions, click the **New** button. (See “Component Browser” on page 5-5).

Each of the other buttons performs a specific action on the currently selected function. To select a function, left-click the list item. The row becomes highlighted. You can change the current selection by left-clicking on a different list item or by using the up and down arrow keys on your keyboard.

Loading and Executing Functions

To load and execute a MATLAB Excel Builder function in your worksheet requires three steps:

1. Load an Excel Builder component.

   Click the **New** button on the Function Viewer to display the **Component Browser**. (See “Component Browser” on page 5-5.) Use this browser to select the component you want to load from the list of all currently installed Excel Builder components. From the selected component, add the method that you want to call.

2. Set the inputs, outputs, and other properties of your function.

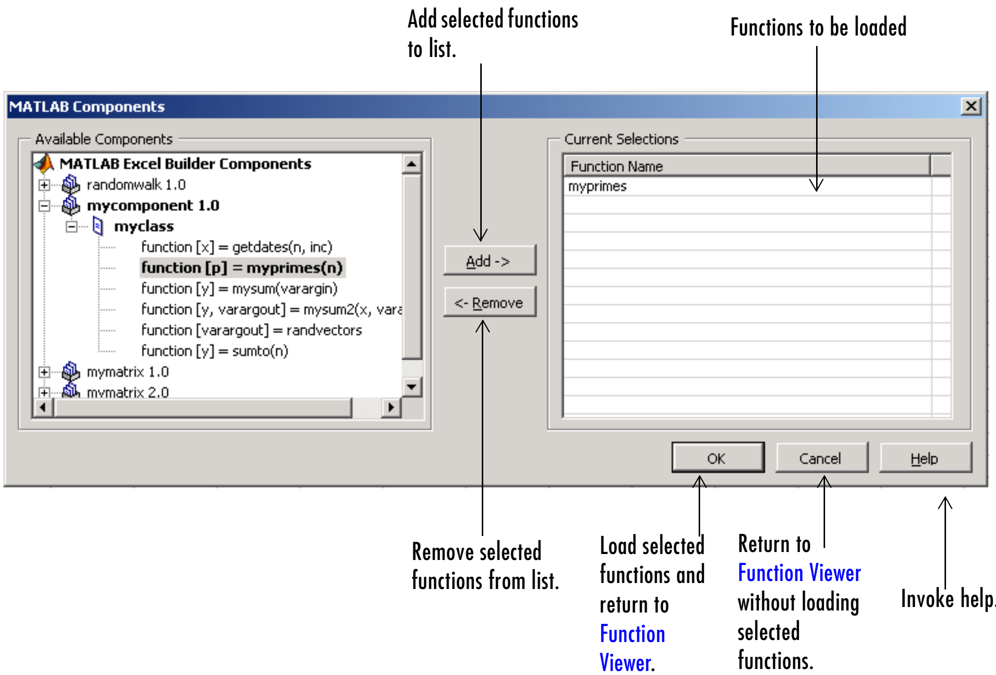
   Click the **Edit** button to display the **Function Properties** dialog box. (See “Function Properties” on page 5-6.)

3. Click the **Execute** button on the Function Viewer.

   When you click the **Execute** button, functions execute in the order displayed in the list.

Component Browser

The Component Browser lists all MATLAB Excel Builder components currently installed on the system. When you click the **New** button on the Function Viewer, this dialog box displays.



The **Component Browser** lists each component by name and version. Expanding a component reveals the class name at the next level. You can also expand the class to reveal the MATLAB functions that make up the class methods.

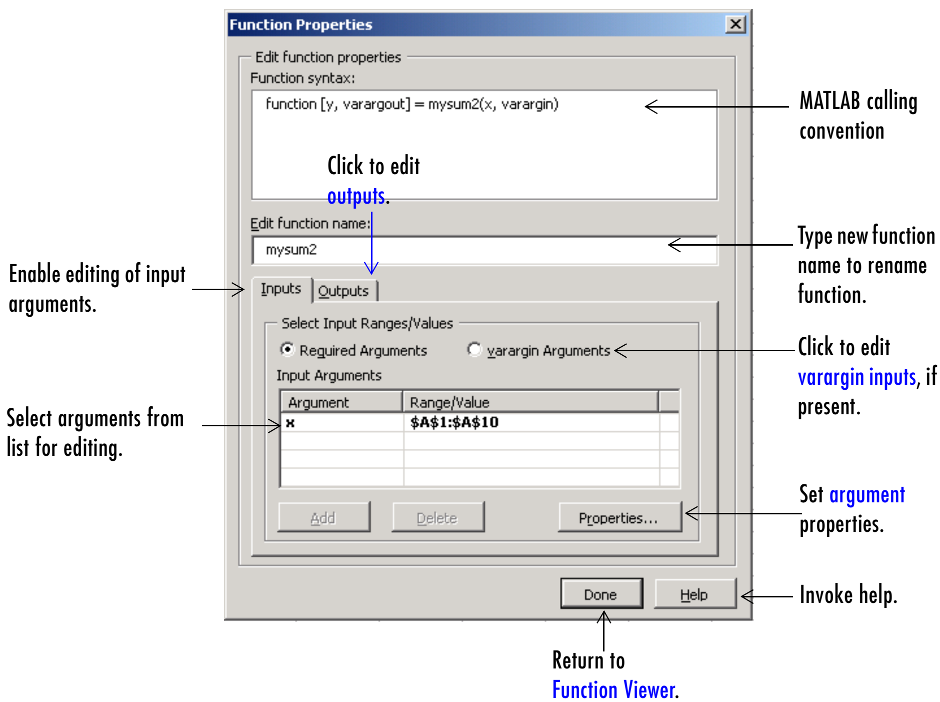
Select the desired method and click the **Add** button to add a function. To load all methods of a class, select the class name and click **Add**. Added functions appear under **Current Selections** on right of the browser. To remove a function before returning to the Function Viewer, highlight it under **Current Selections** and click the **Remove** button.

Function Properties

This group of dialog boxes sets properties and values for the inputs and outputs. You can map inputs and outputs to ranges in your worksheet. You can also rename a function with any of these dialog boxes.

When you click the **Edit** button on the Function Viewer, the dialog box below displays.

Editing Required Inputs



- The **Add** and **Delete** buttons become active when you select **varargin Arguments**.
- Select the **Outputs** tab to switch to editing outputs.

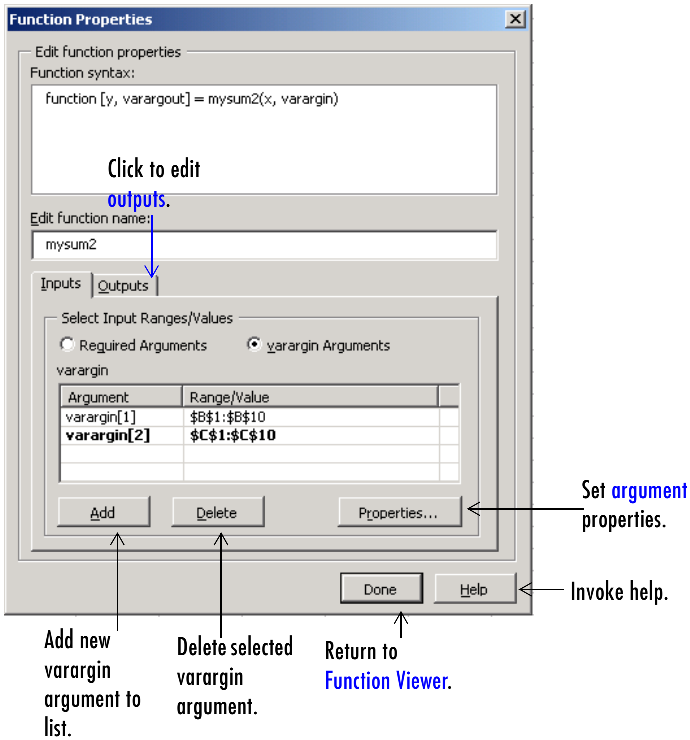
Editing Function Arguments. Function arguments may be either required arguments or `varargin`/`varargout` arguments:

- Required arguments appear first on the left or right sides of a MATLAB function and are not named `varargin` or `varargout`.
- `varargin`/`varargout` arguments always appear as the last input or output. They allow you to specify a variable number of arguments.

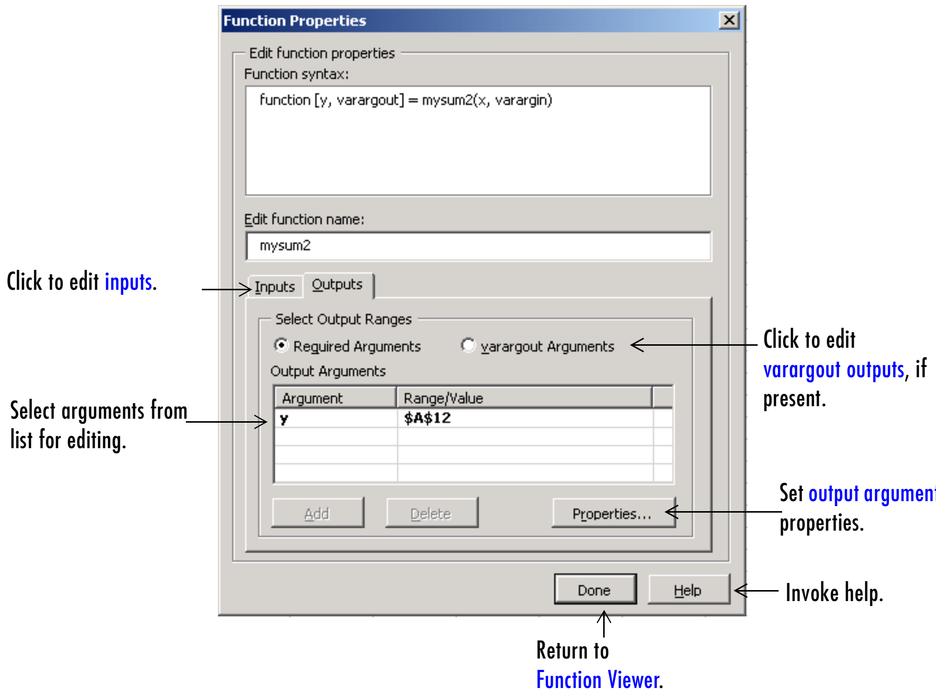
To edit required arguments, select the argument in the list and click the **Properties** button.

Before you can edit `varargin`/`varargout` arguments, you must first explicitly add them using the **Add** button. If the MATLAB function does not have `varargin`/`varargout` arguments, the ability to add arguments to the list is disabled. Once you have added `varargin`/`varargout` arguments, you can edit them in the same way as required arguments.

Editing varargin Inputs

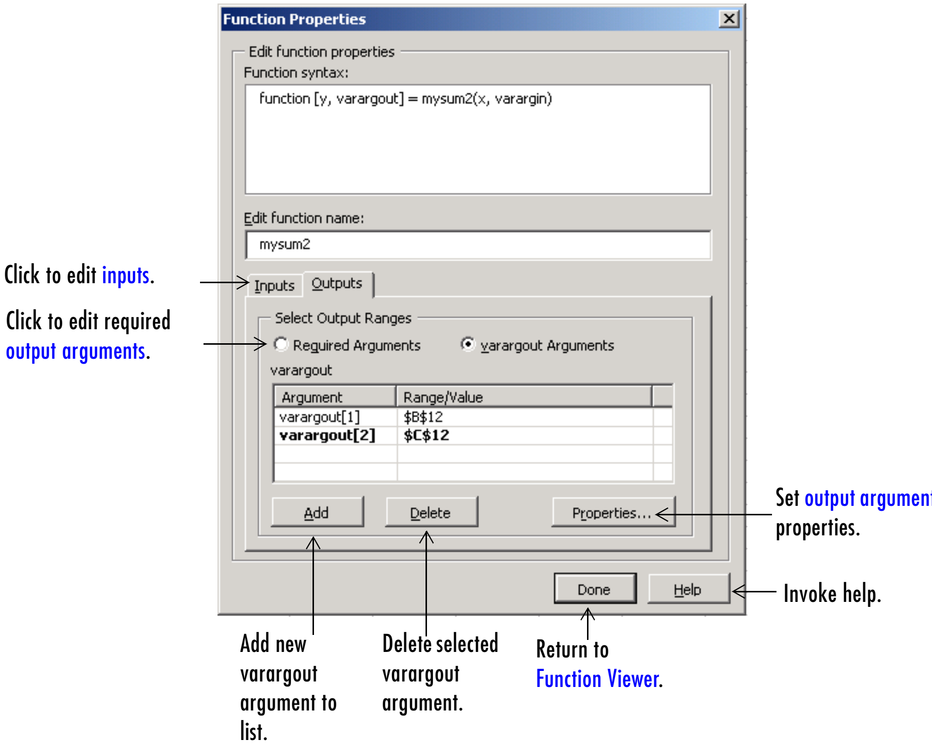


Editing Required Outputs



- The **Add** and **Delete** buttons become active when you select **varargout Arguments**.
- Select the **Inputs** tab to switch to editing inputs.

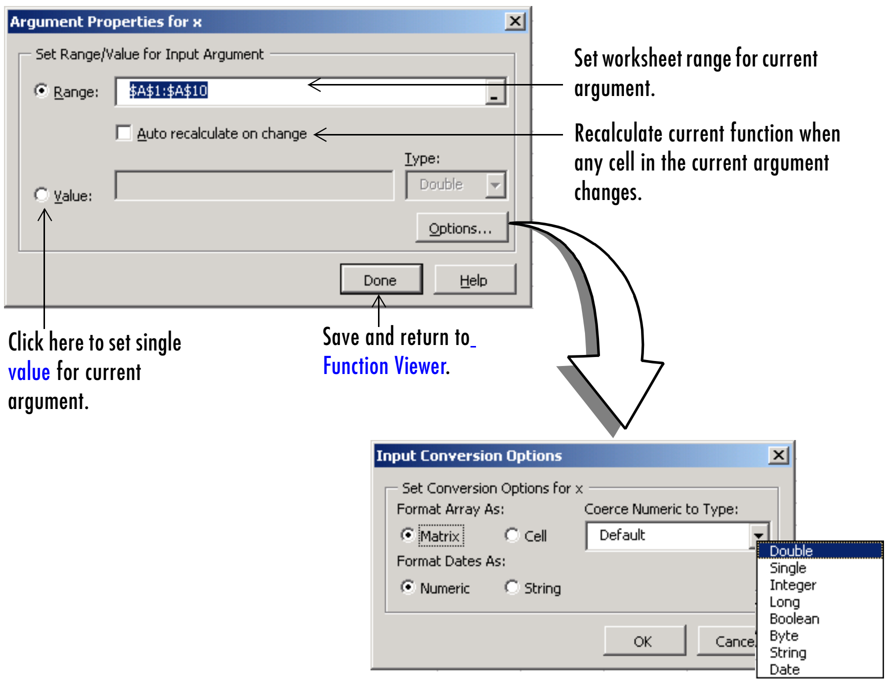
Editing varargout Outputs



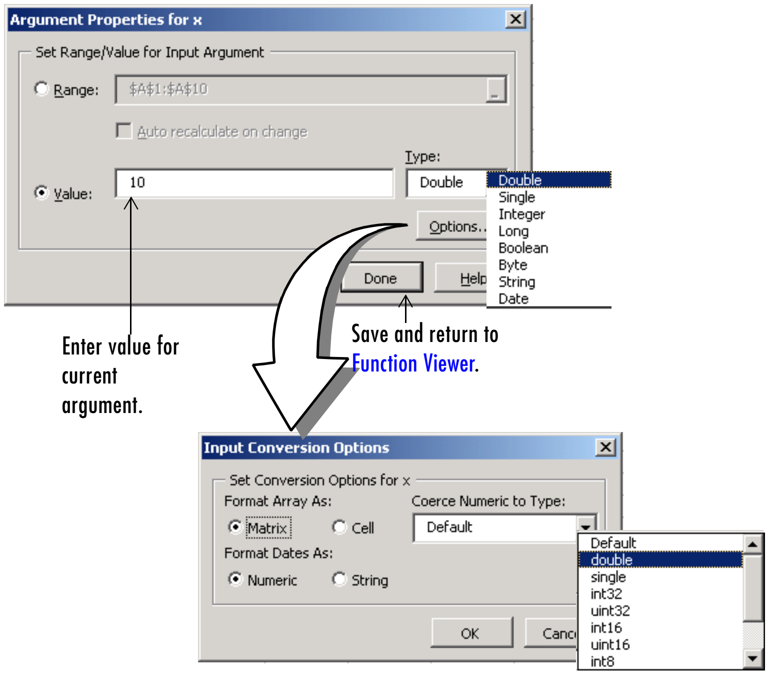
Argument Properties

The **Argument Properties** and related dialog boxes allow you to select worksheet ranges or optionally enter a specific value for an input argument.

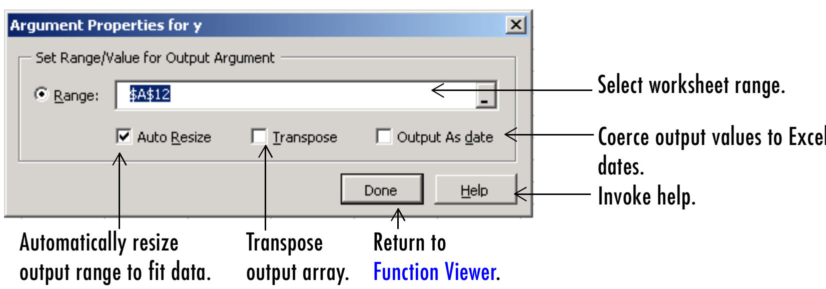
Input Argument Range



Input Argument Value



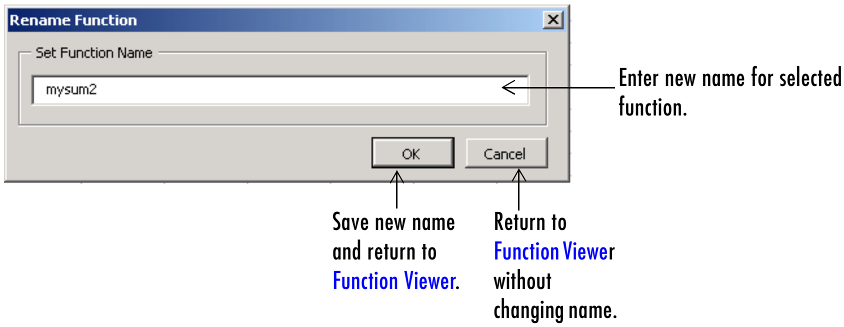
Output Argument Properties



Function Utilities

Rename Function

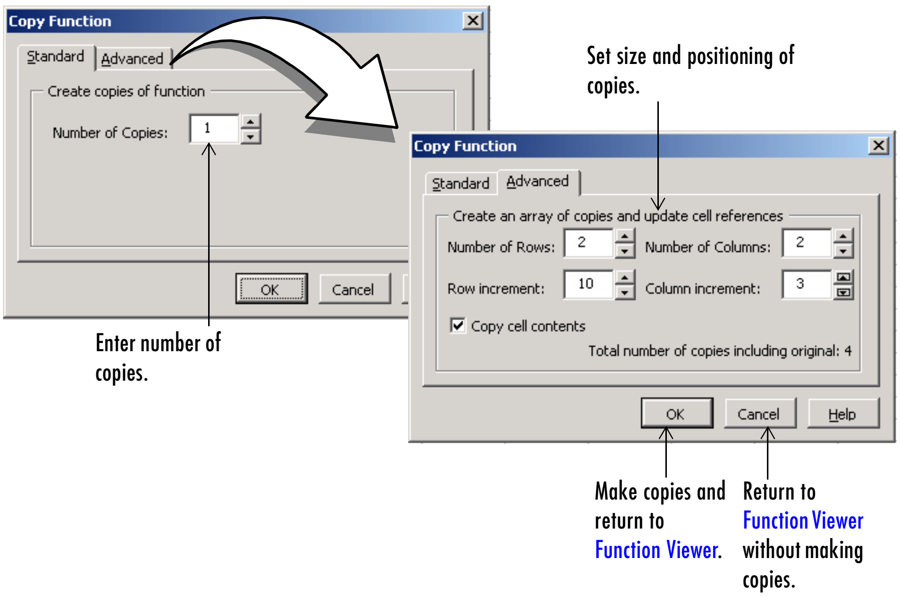
Use this dialog box to rename a function. When you click the **Rename** button on the Function Viewer, this dialog box displays.



Copy Function

Use the **Copy Function** dialog box to make copies of the current function. The **Standard Copy** tab creates a specified number of copies of the function while copying any argument/range values you have set.

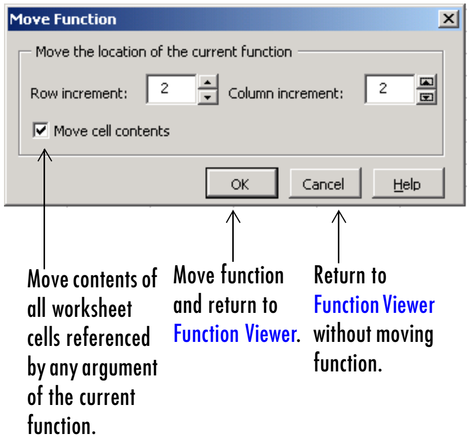
The **Advanced** tab creates a rectangular array of copies of the current function in the current worksheet, and optionally copies the cell contents of ranges referenced by the function's arguments. When you set the number of rows and columns and the row/column increments, the copy process automatically updates cell references by the specified increment amounts.



- Positive increments move rows down and columns to the right. Negative increments move rows up and columns to the left.

Move

Use the **Move Function** dialog box to move the currently selected function to a new position in the current worksheet. When you set the row and column increments, the move process automatically updates cell references by these values. You can also optionally move the cell contents of any ranges referenced by the function.



- Positive increments move rows down and columns to the right. Negative increments move rows up and columns to the left.

6

Function Reference

componentinfo

Purpose Query system registry

Syntax

```
Info = componentinfo(ComponentName, MajorRevision, MinorRevision)
```

Arguments

`ComponentName` (Optional) A MATLAB string providing the name of a MATLAB Excel Builder component. Names are case sensitive. If this argument is not supplied, the function returns information on all installed components.

`MajorRevision` (Optional) Component major revision number. If this argument is not supplied, the function returns information on all major revisions.

`MinorRevision` (Optional) Component minor revision number. Default = `0`.

Description

`Info = componentinfo(ComponentName, MajorRevision, MinorRevision)` returns registry and type information for a MATLAB Excel Builder component. `componentinfo` takes between zero and three inputs and returns an array of structures representing all the registry and type information needed to load and use the component.

When you supply a component name, `MajorRevision` and `MinorRevision` are interpreted as shown below.

Value of MajorRevision	Information Returned
`>0`	Information on a specific major and minor revision
`0`	Information on the most recent revision
`<0`	Information on all versions

If you do not supply a component name, the function returns information for all components installed on the system.

Examples

Example 1.

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

With a component name and major revision supplied, the function returns information for revision 1.0 of `mycomponent`.

Example 2.

```
Info = componentinfo('mycomponent')
```

With a component name but no major revision supplied, the function returns information for all revisions of `mycomponent`.

Example 3.

```
Info = componentinfo
```

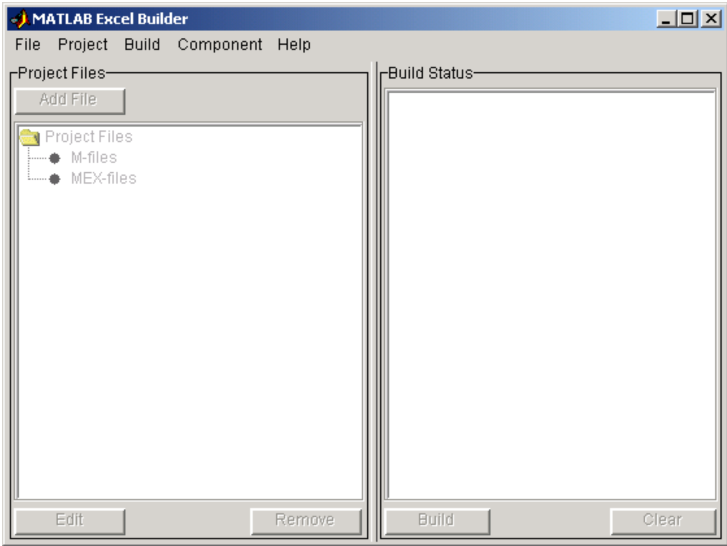
Without any arguments supplied, the function returns information for all installed components.

mxltool

Purpose Graphical user interface to MATLAB Excel Builder

Syntax `mxltool`

Description `mxltool` displays the graphical user interface (GUI) for MATLAB Excel Builder.



A

Producing a COM Object from MATLAB

Capabilities

MATLAB Excel Builder enables you to pass Microsoft Excel worksheet values to a compiled MATLAB model via Visual Basic for Applications (VBA) and to return model output to a cell or range of cells in the worksheet. Each MATLAB Excel Builder component is built as a stand-alone COM object. (COM is an acronym for Component Object Model, Microsoft's binary standard for object interoperability. COM is the widely accepted standard for integration of external functionality into Microsoft Office applications, such as Excel.) Each MATLAB function included in a given component appears as a method of the created COM class. The resulting call syntax from Visual Basic is systematically mapped to the syntax of the original MATLAB. This mapping provides an intuitive bridge from MATLAB, where the functions are created, to Visual Basic, where the functions are ultimately called.

MATLAB Excel Builder provides robust data conversion and array formatting to preserve the flexibility of MATLAB when calling from Visual Basic. Also provided is custom error processing so that errors originating from MATLAB functions are automatically manifested as Visual Basic exceptions. The information returned with the error always references the original MATLAB code, making debugging easy.

A simple versioning mechanism is also built into each component to help manage deployment of multiple versions of the same component. Figure A-1 provides an overview of the process of creating a stand-alone COM object from compiled MATLAB M-files.

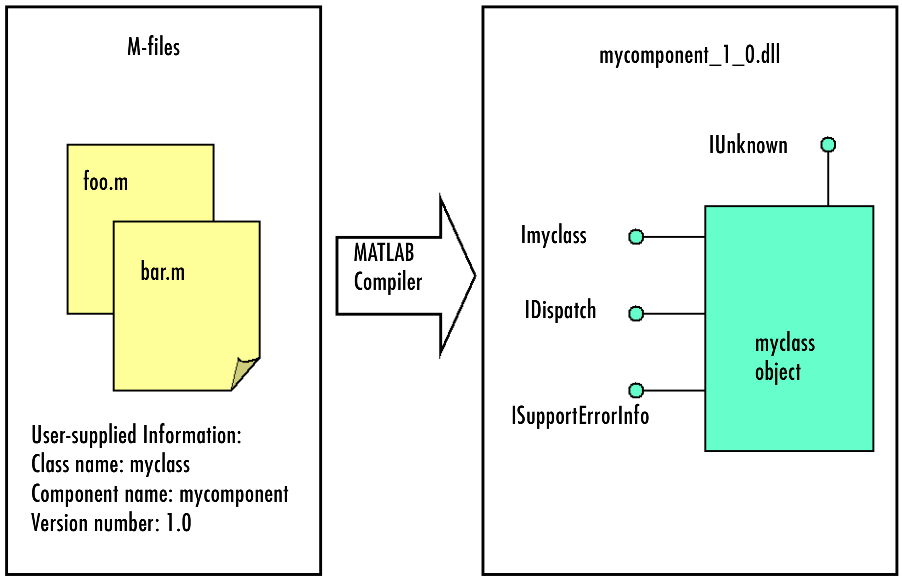


Figure A-1: Creating a Stand-Alone COM Object with the MATLAB Compiler

The process of creating a MATLAB Excel Builder component is completely automatic. The user supplies a list of M-files to process and a some additional information, i.e., the component name, the class name, and the version number. The build process that follows involves code generation, compiling, linking, and registration of the finished component.

Figure A-2 shows the files created at each step in the entire process, from compilation to registration of the final DLL.

Note If you are reading this document online, click on Steps 1 - 5 in the figure for an explanation of what takes place at each specific point in the process.

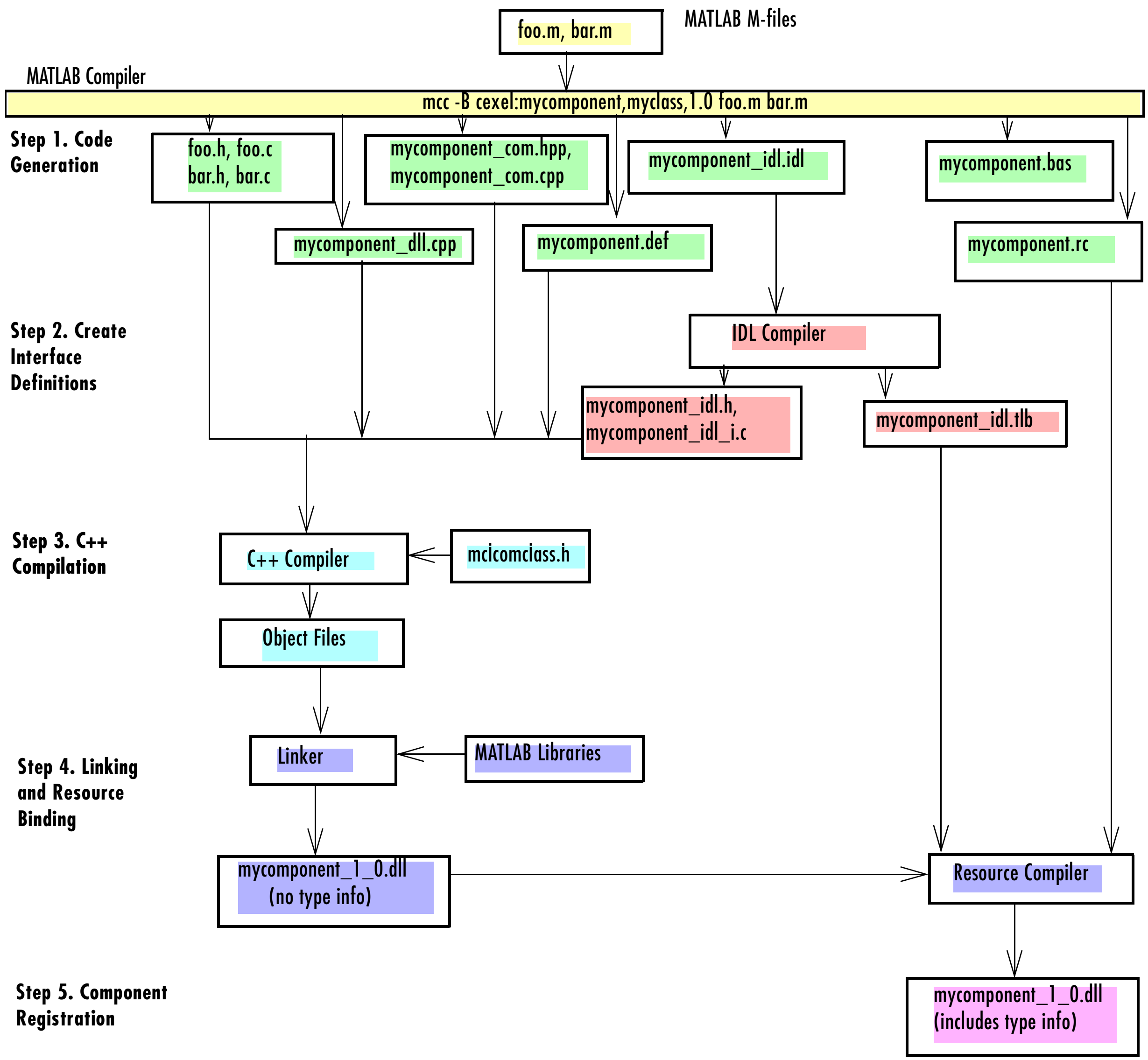


Figure A-2: M-Build Steps and Intermediate Files Created

Step 1. Code Generation

The first step in the build process generates all source code and other supporting files needed to create the component.

The compiler first produces `.c` and `.h` files (`foo.h`, `foo.c`, `bar.h`, and `bar.c`), representing the C-language translation of the M-code in the original M-files (`foo.m` and `bar.m`). 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 (`mycomponent_com.hpp` and `mycomponent_com.cpp`). In addition to these source files, the compiler generates a DLL exports file (`mycomponent.def`), a resource script (`mycomponent.rc`), and a file containing Visual Basic code (`mycomponent.bas`). This file contains VB call wrappers for each class method in the form of formula functions.

Step 2. 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.

Step 3. 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. See “Compiler Requirements” on page -x for a list of supported C++ compilers.

Step 4. 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.

Step 5. Component Registration

The final build step registers the DLL on the system. See "Component Registration" on page C-2 for information about this process.

Calling Conventions

This section describes the calling conventions for MATLAB Excel Builder components, including mappings from the original M-functions to Visual Basic. A function call originating from an Excel worksheet is routed from a Visual Basic function into a compiled M-function, as shown in Figure A-3.

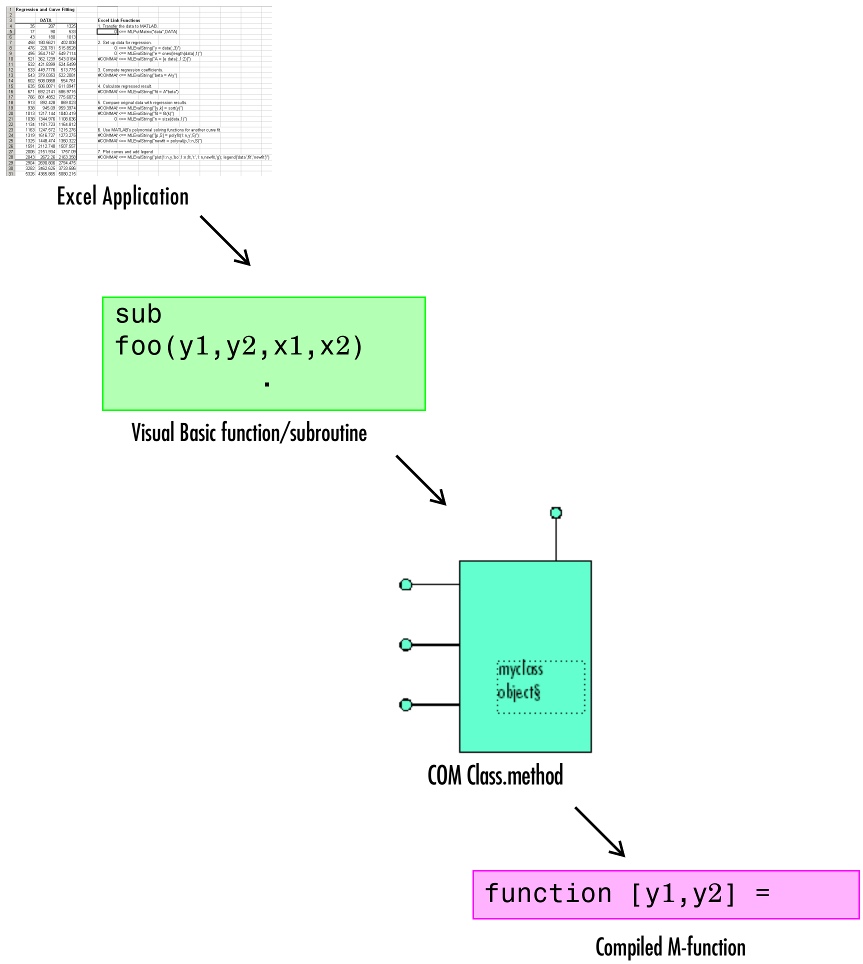


Figure A-3: Function Call Routing

Producing a COM Class

Producing a COM class requires the generation of a class definition file in Interface Description Language (IDL) as well as the associated C++ class definition/implementation files. (See the Microsoft COM documentation for a complete discussion of IDL and C++ coding rules for building COM objects.) The builder automatically produces the necessary IDL and C/C++ code to build each COM class in the component. This process is generally transparent to the user.

As a final step, the builder produces a Visual Basic function wrapper for each method, used to implement an Excel formula function. Formula functions are useful when calling a method that returns a single scalar value with one or more inputs. Use a general Visual Basic subroutine when calling a method that returns array data or multiple outputs.

IDL Mapping

The most generic MATLAB M-function is

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

This function maps directly to the following IDL signature.

```
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);
```

This 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`. (`nargout` is not produced if you compile an M-function containing no outputs.) 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. 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 B-2 lists the rules for conversion between MATLAB arrays and COM VARIANTs.

Visual Basic Mapping

The Visual Basic mapping to the IDL signature shown above is

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

(See the COM documentation for mappings to other languages, such as C++.) 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. MATLAB Excel 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 and Excel Range manipulation.

MATLAB Compiler Output

The MATLAB Excel Builder generates a default Visual Basic function wrapper for each class method with the following format:

```
Function foo(Optional X1 As Variant, _
             Optional X2 As Variant, _
             .
             .
             Optional varargin1 As Variant, _
             Optional varargin2 As Variant, _
             .
             .
             Optional varargin N As Variant) _
             As Variant
     Dim Y1, Y2, ..., varargout As Variant
     Dim varargin As Variant
      .
     (other declarations)
      .
.
     (function body)
      .
      .
     foo = Y1
      .
      .
     (error handling code)
      .
End Function
```

By default, the generated formula function contains an argument list with all the inputs to the method call and a return value corresponding to the first output parameter. The argument list includes each explicit input parameter. If the optional `varargin` parameter is present in the original MATLAB function, additional arguments `varargin1`, `varargin2,...,varargin`*n* are generated, where *n* is a number chosen by the builder. The number *n* is chosen so that the total number of inputs is less than or equal to 32. This function generally includes a declaration for each output parameter as type `Variant`. If the original MATLAB function contains a `varargin`, a variable is declared of type

`Variant` to pass collectively the `varargin1,...,varargin`*n* parameters in the form of a `Variant` array. The main function body contains code for:

- Packing `varargin` parameters if available
- Creating the necessary class instance
- Calling the target method
- Error handling

B

Data Conversion

Data Conversion Rules

This section describes the data conversion rules for MATLAB Excel Builder components. Excel Builder components are dual interface COM objects that support COM Automation compatible data types. When a method is invoked on a Excel 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.

Table B-1 lists the `VARIANT` type codes supported by Excel Builder components. Table B-2 and Table B-3 list the data conversion rules between COM `VARIANT`s and MATLAB arrays.

Table B-1: 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

Table B-1: VARIANT Type Codes Supported (Continued)

VARIANT Type Code (C/C++)	C/C++ Type	Variant Type Code (Visual Basic)	Visual Basic Type	Definition
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
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

Table B-1: VARIANT Type Codes Supported (Continued)

VARIANT Type Code (C/C++)	C/C++ Type	Variant Type Code (Visual Basic)	Visual Basic Type	Definition
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)
<*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++.

Table B-2: 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.	
`structure`	`VT_DISPATCH`	`VT_DISPATCH`	A MATLAB `struct` array is converted to an `MWStruct` object. (See"Class MWStruct" on page D-16.) This object is passed as a `VT_DISPATCH` type.

Table B-2: MATLAB to COM VARIANT Conversion Rules (Continued)

MATLAB Data Type	VARIANT type for Scalar Data	VARIANT type for Array Data	Comments
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 D-25.) This object is passed as a VT_DISPATCH type.

Table B-2: 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 D-23.)
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. See "Class MWComplex" on page D-23.)
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. See "Class MWComplex" on page D-23.)

Table B-2: 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. See "Class MWComplex" on page D-23.)
`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. See "Class MWComplex" on page D-23.)
`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. See "Class MWComplex" on page D-23.)

Table B-2: 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. See "Class MWComplex" on page D-23.)
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. See "Class MWComplex" on page D-23.)
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	

Table B-3: 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	
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	

Table B-3: COM VARIANT to MATLAB Conversion Rules (Continued)

VARIANT Type	MATLAB Data Type (scalar or array data)	Comments
VT_DATE	double	1. 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. 2. VARIANT dates can be optionally converted to strings. See "Data Conversion Flags" on page B-14 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 Excel Builder types MWStruct, MWComplex, MWSparse, and MWArg. See "Utility Library Classes" on page D-3 for information on Excel Builder types.

Table B-3: 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

MATLAB Excel 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 Table B-2 and Table B-3. In some cases this is not possible, e.g., when existing MATLAB code is used in conjunction with a third-party product like Excel.

Table B-4 shows the array formatting flags.

Table B-4: 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`, `VARIANT`s 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 `VARIANT`s, 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.
`OutputArrayIndFlag`	(Applies to nested cell arrays only.) Output array indirection level used with the `OutputArrayFormat` flag. This flag works exactly like `InputArrayIndFlag`.

Table B-4: Array Formatting Flags (Continued)

Flag	Description
`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 an Excel Builder component from Excel where the MATLAB function returns outputs as row vectors, and you want the data in columns.

Data Conversion Flags

MATLAB Excel Builder components contain flags to control the conversion of certain `VARIANT` types to MATLAB types.

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 Table B-3.

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 Table B-3. `mwDateFormatString` 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 Excel Builder components. To process dates with such code, set this property to 0.

C

Registration and Versioning

Component Registration

When the MATLAB Excel Builder 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

MATLAB Excel 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 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.

An Excel 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.

Globally Unique Identifiers

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. The MATLAB Compiler 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, and COM-related information is stored under a top-level key called `HKEY_CLASSES_ROOT`. Under `HKEY_CLASSES_ROOT` are several other keys under which the component writes its information. These keys are defined in Table C-1.

Table C-1: Keys

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.
`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 form `<component-name>.<class-name>` and `<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 Excel Builder 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 (default = 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, the compiler 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 the compiler 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 the compiler does not find an existing component of the specified name, it generates new GUIDs for the component's type library, class, and interface.

Obtaining Registry Information

MATLAB Excel Builder includes the MATLAB function `componentinfo` to query the system registry for any installed Excel Builder components. The function can be executed inside MATLAB with the component name, major version number, and minor version number as arguments. It returns an array of structures with the requested information. Calling `componentinfo` with no arguments returns all Excel Builder components installed on the machine.

The next 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`.

Note Although properties and events may appear in `componentinfo` output fields, Excel Builder components currently do not support them.

```
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. These fields are defined in Table C-2.

Table C-2: 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 MATLAB Excel Builder components have the type library bound into the DLL, this filename is the same as the DLL name and path.

Table C-2: Registry Information Returned by componentinfo (Continued)

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

Table C-2: Registry Information Returned by componentinfo (Continued)

Field	Description

`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 Visual Basic 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 Visual Basic function prototypes

D

Utility Library

This section describes the `MWComUtil` library provided with the MATLAB Excel Builder. 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 Excel 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 D-3) and three enumerated types (see "Enumerations" on page D-30). 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 Visual Basic 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 Excel Builder Utility Library provides several classes:

- “Class MWUtil” on page D-3
- “Class MWFlags” on page D-9
- “Class MWStruct” on page D-16
- “Class MWField” on page D-22
- “Class MWComplex” on page D-23
- “Class MWSparse” on page D-25
- “Class MWArg” on page D-28

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 D-3
- “Sub MWPack(pVarArg, [Var0], [Var1], … ,[Var31])” on page D-4
- “Sub MWUnpack(VarArg, [nStartAt As Long], [bAutoResize As Boolean = False], [pVar0], [pVar1], ..., [pVar31])” on page D-6
- “Sub MWDate2VariantDate(pVar)” on page D-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 Excel Builder components. An error is generated if a method call is made to a member class of any Excel 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`.

Argument	Type	Description
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 (See “Data Conversion Rules” on page B-2 for more information on conversion between MATLAB and COM date values). 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 B-2 for more information on conversion between MATLAB and COM Automation types). All MATLAB Excel 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 D-10
- "Property DataConversionFlags As MWDataConversionFlags" on page D-13
- "Sub Clone(ppFlags As MWFlags)" on page D-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 D-10
- "Property InputArrayIndFlag As Long" on page D-11
- "Property OutputArrayFormat As mwArrayFormat" on page D-11
- "Property OutputArrayIndFlag As Long" on page D-12
- "Property AutoResizeOutput As Boolean" on page D-12
- "Property TransposeOutput As Boolean" on page D-12

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

Table D-1: Array Formatting Rules for Input Arrays

Value	Behavior
`mwArrayFormatAsIs`	Converts arrays according to the default conversion rules listed in Table B-3, COM VARIANT to MATLAB Conversion Rules, on page B-10.
`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.
`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.

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 Excel Builder class methods. The default value is `mwArrayFormatAsIs`. The behaviors indicated by this flag are listed in the next table.

Table D-2: Array Formatting Rules for Output Arrays

Value	Behavior
`mwArrayFormatAsIs`	Converts arrays according to the default conversion rules listed in Table B-2, MATLAB to COM VARIANT Conversion Rules, on page B-5.
`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 Excel 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 D-13
- "Property InputDateFormat As mwDateFormat" on page D-13
- "PropertyOutputAsDate As Boolean" on page D-15
- "PropertyDateBias As Long" on page D-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 B-10.

Property InputDateFormat As mwDateFormat. This property converts dates passed as input parameters to method calls on Excel Builder classes. The default value is `mwDateFormatNumeric`. The behaviors indicated by this flag are shown in Table D-3, Conversion Rules for Input Dates.

Table D-3: Conversion Rules for Input Dates

Value	Behavior
`mwDateFormatNumeric`	Convert dates to numeric values as indicated by the rule listed in Table B-3, COM VARIANT to MATLAB Conversion Rules, on page B-10.
`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 Excel 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 D-16
- "Property Item([i0], [i1], ..., [i31]) As MWField" on page D-17
- "Property NumberOfFields As Long" on page D-20
- "Property NumberOfDims As Long" on page D-20
- "Property Dims As Variant" on page D-20
- "Property FieldNames As Variant" on page D-20
- "Sub Clone(ppStruct As MWStruct)" on page D-21

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 D-22
- "Property Value As Variant" on page D-23
- "Property MWFlags As MWFlags" on page D-23
- "Sub Clone(ppField As MWField)" on page D-23

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 method's 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

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 method's 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 D-26
- "Property NumColumns As Long" on page D-26
- "Property RowIndex As Variant" on page D-26
- "Property ColumnIndex As Variant" on page D-26
- "Property Array As Variant" on page D-26
- "Property MWFlags As MWFlags" on page D-26
- "Sub Clone(ppSparse As MWSparse)" on page D-27

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 the any flags set on the object whose method's 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 D-29
- "Property MWFlags As MWFlags" on page D-29
- "Sub Clone(ppArg As MWArg)" on page D-29

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 the any flags set on the object whose method's 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 Excel Builder Utility Library provides three enumerations (sets of constants):

- "Enum mwArrayFormat" on page D-30
- "Enum mwDataType" on page D-30
- "Enum mwDateFormat" on page D-31

Enum mwArrayFormat

The `mwArrayFormat` enumeration is a set of constants that denote an array formatting rule for data conversion. Table D-4 lists the members of this enumeration.

Table D-4: 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. Table D-5 lists the members of this enumeration.

Table D-5: mwDataType Values

Constant	Numeric Value	MATLAB Type
`mwTypeDefault`	0	`N/A`
`mwTypeLogical`	3	`logical`

Table D-5: mwDataType Values (Continued)

Constant	Numeric Value	MATLAB Type
`mwTypeChar`	4	`char`
`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. Table D-6 lists the members of this enumeration.

Table D-6: mwDateFormat Values

Constant	Numeric Value	Description
`mwDateFormatNumeric`	0	Format dates as numeric values.
`mwDateFormatString`	1	Format dates as strings.

E

Troubleshooting

This section provides a table showing errors you may encounter using MATLAB Excel Builder, probable causes for these errors, and suggested solutions.

Table E-1: MATLAB Excel Builder Errors 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`*`.1_0: Error getting data conversion flags.`	Usually caused by `mwcomutil.dll` not being registered.	Open a DOS window, change directories to `<matlab>\bin\win32` (`<matlab>` represents the location of MATLAB on your system), and run the command `mwregsvr mwcomutil.dll`.
`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, open a DOS window, change directories to `<projectdir>\distrib` (`<projectdir>` represents the location of your project files), and run the command: `mwregsvr <projectdll>.dll`.
`Error in VBAProject: Automation error The specified module could not be found.`	This usually occurs if MATLAB is not on the system path.	Place `<matlab>\bin\win32` on your path.

Table E-1: MATLAB Excel Builder Errors and Suggested Solutions (Continued)

Message	Probable Cause	Suggested Solution
`LoadLibrary("`*`component_name`*`_1_0.dll") failed - The specified module could not be found.`	You may get this error message while registering the project DLL from the DOS prompt. This usually occurs if MATLAB is not on the system path.	Place `<matlab>\bin\win32` on your path.
`Cannot recompile the M file` *`xxxx`* `because it is already in the library libmmfile.mlib.`	The name you have chosen for your M-file duplicates the name of an M-file already in the library of precompiled M-files.	Rename the M-file, choosing a name that does not duplicate the name of an M-file already in the library of precompiled M-files.

Table E-2: Excel Errors and Suggested Solutions

Message	Probable Cause	Suggested Solution
`The Macros in this project are disabled. Please refer to the online help or documentation of the host application to determine how to enable macros.` Note: Wording may vary depending upon the version of Excel you are running.	The macro security for Excel is set to **High**.	Set Excel macro security to **Medium** on the **Security Level** tab (**Tools > Macro > Security**).

Table E-3: Function Wizard Problems

Problem	Probable Cause	Suggested Solution
The Function Wizard Help does not display.	The Function Wizard Help file (`mlfunction.chm`) is not in the same directory as the Function Wizard add-in (`mlfunction.xla`).	Copy the Help file (`mlfunction.chm`) into the same directory as the add-in.

Index